Учебник по BlackBox

Документ еще нуждается в редактировании. Нужно время. --Ф.В.Ткачев, 2009-10-09

Оглавление

Часть I: Подходы к проектированию

В первой части руководства по системе Блэкбокс дано введение в проектные схемы, которые используются в BlackBox. Знание этих схем облегчает понимание и запоминание более детальных проектных решений в различных модулях BlackBox.

Вторая часть руководства показывает, как используются самые важные библиотечные компоненты: формы, элементы управления и текстовые компоненты.

В третьей части руководства с помощью примеров нарастающей сложности показывается, как создавать новые объекты изображения (вьюшки).

1 Взаимодействие с пользователем

В этом разделе мы обсудим, как пользовательский интерфейс можно сделать дружественным к пользователю. Графические пользовательские интерфейсы — только часть ответа. Ведь здесь сразу встает другая проблема: как реализовать подобный интерфейс с наименьшими усилиями? Подход к проектированию, который решает эти проблемы, удивительно фундаментальный и имеет глубокое влияние на способы создания современного ПО. Он называется объектно-ориентированным программированием.

1.1 Дружественность к пользователю

Сегодня в мире используется не менее 100 миллионов персональных компьютеров. При таком большом числе пользователей, которые не являются специалистами по компьютерам, важна легкость использования компьютеров. Это подразумевает, что операционная система и приложения должны быть дружелюбными к пользователю. Что это значит? Здесь есть несколько аспектов.

Во-первых, различные приложения должны быть согласованы друг с другом: функция, которая доступна в различных программах должна иметь сходный пользовательский интерфейс во всех этих программах, так, чтобы пользователю не нужно было учиться делать различными способами одно и то же. Первый способ — опубликовать правила построения пользовательского интерфейса, которых должны придерживаться все разработчики ПО. Такой подход был впервые применен компанией Apple в системе Macintosh. Второй подход — реализовать функцию только один раз, чтобы ее можно многократно использовать в различных приложениях. Это идея компонентного ПО. Разработчики Apple так далеко не пошли, но они по крайнем мере дали много стандартных процедур в довольно богатой инструментальной библиотеке Toolbox, встроенной в Mac OS.

Во-вторых, метафора рабочего стола, где значки графически представляют файлы, каталоги, программы и другие сущности, позволила заменить труднозапоминаемые команды, которые нужно набирать с клавиатуры, более интуитивными прямыми действиями: например, значок с помощью мыши может быть брошен в мусорную корзину вместо набора с клавиатуры команды вроде "erase myfile". Подобный богатый пользовательский интерфейс требует обширной функциональности. Не удивительно, что инструментальная библиотека Toolbox Mac OS включает процедуры поддержки для работы с окнами, меню, мышью, диалоговыми окнами, графикой и др. Toolbox делает возможным построение приложений, которые действительно придерживаются обнародованных правил построения пользовательского интерфейса. К сожалению, программирование в такой среде означает огромный скачок в сложности. В результате Macintosh оказался гораздо проще в использовании, но и гораздо труднее для программирования.

Сегодня метафора проводника — все более популярное добавление к традиционным графическим пользовательским интерфейсам. Вместе с доступом к Интернету она открывает пользователю компьютера целую вселенную документов — и еще больше усложняет работу разработчика.

Наконец, один из наиболее общих и важных аспектов дружелюбности к пользователю - это исключение модальных диалогов [modes]. Модальный пользовательский интерфейс разделяет взаимодействие с пользователем на различные классы и дает свой контекст для каждого класса. Переключение между контекстами громоздко. Например, в программе для работы с базой данных пользователь может открыть окно для ввода данных. В этом окне нельзя удалить записи, осуществить поиск записей, сделать примечания, прочитать электронную почту и т.д. Для каждой из этих задач следует вначале покинуть окружение ввода данных. Это громоздко и напоминает способ вычислений, ориентированный более на приложение, чем на документ. При работе со сложными программами часто трудно уследить, в каком контексте вы сейчас находитесь, какие команды здесь доступны и куда вы можете пойти далее.

Рисунок 1-1 иллюстрирует пример состояний и возможные переходы состояний гипотетического модального приложения.

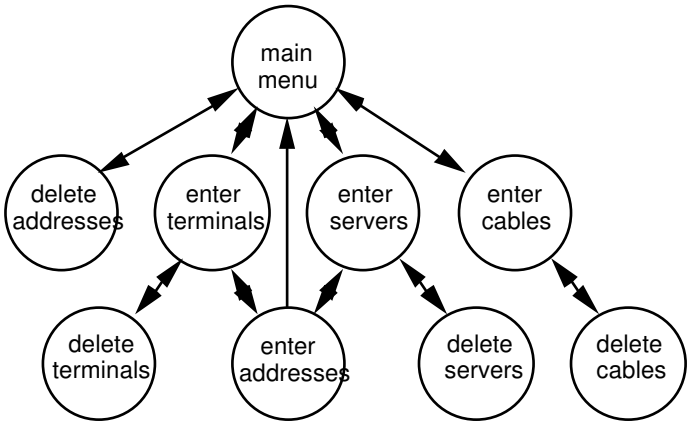

Рисунок 1-1. Возможности навигаации в модальном пользовательском интерфейсе.

Немодальные пользовательские интерфейсы не имеют раздельных рабочих контекстов [working environment] или, по крайней мере, позволяют вам переключаться между контекстами без потери состояний других. Например, если по ходу ввода данных вы захотите взглянуть на что-то с помощью браузера библиотек или утилиты текстового поиска, вы не потеряете часть данных, которую уже ввели. Если нет раздельных рабочих контекстов, то навигация становится простой и очевидной, если система дает информацию пользователю о текущем состоянии контекста и его возможностях.

Современный графический пользовательский интерфейс — основной пример преимущественно немодального пользовательского интерфейса. Здесь одновременно могут быть открыты несколько окон (контекстов), но хорошо видно, с каким окном вы сейчас взаимодействуете (это обычно "верхнее" окно). Вы можете, работая в одном окне, временно поработать в другом (тем самым сделав его "верхним"), и затем переключится обратно, чтобы продолжить работу в первом окне.

Эти пользовательские интерфейсы — огромное улучшение по сравнению со старомодными модальными интерфейсами. Пользователи чувствуют себя более комфортно, потому что им понятнее, где они находятся и что могут делать. Тем не менее, даже современные ГПИ далеки от совершенства. Большинство из них все еще используют модальные диалоговые окна, т.е. окна, которые вы не можете оставить, чтобы временно поработать в других. Например, типичное модальное диалоговое окно позволяет открыть файл, но не позволяет непосредственно переключиться на утилиту поиска файлов и обратно. Диалоговое окно "связывает" пользователя. Даже необходимость перевести окно наверх, прежде чем можно будет работать с ним, создает неудобную и не необходимую модальность, по крайней мере если есть достаточно большой экран, чтобы можно было расположить все нужные окна без перекрытий. И многие программы для работы с базами данных не разрешают держать

открытыми одновременно несколько форм ввода данных или форм для манипуляций с данными. Веб браузер ближе всего сегодня к истинно немодальному пользовательскому интерфейсу: если вы ввели неполные данные в HTML-форму и затем решили переключится на другую веб страницу, то браузер не помешает вам это сделать.

Компонентный каркас Блэкбокс более радикален, чем остальные инструменты, в том, что модальные окна в нём просто не поддерживаются. Все диалоговые окна немодальные, за исключением очень малого числа встроенных в операционную систему, таких как стандартное окно открытия файлов. В общем случае, вы можете всегда оставить диалоговое окно и вернуться к нему позже.

1.2 Циклы обработки сообщений

Реализация немодальной программы выглядит обманчиво просто: программа ожидает пользовательские события, такие как щелчок мышкой или нажатие клавиши, реагирует подходящим образом, ожидает следующего события и т.д. Такой стиль программирования с центральным циклом, который опрашивает операционную систему о событиях и затем вызывает подходящие обработчики для них, называется программированием, управляемым событиями.

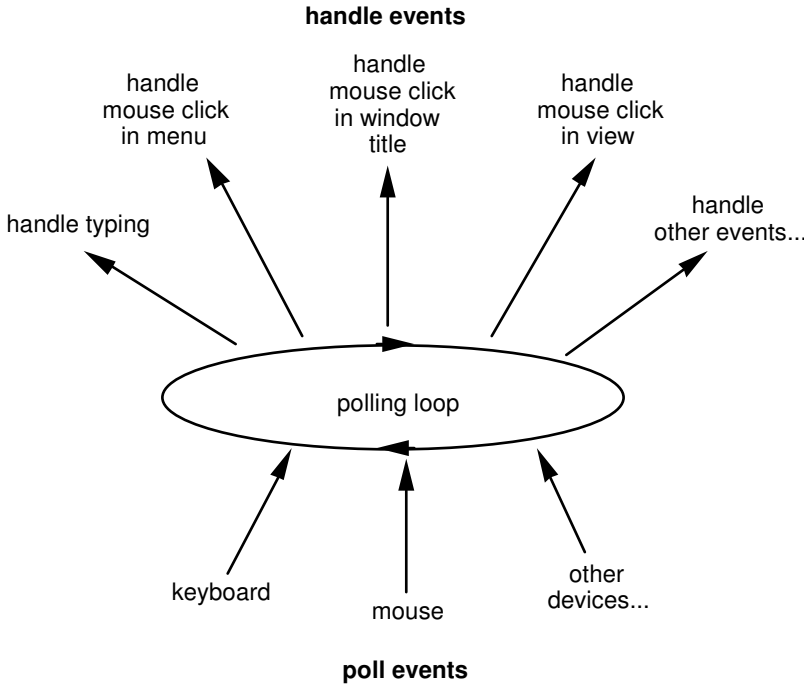

Рисунок 1-2. Цикл обработки событий для немодальной программы

Цикл опроса событий типичной программы, управляемой событиями, выглядит подобно следующему программному фрагменту:

```
PROCEDURE Main;
  VAR event: OS.Event;
BEGIN
  LOOP
    event := OS.NextEvent();   (* запросить у ОС информацию о следующем событии *)
    IF event IS KeyDownEvent THEN
      HandleKeyDown(event)
    ELSIF event IS MouseDownEvent THEN
      IF MouseInMenu(event) THEN
        HandleMenuEvent(event)
```

```
      ELSIF MouseInWindowTitle(event) THEN
        HandleTitleEvent(event)
      ELSIF MouseInWindowBorder(event) THEN
        HandleBorderEvent(event)
      ELSIF MouseInWindowContents(event) THEN
        HandleContentsEvent(event)
      ELSIF...
        ...
      END
    ELSIF ...
      ...
    END
  END Main;
```

Листинг 1-3. Цикл обработки событий с каскадным выбором обработчиков событий.

В реальной программе такие каскады условных операторов могут запросто занимать несколько страниц листинга, и они всегда похожи. Таким образом, напрашивается мысль заложить этот код в библиотеку (или, еще лучше, в операционную систему). В такой библиотеке можно было бы реализовывать стандартные правила построения пользовательского интерфейса. Например, процедура *HandleTitleEvent* различала бы, в каком именно месте заголовка окна произошел щелчок мышкой. В зависимости от точного положения окно будет перемещено на другое место, увеличено, закрыто и т.п. Это — универсальное поведение, которое может и для единообразия должно быть реализовано единожды. Однако процедуры типа *HandleContentsEvent* имеют иной характер: стандартная библиотека не может знать, что должно произойти, когда пользователь щелкнул внутри окна.

Если мы требуем, чтобы сама библиотека никогда не нуждалась в адаптации к конкретному приложению, то получается система со специфической структурой, в которой библиотека иногда вызывает процедуры приложения. Такой вызов называется обратным [call-back]. Если мы изобразим приложение находящимся над библиотекой, то станет очевидным, почему такие вызовы также называются отложенными вызовами [up-call], а полученный стиль программирования — "перевернутым" программированием [inverted programming].

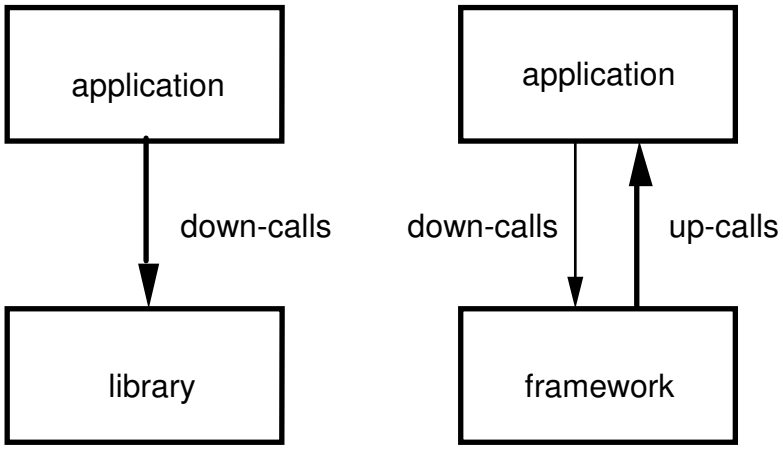

Рисунок 1-4. Образец проектирования "перевернутого" программирования.

Библиотека, которая систематически основана на "перевернутом" программировании, называется *каркасом* [framework]. Это полузаконченный продукт, который должен достраиваться с помощью динамического подключения процедур. Для некоторых процедур это может быть необязательным: процедура *HandleTitleEvent* может иметь реализацию по

умолчанию, в которой реализованы стандартные правила построения пользовательского интерфейса. Такие процедуры могут быть заменены, только если возникнет нужда в нестандартном поведении.

Опыт показывает, что процедура, подключенная к каркасу, многократно обращается к одим и тем же данным. Например, процедура *HandleTitleEvent* будет часто обращаться к окну, в котором пользователь щелкнул мышкой. Поэтому удобно "упаковать" процедуру и соответствующие данные (ее состояние) в капсулу. Такая капсула называется *объектом*, и тем самым "перевернутое" программирование развивается до *объектно-ориентированного программирования*.

Итак, рассуждая о программах, дружелюбных к пользователю, мы пришли к задаче уменьшения объема повторяющегося и сложного кода в циклах обработки событий, которая в свою очередь привела нас к стилю "перевернутого" программирования; к каркасам, как способу реализации этого стиля программирования в коде; и к объектно-ориентированному программированию как удобному средству реализации каркасов.

Система Блэкбокс скрывает цикл обработки событий от программистов прикладных программ. Он даже идет дальше каркасов-предшественников в том, что скрывает и платформо-зависимые черты, такие как окна и меню. Этого удалось добиться, сосредоточившись на абстракциях, которые представляют содержимое окна: так называемый тип *View*. Мы расскажем об этом более подробно в Главе 2.

1.3 Многопользовательский доступ

Благодаря персональным компьютерам пользователи перестали зависеть от централизованных отделов ИТ (информационных технологий), их бюрократизма и перегруженных серверов, работающих в режиме разделения времени. Эта независимость — хорошая вещь, если она может быть соединена с интеграцией там, где это полезно и выгодно. Локальные сети, глобальные сети и затем Интернет сделали интеграцию возможной. Когда два компьютера взаимодействует через сеть, один из них запрашивает другой о предоставлении некоего сервиса, например, о передаче файла через сеть. Компьютер, который выдает запросы, называют *клиентом*, другой — *сервером*. Одна и та же машина может функционировать и как клиент, и как сервер, но эти роли часто фиксированы: клерк в филиале банка всегда использует компьютер-клиент, а большая стойка в кондиционируемом хранилище штаб-квартиры банка всегда функционирует как сервер. Внутренняя сеть банка может соединять тысячи клиентов и дюжины серверов. Но даже самая маленькая сеть, очевидно, требует разделения приложений на две части: клиентскую и серверную. Поэтому такой тип архитектуры называют *клиент/серверными* вычислениями. Он позволяет распределить разные задачи между наиболее подходящими машинами. На уровне предприятия наиболее популярное разпределение ролей — возложить обязанности обслуживания централизованной базы данных на *сервер базы данных*, а оставшуюся функциональность — на машины-клиенты. Это называется *двухуровневой* архитектурой. Она требует *толстых клиентов*, т.е. машины-клиенты должны выполнять все, кроме собственно доступа к базе данных. Чтобы уменьшить эту нагрузку, наиболее развитые системы управления базами данных позволяют выполнять некоторый код на самом сервере как интерпретируемые *хранимые процедуры*. Это может существенно увеличить производительность за счет уменьшения объема данных, пересылаемых по сети в обоих направлениях.

Если количество клиентов большое, то становится трудно вовремя обновлять и поддерживать согласованность программного обеспечения на всех машинах-клиентах. Данная проблема может быть уменьшена с помощью *трехуровневой* архитектуры, где между клиентами и серверами находятся специальные *серверы приложений*. Функции "тонких" клиентов сокращены до реализации пользовательских интерфейсов, серверы баз данных управляют базами данных, а серверы приложений реализуют все знание о конкретных приложениях ("бизнес-логика").

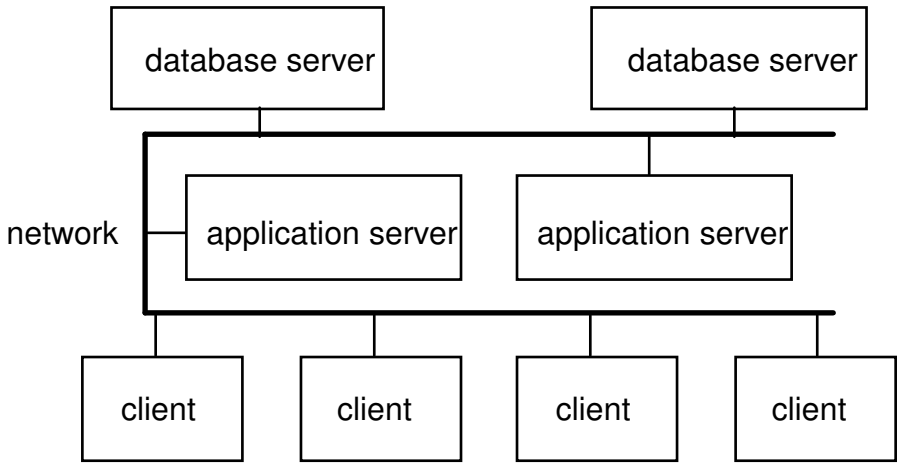

Рисунок 1-5. Проектная схема трехуровневой клиент/серверной архитектуры.

Трехуровневая архитектура неплохо управляема и масштабируема по размеру. Заметим, что программная система, разделенная таким путем, может быть в крайнем случае "ужата" за счет размещения клиента, сервера приложений и сервера баз данных на одной машине. Обратное неверно: монолитную программу нелегко разделить в соответствии с архитектурой клиент/сервер.

Клиенты и серверы соединены через сеть. Связь осуществляется либо на низком уровне абстракции нетипизированных потоков байтов (например, используя интерфейс сокетов для TCP/IP коммуникации), либо на высоком уровне абстракции типизированных данных. В последнем случае используются либо распределенные объекты (для непосредственной коммуникации точка-точка), либо объекты сообщений (для групповой или отложенной коммуникации).

Подсистема *Comm* компонентного каркаса BlackBox предоставляет простую абстракцию коммуникации через поток байтов, на основе которой могут быть построены службы сообщений. Между прочим, это может быть использовано для реализации удаленных элементов управления, т.е. таких, которые отображают и манипулируют данными на удаленной машине. Например, таким способом можно проверять внутреннее состояние встроенной системы.

Подсистема *Sql* BlackBox Component Framework обеспечивает простой интерфейс распределенных объектов, специализированный для доступа к реляционным базам данных (SQL-серверам). Более общие распределенные объекты, основанные на технологии DCOM [COM], могут быть доступны и реализованы при помощи компилятора Direct-To-COM для языка Компонентный Паскаль.

1.4 Настройки на разные языки пользователей

Глобальная природа современного бизнеса програмного обеспечения часто требует выпуска "локализованных" версий программного пакета. Это требует, как минимум, перевода в программе всех констант-цепочек литер, которые может увидеть пользователь. Например, в такой стране как Швейцария с ее четырьмя официальными языками,часто требуется, чтобы у одного и того же программного продукта были версии на немецком, французском и итальянском языках. Может даже потребоваться адаптировать к разным языкам формат диалоговых окон, т.к. заголовки и ярлыки элементов управления имеют разную длину в различных языках.

Если такие зависимые от языка аспекты жестко закодированы в исходном тексте программы, то локализация версии требуют редактирования и компиляции исходного кода. Это всегда деликатный вопрос, т.к. легко могут быть внесены ошибки и несогласованности. Кроме того, это очень неудобно: из-за отсутствия наглядности трудно

менять неинтуитивно "впаянный" непосредственно в исходный текст программы формат диалоговых окон, и слишком громоздко выполнять полный цикл редактирования, компиляции и запуска, если нужно всего лишь изменить расположение или размер элемента управления. Разработано много утилит, которые предлагают более удобные специализированные редакторы, в частности, редакторы диалоговых окон. Например, такая утилита может генерировать программный код из интерактивно построенного диалогового окна, и этот код затем компилируется. Так как программист должен редактировать сгенерированный исходный код, легко внести рассогласованность между расположением элементов управления и программным текстом.

К счастью, существует существенно лучший способ решить эту проблему, избежав промежуточного генератора исходного кода и использования выдачи редактора непосредственно в программе. Редактор сохраняет информацию о формате диалогового окна в так называемом файле *ресурсов*. Программа затем считывает такие ресурсы в нужный момент, например, при открытии диалогового окна. Такие ресурсы могут рассматриваться как сохраняемые объекты, которые достаточно модифицировать только при изменении конфигурации программной системы.

Итак, проблема удобной адаптации программы к различным естественным языкам может быть решена выделением параметров, зависящих от языка и региона, из исходного кода. Их располагают в ресурсных файлах, которые могут быть изменены удобными специализированными редакторами.

Разделение деталей пользовательского интерфейса и собственно программной логики — это способ сделать клиентское ПО более модульным и лучше адаптируемым. В предельном случае ресурсы почти *становятся* клиентским ПО: веб браузер на клиентской машине — это все что нужно для пользовательского интерфейса; ресурсы (HTML тексты) загружаются с сервера по мере необходимости. (Однако такой подход с "ультра тонкими" клиентами становится менее разумным с увеличением сложности самих браузеров.)

Система Блэкбокс использует свой стандартный формат документов для хранения ресурсов. Например, диалоговое окно и его элементы управления — просто части некоего составного документа, хранимого в файле. То же визуальное представление используется для проектирования диалогового окна и при его фактическом использовании — при этом нет необходимости в отдельном редакторе. Ресурсы могут быть доступны в версиях для нескольких языков одновременно, и языки могут даже переключаться во время исполнения. Это полезно для специализированных программных приложений, используемых в регионах, где разговаривают на нескольких языках.

2 Составные документы

В предыдущих разделах мы обсудили разнообразные проблемы, которые появляются в интерактивных системах, и мы увидели различные архитектуры и подходы к проектированию, которые помогают их решить.

В следующих разделах мы сосредоточим внимание на проблемах программирования, которые ставят составные документы, и на проектных схемах, помогающих их решить.

2.1 Сохраняемость

Будем называть объект, который содержится в окне, *объектом представления* или, короче, *вьюшкой* (View; см. О переводе терминологии). Вьюшка изображает себя в окне, и она может интерпретировать пользовательский ввод в этой области, такой как щелчок мышкой. В наиболее общем случае вьюшку можно открыть и сохранить как документ. Другие вьюшки могут быть созданы динамически или загружены из файлов ресурсов. Элементы управления диалогового окна "Найти и заменить" — пример последнего случая.

В общем случае, вьюшка — сохраняемый объект. Это предполагает механизм для выгрузки и загрузки визуального представления. Используя объектно-ориентированное программи-

рование, мы можем моделировать вьюшку как конкретизацию общего сохраняемого объекта, который называется *хранилище*. Оно может выгрузить и загрузить свое сохраняемое состояние. Вьюшка — это конкретизация хранилища, т.е. его подтип. В дополнение к выгрузке и загрузке себя, вьюшка знает, как изображать себя, и может обрабатывать пользовательский ввод. Эти общие вьюшки могут быть конкретизированы дальше, например, до визуальных представлений текста, визуальных представлений рисунков и т.п. (см. рис. 2-1).

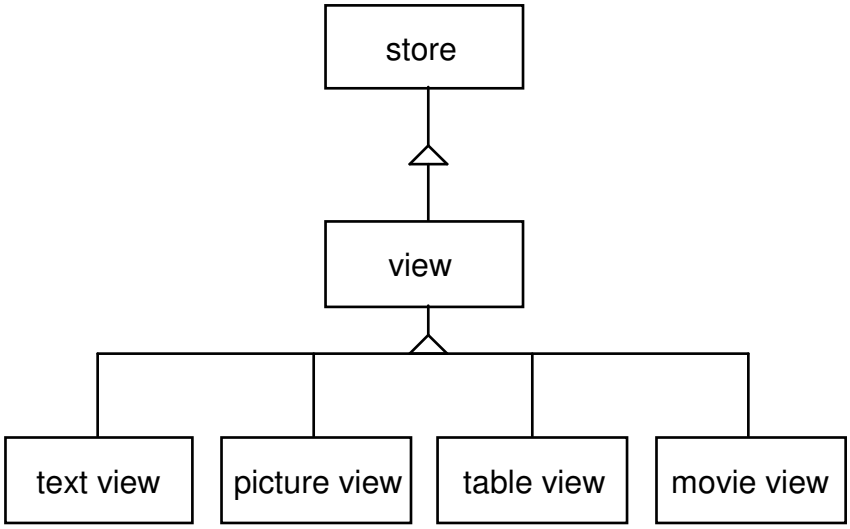

Рисунок 2-1. Соотношения подтипов между визуальными представлениями и хранилищами.

В системе Блэкбокс хранилища представлены как тип Store в модуле Stores; вьюшки представлены как тип View в модуле Views; и существует множество различных специфичных вьюшек, такие как тип View в модуле TextViews, или тип View в модуле FormViews, или тип Control в модуле Controls (см. рис. 2-2). И это только малая часть, в реальности же типов вьюшек намного больше. Формат файлов механизма хранилища не зависит от платформы, т.е. различия в порядке байтов учитываются должным образом.

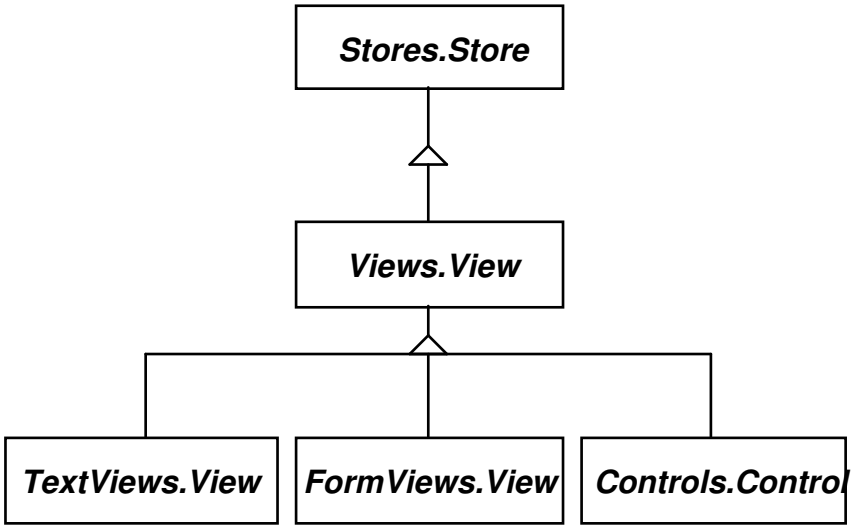

Рисунок 2-2. Соотношения подтипов между визуальными представлениями и хранилищами в BlackBox

Часто копирование хранилища эквивалентно записи его во временный файл и последующему его прочтению (выгрузка старого объекта в файл, затем создание и

загрузка нового объекта из файла). Это делает удобным объединение поддержки сохраняемости и копирования в одной и той же абстракции хранителя. Тем не менее, хранилища в BlackBox способны изменить свою сущность или "исчезнуть" при выгрузке в файл, в этом случае копирование не эквивалентно более выгрузке/загрузке. Например, вьюшки-маркеры ошибок, которые указывают синтаксическую ошибку, удаляют сами себя при выгрузке — хотя видимый маркер ошибки, подобно другим вьюшкам, может быть скопирован с помощью обычного механизма (drag-n-drop). Чтобы обеспечить такую гибкость, Блэкбокс не трактует копирование автоматически как пару выгрузка/загрузка. Хранилища имеют явные методы Externalize, Internalize и CopyFrom. При выгрузке они получают возможность подставить вместо себя другие хранилища, возможно, NIL. Этот механизм, позволяющий изменить свою сущность на другую, соответствует шаблону проектирования *заместитель* [proxy].

2.2 Независимость от устройства отображения

Вычислительный мир разнороден, с различными аппаратными и операционными платформами на рынке. Интернет, с его миллионами платформ на любой вкус, окончательно сделал ясным, как важна независимость кода и данных от платформы. Этого нелегко достичь, потому что программное и аппаратное обеспечение может широко изменятся между платформами. Устройства отображения, такие как мониторы и принтеры, — хороший пример. Их пространственное и цветовое разрешения могут чрезвычайно различаться.

Чтобы обеспечить независимость программ от платформы, вьюшки и их сохраняемое состояние не должны ссылаться на конкретные устройства. Вместо этого *пространство документа* и *пространство устройства отображения* должны быть четко разделены, а подходящее отображение между ними должно обеспечиваться средой. Например, в пространстве документа расстояния измеряются в хорошо определенных единицах, таких как дюймы или миллиметры, тогда как в пространстве устройства отображения расстояния измеряются в пикселях, каким бы ни был их реальный размер.

Для моделирования разделения между пространствами устройства отображения и документа, где пространство документа сформировано сохраняемыми объектами изображения (вьюшками), нам нужна абстракция устройства отображения. Мы называем такой объект *фреймом* [frame]. Фрейм — это прямоугольный участок для рисования, вместе с которым предоставляется некий набор примитивов рисования. Когда он делается видимым, вьюшка получает фрейм и через этот фрейм может изобразить себя на экране (или принтере). Фрейм выполняет преобразование между единицами пространства документа и пикселями на устройстве отображения, так что визуальному представлению никогда не нужно иметь дело с размерами пикселов напрямую.

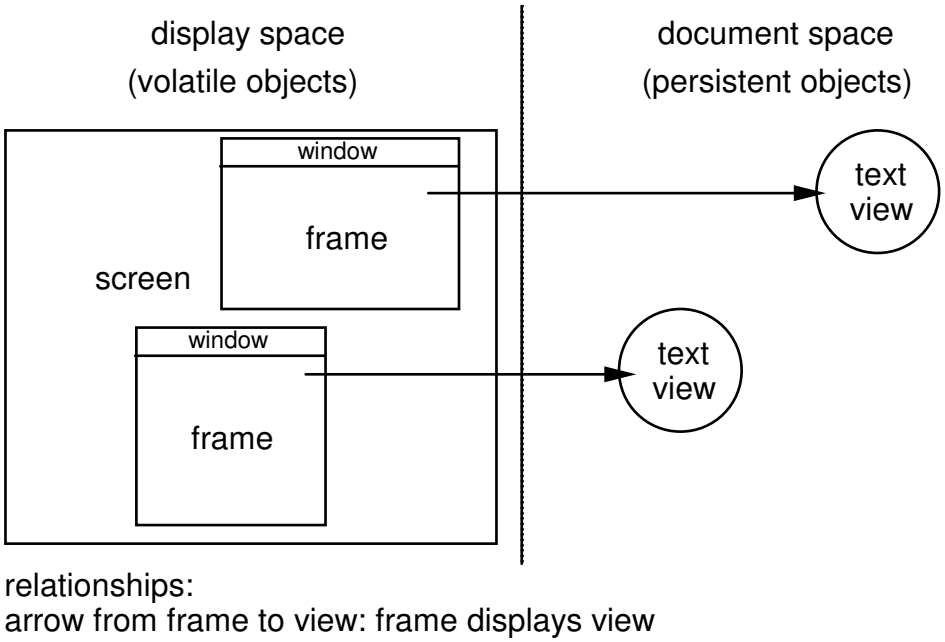

Рисунок 2-3. Разделения пространств устройства отображения и документа

В системе Блэкбокс единицы пространства документа измеряются в 1/36000 мм. Это значение выбрано так, чтобы минимизировать ошибки округления для типичных разрешений экрана. Фрейм BlackBox'a обеспечивает локальную систему координат для своего визуального представления. Пока вы можете рассматривать фрейм как окно, а вьюшку — как документ. Вьюшка, которая не изображена, не имеет фрейма. Фреймы — это "легкие" объекты, которые создаются и удаляются по необходимости. Они создаются и разрушаются каркасом, и нет необходимости реализовывать или расширять их программисту вьюшек.

Единственный случай, где необходимо переопределять фреймы, — это когда для зависимых от платформы элементов управления или OLE-объектов нужно создать свернутые вьюшки Блэкбокса. Т.к. написание таких вьюшек — «грязная» работа, каркас обеспечивает стандартные свертки OLE (версия для Windows) и свертки для важных (до OLE) стандартных элементов управления.

2.3 Редактирование с несколькими вьюшками

Простые вьюшки хороши, когда они достаточно малы, чтобы уместиться на экране. Однако тексты, таблицы или графики легко могут стать больше, чем экран или печатаемая страница. В этом случае фрейм (который лежит строго в пределах своего устройства отображения) может отображать только часть вьюшки, функционируя как окно, в котором видна только часть воображаемого полного изображения вьюшки. Чтобы увидеть ее другие части, нужно выполнить *прокрутку*, т.е. сдвинуть окно так, чтобы стала видна другая часть вьюшки. Например, можно измерить первую строку, показываемую текстовой вьюшкой, так, чтобы вместо самой верхней строки текста ею стала другая, и тогда вьюшка покажет часть текста, расположенную ниже этой строки (см. рис. 2-4).

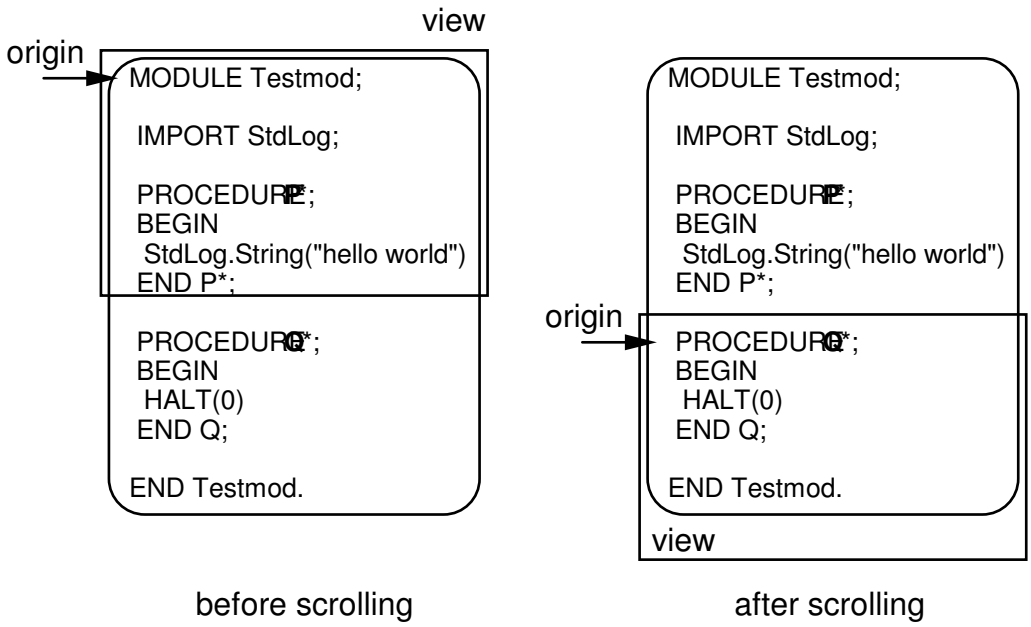

Рисунок 2-4. Прокрутка визуального представления текста

Необходимость выполнять прокрутку может стать большим неудобством при работе с двумя удаленными друг от друга частями вьюшки, потому что в таком случае придется часто выполнять прокрутку туда и обратно. Эту проблему можно решить с помощью *редактирования нескольких объектов изображения (вьюшек)*: одни и те же данные представлены в более чем одной вьюшке. Эти вьюшки могут отличаться только своим началом, или они могут отличаться более существенно. Например, одна вьюшка может показывать список числовых значений как таблицу, а другая визуализировать те же данные как круговую диаграмму (см. рис. 2-5).

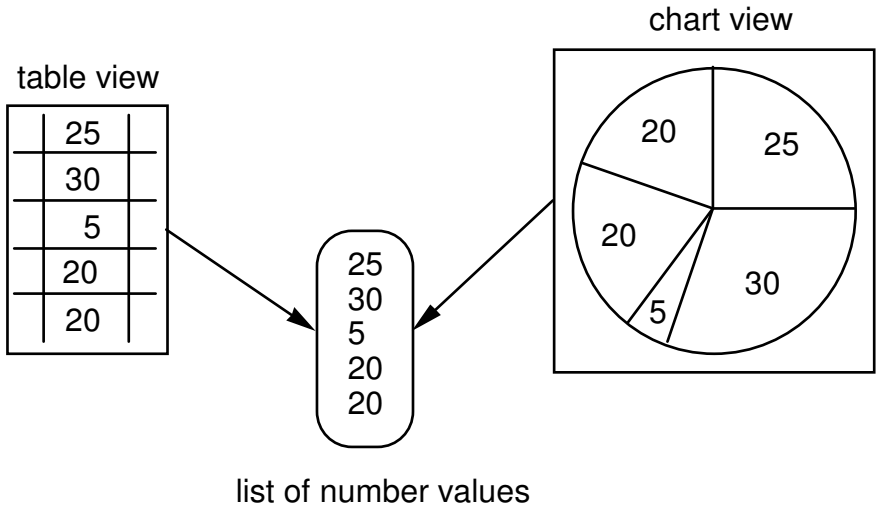

Рисунок 2-5. Различные типы объектов изображения, представляющие одни и те же данные

Объект, представляющий данные, которые могут быть представлены в различных вьюшках, называется *моделью*. Разделение на вьюшку и модель восходит к проекту Smalltalk в Xerox PARC. В терминологии проектных схем вьюшка — это *наблюдатель* модели.

В системе Блэкбокс разделение модель/вьюшка не является обязательным. Маленькие вьюшки фиксированного размера, которые легко умещаются в окне, обычно не нуждаются в отдельных моделях, тогда как большие или более чистые нуждаются. Это может происходить в ситуации, подобной следующей, где вьюшка имеет отдельную модель, и оба

вместе они формируют документ, который изображается в окне:

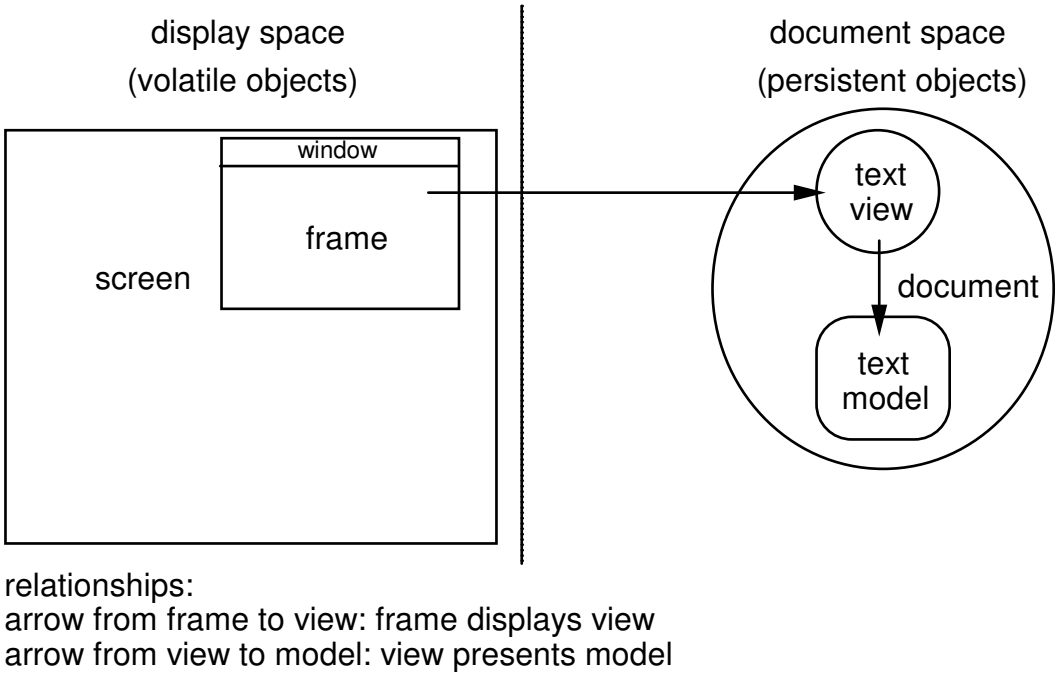

Рисунок 2-6. Разделение моделей и вьюшек в BlackBox.

Пользователь может открыть второе окно, в котором будет изображаться новая вьюшка, но для той же модели. Это происходит в следующей ситуации:

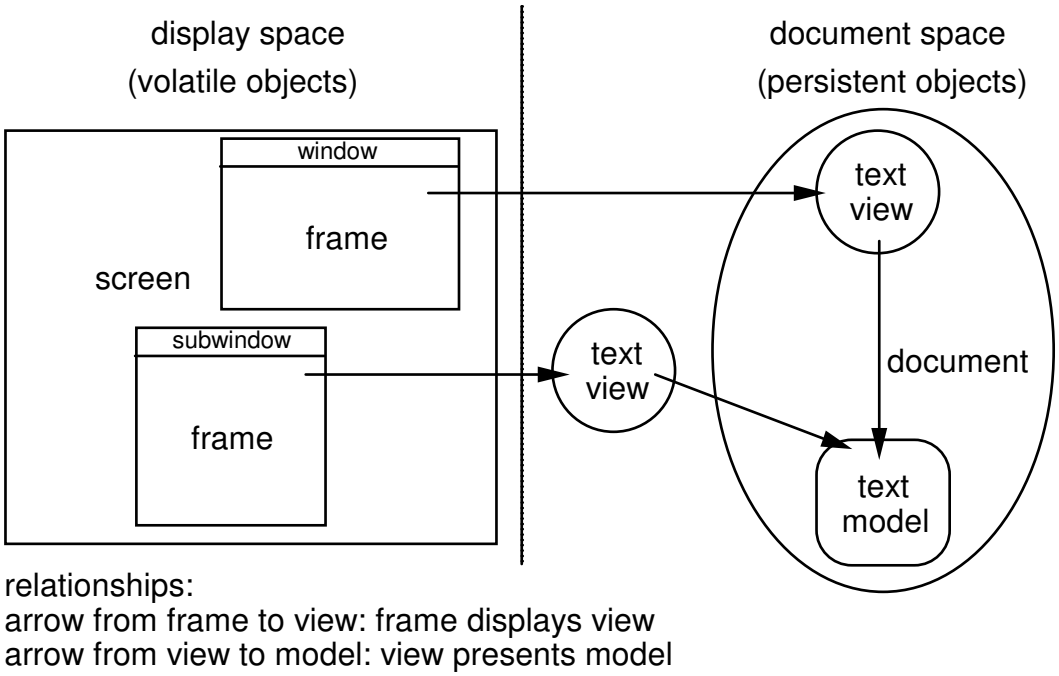

Рисунок 2-7. Множество вьюшек для одной и той же модели

Только одна из двух вьюшек выгружается, когда документ сохраняется в файл. Чтобы избежать путаницы, одно окно выделяется как главное, тогда как все другие являются так наз. *дочерними окнами* [subwindow]. Если закрыть дочернее окно, то закроется только это окно. Но если закрыть главное окно, то одновременно с ним закроются и все дочерние окна.

2.4 Распространение изменений

Редактирование при наличии нескольких вьюшек создает проблему согласованности. Редактирование вьюшки означает, что она заставляет измениться свою модель. В результате должны быть, если нужно, обновлены *все* вьюшки, изображающие эту модель, а не только та вьюшка, с которой пользователь в данный момент работает. Например, если пользователь изменяет значение, работая со вьюшкой, представляющей таблицу на рис. 2-5, то круговая диаграмма в другой вьюшке должна обновиться соответственно, потому что изменилась их общая модель.

Такое распространение изменений легко реализовать. Когда модель завершила обновление своих данных, она уведомляет все вьюшки, которые ее изображают, посылая им *сообщение об обновлении*, описывающее операцию, которая была выполнена.

Соответствующий механизм уведомления легко реализовать с помощью прохода по линейному списку вьюшек, связанных с данной моделью. Такую функциональность можно обеспечить раз и навсегда, сосредоточив ее в соответствующем классе. Это класс, обеспечивающий распространение информации об изменениях, должен обеспечить интерфейс, состоящий из двух частей. Первая часть позволяет регистрировать новые вьюшки, которые хотят прослушивать сообщения об обновлении ("подписка"). Вторая часть позволяет посылать сообщения об обновлении ("публикация"):

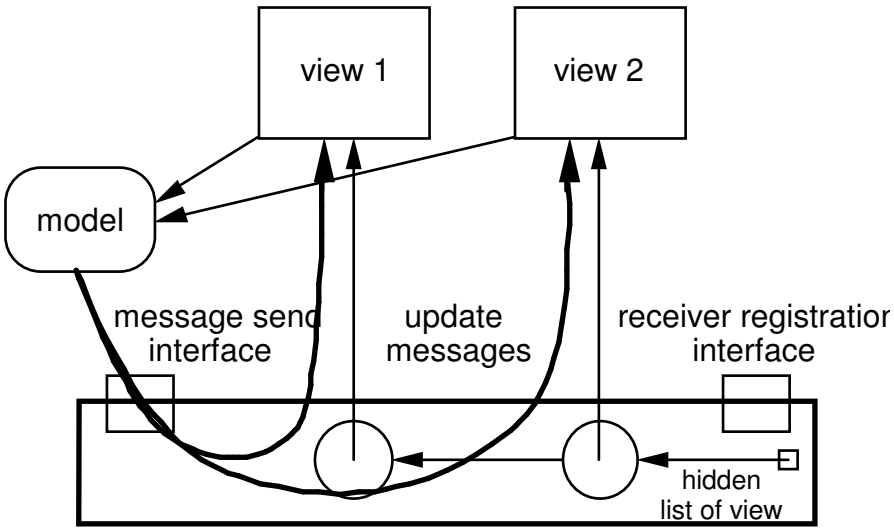

Рисунок 2-8. Механизм распространения сообщений об изменениях

Этот механизм имеет важное преимущество: он разделяет модель и вьюшку так, что самой модели нет необходимости знать о своих вьюшках, и список вьюшек может быть инкапсулирован в механизм распространения изменений. Такая связь между моделью и вьюшкой желательна, потому что она уменьшает сложность. Модель становится независимой от своих вьюшек. Это важно для расширяемой системы, потому что позволяет добавлять новые типы вьюшек без изменения модели. Такое было бы невозможно, если модель должна была знать слишком много о своих вьюшках.

Иногда механизм, описанный выше, делает слишком много. Он посылает сообщения об обновлении всем вьюшкам, независимо от того, нужно их обновлять или нет. Например, когда в текстовой модели удалено слово, то некоторая текстовая вьюшка может изображать часть текста, на которую удаление совершенно не повлияло. Такая вьюшка не должно ничего делать. Конечно, все вьюшки, которые *отражают* измененную часть текста, должны быть обновлены соответственно.

По этой причине система Блэкбокс посылает сообщения об обновлении только тем вьюшкам, которые имеют фреймы. Т.к. каждый фрейм знает свою вьюшку, механизм

распространения изменений в действительности регистрирует фреймы, а не вьюшки. Каркас уже имеет список фреймов для управления окнами, поэтому этот список можно использовать многократно. Таким образом, вместо того, чтобы держать один объект распространения изменений для каждой модели, BlackBox использует один глобальный сервис распространения изменений, встроенный в менеджер окон. Сообщения представлены как статичные записи сообщений. Это нетрадиционная, но надежная, эффективная и легкая реализация проектной схемы наблюдатель.

2.5 Вложенные вьюшки

До сих пор мы принимали, что вьюшка (возможно с моделью) — это документ, и фрейм — это окно. В действительности, все немного сложнее. Причина в том, что вьюшки могут быть вложенными одна в другую. Некоторые вьюшки могут содержать ("внедрять") другие вьюшки; по этой причине они называются контейнерами. Некоторые контейнеры очень общие и могут содержать любое число и любые типы других вьюшек. Например, вьюшка, изображающая текст, может содержать в себе вьюшки для таблиц, рисунков и т.д. Она может даже содержать другую текстовую вьюшку. Другими словами, можно создавать сколь угодно глубокие иерархии вьюшек, вложенных одна в другую.

Как это влияет на то, о чем мы до сих пор говорили? Очевидно, нам нужно обобщить нашу нотацию вьюшек и фреймов. Вьюшка может содержать другие вьюшки, которые могут содержать другие вьюшки и т.д.; т.е. вьюшки образуют иерархию. В принципе такие иерархии могут даже не быть деревьями, а произвольными ориентированными ациклическими графами. Это значит, что некоторые вьюшки в документе могут ссылаться на одну и ту же модель, но ссылки не могут идти «вверх» в иерархии вьюшек данного документа, потому что это привело бы к бесконечной рекурсии при прорисовке документа, подобно телевизору, изображающего самого себя.

Фреймы представляют подмножество иерархии вьюшек. Так как фрейм всегда полностью содержится в своем родительском фрейме, иерархия фреймов обязательно образует дерево.

Документ можно теперь рассматривать как корень ориентированного ациклического графа из вьюшек, и окно можно рассматривать как корень дерева фреймов.

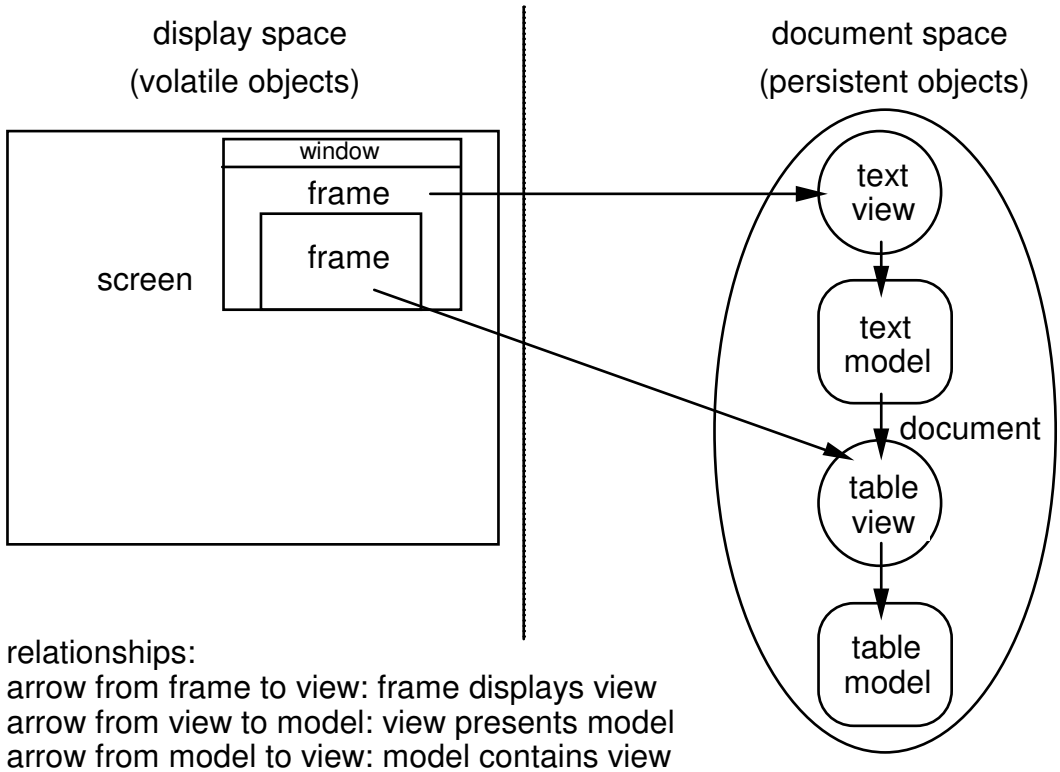

Рисунок 2-9. Вложенные вьюшки и фреймы

Это понятное обобщение, которое не добавляет неожиданной сложности. Это упрощенная реализация проектной схемы *компоновщик* [composite]. Она была упрощена, потому что защищенность типов в Компонентном Паскале делает ненужным абстрактный тип вьюшек со специфичным для контейнера операциями, которые не имеют смысла для вьюшек, не являющихся контейнерами.

К сожалению, дальнейшие сложности возникают, когда выполняется редактирование с несколькими вьюшками, а вьюшки являются составными. Прежде всего, редактирование с несколькими вьюшками означает разделение вьюшек и моделей. Вложенная вьюшка принадлежит модели, т.к. это часть данных, которые изображает вьюшка-контейнер. Например, пользователь, работающий с текстовой вьюшкой, может удалить внедренные вьюшки также, как он может удалить внедренные символы. Рисунок 2-10 иллюстрирует ситуацию двух вьюшек-контейнеров с контейнерной моделью, в которую внедрена другая вьюшка:

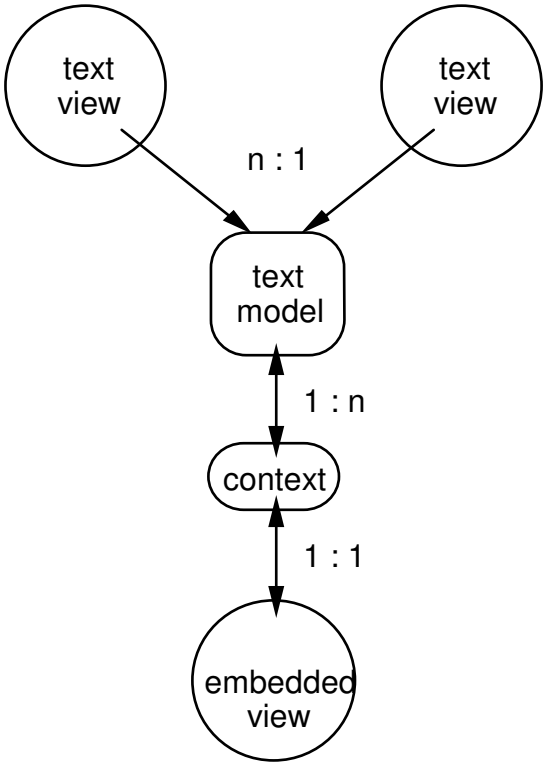

Рисунок 2-10. Контейнерные вьюшки с моделью, контекстом и внедренной вьюшкой

Отметим интересную связь между моделью и внедренной вьюшкой: объект контекста. Он описывает положение и, возможно, другие атрибуты одной внедренной вьюшки. Тип контекста определяется контейнером. Контекст создается контейнером, когда вьюшка внедряется. Внедренная вьюшка может спросить свой контекст о своем положении в контейнере и может даже получить большую информацию. Например, контекст текстового контейнера может обеспечить информацию о позиции внедренной вьюшки в тексте (индекс символа), о его текущем шрифте и цвете и т.д.

Внедренная вьюшка может использовать свой объект контекста не только для получения информации о контейнере, она может также заставить контейнер выполнить некоторое действие. Например, внедренная вьюшка может попросить контейнер через объект контекста уменьшить или увеличить себя. Контейнеры могут выполнить подобные запросы или проигнорировать их. В переговорах между контейнером и одной из его внедренных вьюшек контейнер всегда имеет решающее слово. Например, контейнер обычно не разрешает внедренной вьюшке стать шире самого контейнера.

Объект контекста можно рассматривать как интерфейс обратного вызова, который контейнер устанавливает во внедренной вьюшке как подключаемый модуль, поэтому вьюшка может обратиться к контейнеру и получить его сервисы. Контекст — часть механизма оповещения, который позволяет контейнеру просматривать сообщения от содержащихся в нем вьюшек.

В последующих рисунках объекты контекста не изображены явно, чтобы рисунки было легче понимать. Просто помните, что каждая вьюшка носет контекст, который она может использовать по необходимости.

Теперь, когда поддерживаются вложенные вьюшки, а также разделение модели и изображения, становится возможным разрешить использовать дочерние окна для внедренных вьюшек, а не только для корневых вьюшек. Это может быть очень полезно, если нужно редактировать маленькую вьюшку. Тогда для нее просто открывается дочернее окно, в котором вьюшка показана с увеличением, и редактирование упрощается.

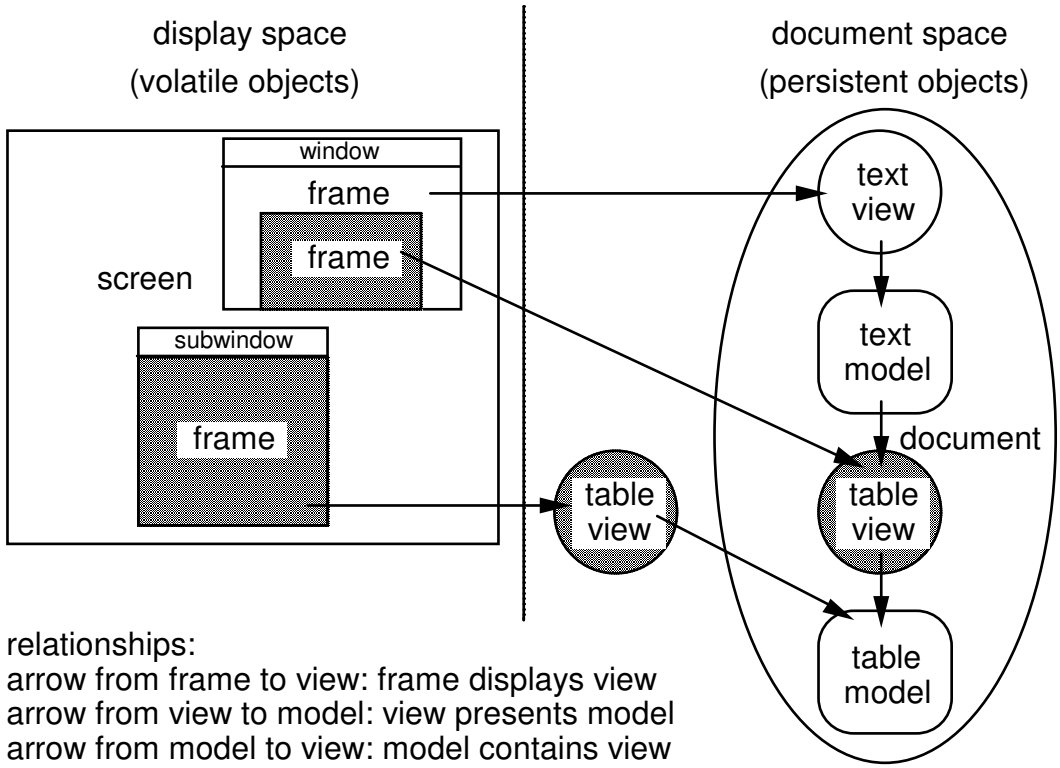

Рисунок 2-11. Дочернее окно на внедренную вьюшку.

Составные документы порождают дополнительную сложность. Для корневых вьюшек состояние прокрутки не сохраняется, когда документ сохраняется на диске. По крайней мере большинство пользователей предпочитают, чтобы состояние прокрутки вьюшки не сохранялось. Но дело обстоит иначе с внедренными вьюшками. Возможность установки начальной позиции внедренной вьюшки нужна не столько для удобства (для этого лучше подходят дочерние окна), сколько для печати, чтобы пользователь мог выбрать, какая часть вьюшки должна быть видна в контейнере.

То же верно для механизма отмены/повтора: прокрутка в корневой вьюшке не является редактированием документа, и поэтому механизм отмены/повтора добавил бы больше неудобства, чем пользы. Однако прокрутка внедренной вьюшки должна рассматриваться как настоящая операция редактирования, и поэтому должна быть отменяемой/повторяемой. В итоге это означает, что реализатор вьюшки должен трактовать неперсистентное состояние вьюшки по-разному в зависимости от того, является ли она корневой или

внедренной.

Объединение идеи вложенных вьюшек с идеей редактирования с несколькими вьюшками также разрушает взаимно однозначное соответствие между фреймами и вьюшками. Как мы скоро увидим, это наиболее серьезная проблема, которую должен решать проектировщик.

Чтобы понять, почему вьюшка может изображаться в различных фреймах, рассмотрим ситуацию на рисунке 2-12:

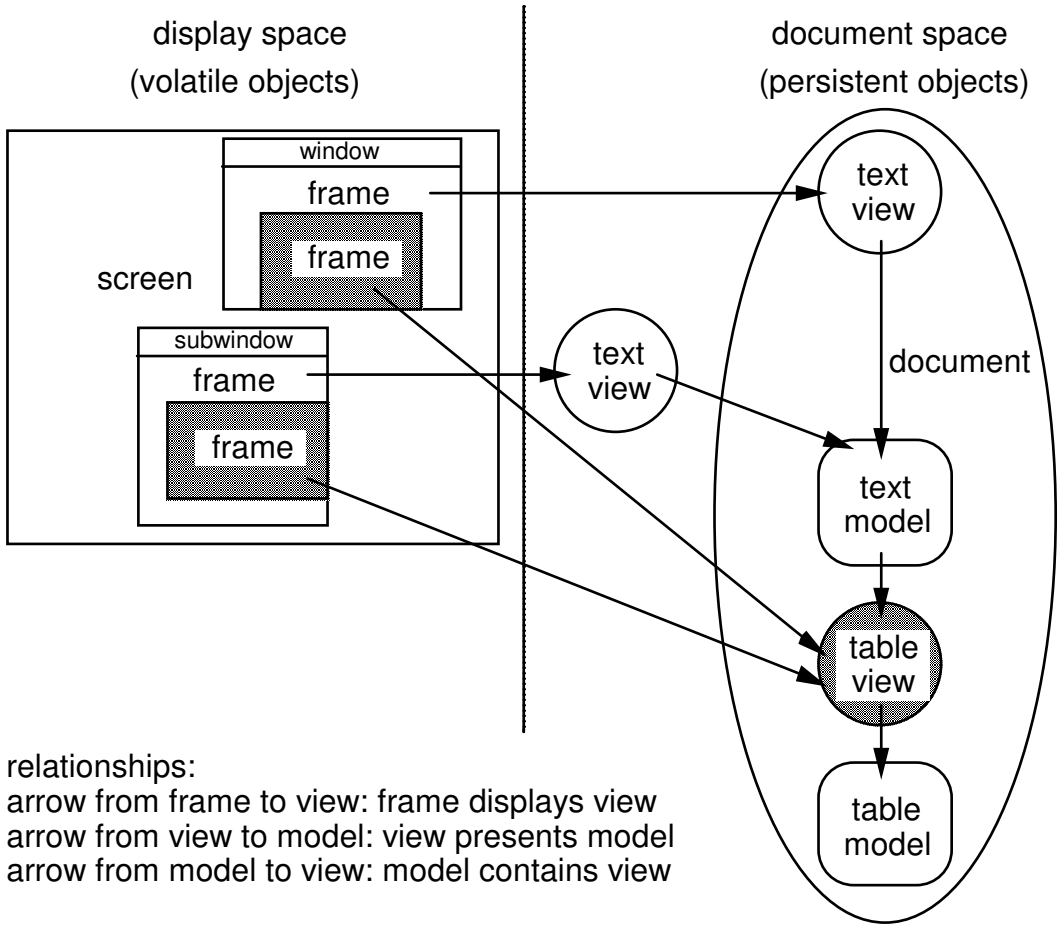

Рисунок 2-12. Несколько фреймов, отображающих одну и ту же вьюшку.

Такая ситуация не может возникнуть при отсутствии вложенных вьюшек, потому что самая внешняя вьюшка всегда копируется (чтобы обеспечить возможность независимой прокрутки или разрешить другие специфичные для нее операции). Но когда вьюшка внедрена в контейнер, изображаемый в двух различных окнах, то тогда эта вьюшка изображается в двух различных фреймах. Это делает управление ссылками между фреймами и вьюшками более сложным; но, что более важно, это делает более сложным распространение изменений и обновление экрана.

Почему? Предположим, что имеется n фреймов, отображающих одну и ту же вьюшку. Пусть теперь модель вьюшки изменилась, и информация об этом изменении рассылается всем вьюшкам, показывающим данную модель. Так как для этой модели есть только одна вьюшка, то механизм рассылки информации об изменениях, который мы обсудили ранее, пошлет вьюшке сообщение об обновлении ровно один раз. Тем не менее, есть n мест на экране, которые нужно обновить. Следовательно, вьюшка должна выполнить n обновлений, пройдя по всем фреймам, которые ее отображают.

Для вьюшки корректно поддерживать список фреймов, в которых она отображается, — сложная задача. Поэтому Блэкбокс освобождает проектировщика от этой чреватой ошибками задачи. Вьюшка в BlackBox'e не содержит списка фреймов. Вместо этого каркас

поддерживает деревья фреймов автоматически. Именно по этой причине на вышеприведенном рисунке стрелки от фреймов к вьюшкам направлены только в одну сторону: в Блэкбоксе вьюшка не знает свои фрейм(ы). Нужные фреймы передаются вьюшке, когда каркас просит ее сделать что-то, связанное с рисованием. (Такой подход связан с проектной схемой *приспособленец* [flyweight].)

Когда вьюшка получает сообщение об обновлении и решает, что нужно обновить свои фрейм(ы), она просто просит каркас послать другое сообщение об обновлении всем своим фреймам. Каждый из этих фреймов попросит вьюшку обновить его, когда это будет необходимо. Этот механизм будет более подробно обсужден в главах, посвященных построению вьюшек.

Так как теперь больше нет взаимно однозначного соотношения между вьюшками и фреймами, полезно подытожить различия между фреймами и вьюшками:

аспект	**фрейм**	**вьюшка**
живет в	пространстве устройства отображения	пространстве документа
persistent	нет	да
расширяется проектировщиком	обычно нет	да
может содержать зависимые от устройства данные	да	нет
иерархия	дерево	ацикличный орграф
связан с	одной вьюшкой	0..n фреймами
обновление провоцируется	его вьюшкой	его моделью (если есть)
полностью содержится в контейнере	да	не обязательно

Таблица 2-13. Различия между фреймами и вьюшками

2.6 Ленивое обновление

В предыдущем разделе мы увидели, как обновления выполняются в два шага: модель уведомляет свои вьюшки об изменении, а вьюшки уведомляют свои фреймы об области, которую нужно обновить.

Теперь рассмотрим сложное изменение модели. Например, выделенные графические объекты, которые являются частью модели графиков, перемещаются на некоторое расстояние. Это возможно было бы выполнить в три фазы: в первой фазе выделенные объекты были бы удалены (требование обновления), затем выделенные объекты были бы перемещены, и, наконец, выделенные объекты были бы вставлены снова в их позицию назначения (требование другого обновления). Этот подход не очень эффективен, т.к. он требует двух уведомлений и частичных обновлений. Кроме того, необходима осторожность в последовательности действий, т.к. некоторым из них нужно состояние модели до перемещения, а другим — после перемещения.

Есть гораздо более простой и эффективный подход. Идея — отложить вторую фазу обновлений, т.е. фазу, когда перерисовываются фреймы вьюшки. Вместо немедленного обновления фреймов вьюшка запоминает только геометрическую область, которую нужно обновить. В вышеприведенном примере она могла объединить области графических объектов до и после перемещения (см. рис. 2-14).

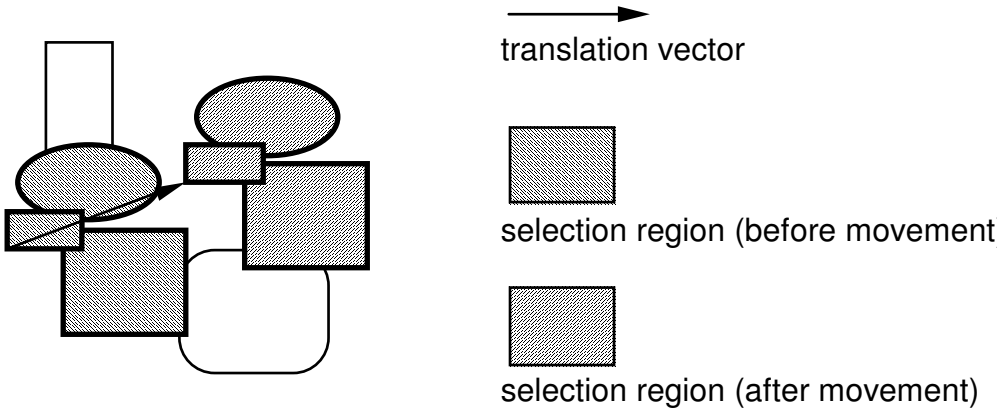

Рисунок 2-14. Область обновления для группы перемещенных объектов

Накопленная область обновления потом обновляется за один единственный шаг после того, как было осуществлено перемещение графических объектов. В рисунке выше область обновления — это вся заштрихованная область, состоящая из области, занятой выделенными объектами до перемещения, и области, занятой ими после перемещения.

Каркас должен поддерживать этот подход, обеспечивая некий механизм для добавления геометрических областей и перерисовки фрейма точно в этой области ("отсекая" все ненужное рисование для увеличения скорости и минимизации мерцания).

Механизм *ленивых обновлений* используется в системе Блэкбокс, как и во всех основных существующих оконных системах. Единственный его недостаток в том, что для долго выполняющихся команд могут понадобиться промежуточные обновления, чтобы информировать пользователя о том, как проходит выполнение команды. По этой причине система Блэкбокс предоставляет возможность форсировать обновления.

2.7 Режимы контейнера

При создании графического пользовательского интерфейса удобно создавать строение формы интерактивно, нежели "прошивать" координаты элементов управления в исходном коде программы. Визуальный конструктор, т.е. графический редактор специального назначения, можно использовать для интерактивного изменения строения форм. Это предполагает два различных режима использования формы и ее элементов управления: разработки и исполнения. Когда редактируется строение формы в визуальном конструкторе — это *режим разработки*: форма строится, но еще не используется. Законченную форму можно сохранить и позже открыть для использования конечным пользователем. Это называется *режимом исполнения*.

Существует два различных подхода для управления строением в режиме исполнения: либо каркас позволяет открывать данные, которые были сохранены визуальным конструктором, либо он исполняет код, который был сгенерирован из этих сохраненных данных. Первый подход требует интерпретатор для данных визуального конструктора, второй подход требует генератора между визуальным конструктором и действительным использованием формы. Обычно генератор генерирует исходный код, который, возможно, завершается программистом и потом компилируется в машинный код, используя стандартный компилятор языка программирования.

Сегодня последний подход следует считать устаревшим, хотя он еще используется во многих утилитах: это просто неудобная излишняя работа. Более того, когда это заставляет программиста редактировать сгенерированный исходный код, последовательные изменения строения формы становятся проблематичными: исходный код должен сгенерироваться заново без потерь изменений, которые сделал программист. Это ненужный источник несогласованностей, сложности и это замедляет разработку.

Первый подход намного проще. Вывод визуального конструктора рассматривается как ресурс, который может быть использован *как есть* во время исполнения. Это требует интерпретатора формата файла визуального конструктора.

Наиболее элегантный подход — использовать тот же редактор (т.е. реализацию визуального представления) и, следовательно, тот же формат файла и для режима разработки, и для режима исполнения. В качестве формата файла можно снова использовать стандартный формат составного документа. Это значит, что строение формы в режиме разработки полностью является обычным составным документом. В режиме исполнения это все тот же документ, только его взаимодействие с пользователем различно.

В режиме разработки элемент управления — пассивный прямоугольник, который можно перемещать, копировать и удалять, но у него нет своего интерактивного поведения. В режиме исполнения элемент управления может получить фокус и его *содержимое* можно редактировать. Например, строку, содержащуюся в элементе управления *текстовое поле*, можно редактировать, или кнопку можно нажать и отпустить.

Прямое использование составных документов как пользовательских интерфейсов называется *составными пользовательскими интерфейсами*. Заметьте, это обобщение составных документов делает термин "документ" вводящим в заблуждение, потому что программа может содержать много форм (и, следовательно, документов составного пользовательского интерфейса), даже если она не обрабатывает документы в традиционном смысле (т.е. как видимых конечным пользователем).

Компонентный каркас Блэкбокса использует свой механизм поддержки редактирования с несколькими вьюшками, чтобы предоставить еще одну полезную возможность. Так как документ можно открыть в нескольких вьюшках одновременно, оказывается возможным открыть проектируемую диалоговую форму в двух окнах. Блэкбокс позволяет использовать вьюшки диалоговых форм в различных режимах: построения и маски. Эти режимы соответствуют этапу разработки и этапу использования, но Блэкбокс дает здесь одно серьезное улучшение: работать в двух режимах можно одновременно. Например, одно окно может показывать форму в режиме построения, а другое окно — в режиме маски (см. рис. 2-15). Когда разработчик редактирует форму в окне режима построения, любое изменение формы немедленно отображается в окне режима маски. И наоборот: пользовательский ввод в окне режима маски немедленно становится видимым в окне построения. Эта возможность возникает благодаря стандартному механизму распространения информации об изменениях в Блэкбоксе.

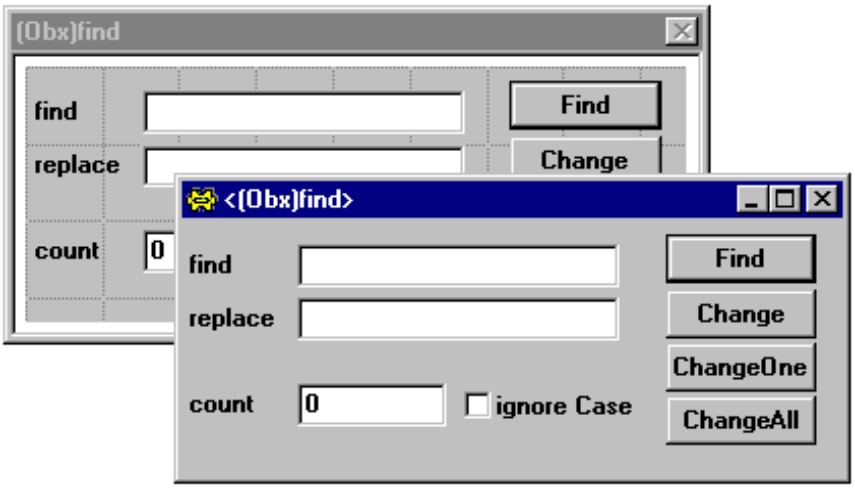

Рисунок 2-15. Форма ввода данных в режиме построения (заднее окно) и в режиме маски.

На самом деле возможности Блэкбокса еще более общие. В дополнение к режимам построения и маски, он поддерживает еще несколько режимов. Режим *редактирования* позволяет изменять расположение и содержимое элементов управления одновременно, а режим *просмотра* делает форму доступной только для чтения, разрешая лишь выделение и копирование частей документа.

Эти общие режимы доступны не только в графических формах. Они — возможности, разделяемые *всеми* общими вьюшками-контейнерами, включая представления текстов. Это значит, что можно (а иногда и более удобно) использовать текст вместо диалоговой формы при разработке диалогового окна. Пользователь не заметит разницы. Чтобы упростить построение таких мощных контейнеров, каркас обеспечивает исчерпывающую абстракцию контейнера с абстрактным классом модели контейнера, с абстрактным классом вьюшки-контейнера и с абстрактным классом контроллера для контейнера. Контроллер можно рассматривать как отделившуюся часть вьюшки. Он выполняет все взаимодействие с пользователем, включая обработку ввода с клавиатуры и манипуляций мышью, и он также управляет выделениями. Вьюшки и контроллеры во время исполнения находятся во взаимно однозначном соответствии. Выделение части визуального представления в объект контроллера повышает расширяемость (различные контроллеры могут быть реализованы для одной и той же вьюшке, и один и тот же контроллер работает со всеми конкретными реализациями вьюшки), и это также позволяет уменьшить сложность: контроллер сложной вьюшки, например, вьюшки-контейнера, может стать большим, что делает хорошей идеей разделить обязанности между различными типами (и обычно между различными модулями). В терминах проектных схем контроллер — это объект *стратегии*.

Абстракции контейнеров BlackBox'а были разработаны, чтобы абстрагироваться от специфических пользовательских интерфейсов контейнеров. Это значит, что детали внешнего вида контейнера (такие как обозначение фокуса OLE-объекта пунктирным прямоугольником) спрятаны от разработчика. В свою очередь, это позволяет реализовать контейнеры различным образом для различных платформ, без влияния на разработчика контейнеров. В частности, реализован внешний вид контейнеров, похожих на OLE и Apple OpenDoc. Хотя OpenDoc более не актуален, некоторые принципы пользовательского интерфейса, разработанные для него, полезны и сегодня.

2.8 Обработка событий

Когда пользователь взаимодействует с составным документом, это взаимодействие в каждый момент времени происходит в одной вьюшке. Эта вьюшка называется текущим *фокусом*. Щелчком в другом месте пользователь может изменить текущий фокус. Среди прочих событий фокус обрабатывает события от клавиатуры, т.е. нажатие клавиши интерпретируется вьюшкой, владеющей фокусом. Вьюшка, владеющее фокусом, и все содержащие его вьюшки называются *путем к фокусу*, где самая внутренняя вьюшка является фокусом.

Есть возможность позволить каркасу пользовательского интерфейса управлять текущим фокусом. Это требует централизованного управления фокусом. Более легкий подход — сделать управление фокусом децентрализованной деятельностью: оставить управление фокусом индивидуальным вьюшкам-контейнерам, которые все равно должны иметь дело с изменениями фокуса и передачей фокуса. Каждый контейнер просто запоминает, какая из его внедренных вьюшек является текущим фокусом, если такой есть. Контейнер не запоминает, находится ли эта вьюшка на текущем пути к фокусу или нет.

В Блэкбоксе пользовательские события посылаются вдоль пути к фокусу как записи сообщений, начиная с самой внешней вьюшки. Записи сообщений — статичные (размещаемые на стеке) переменные некоторого подтипа от *Controllers.Message*. Существуют сообщения контроллера для ввода с клавиатуры, перемещения мыши, механизма drag-n-drop и т.д. Сообщение контроллера продвигается по пути к фокусу, пока одна из вьюшек либо проинтерпретирует, либо отклонит его. Эта вьюшка и является фокусом по определению.

В терминах проектных схем, путь к фокусу — это *"цепочка обязанностей"*.

Что происходит, когда вьюшка, владеющая фокусом, получает и интерпретирует сообщение контроллера? Вьюшка (или ее модель, если таковая имеется) выполняет некоторую операцию над своим состоянием. Если эта операция влияет на персистентное состояние вьюшки или модели, она должна быть обратимой ("отменяемой"). Как реализован механизм отмены/повтора?

Главная идея здесь в том, что при получении сообщения контроллера вьюшка/модель не выполняют изменения состояния непосредственно. Вместо этого, она создает специальный *объект операции* и регистрирует его в каркасе. Каркас потом вызывает подходящую процедуру этого объекта для фактического выполнения/отмены/повторного выполнения действия.

Управление операциями осуществляется отдельно для каждого документа. Каждый документ содержит два стека операций: стек отмены и стек повторения. При выполнении операции в первый раз объект операции помещается в стек отмены, при этом стек повторения очищается. Когда пользователь выполняет отмену, производится отмена операции с вершины стека отмены, соответствующий объект операции удаляется из этого стека и помещается в стек повторения. Когда пользователь выполняет повторение, выполняется операция с вершины стека повторения, соответствующий объект удаляется из этого стека и помещается в стек отмены. Стеки отмены и повторения документа очищаются при сохранении документа (точка проверки). Кроме того, они могут быть очищены или уменьшены, когда каркас превышает лимит памяти. Для этой цели сборщик мусора информирует каркас о недостатке памяти, чтобы удалить старые объекты операций.

Некоторые операции, такие как перемещающие drag-n-drop (в отличие от обычного копирующего drag-n-drop), изменяют одновременно два документа. В таких редких случаях, создаются две операции: одна для документа-источника (например, операция удаления) и одна для документа-получателя (например, операция вставки).

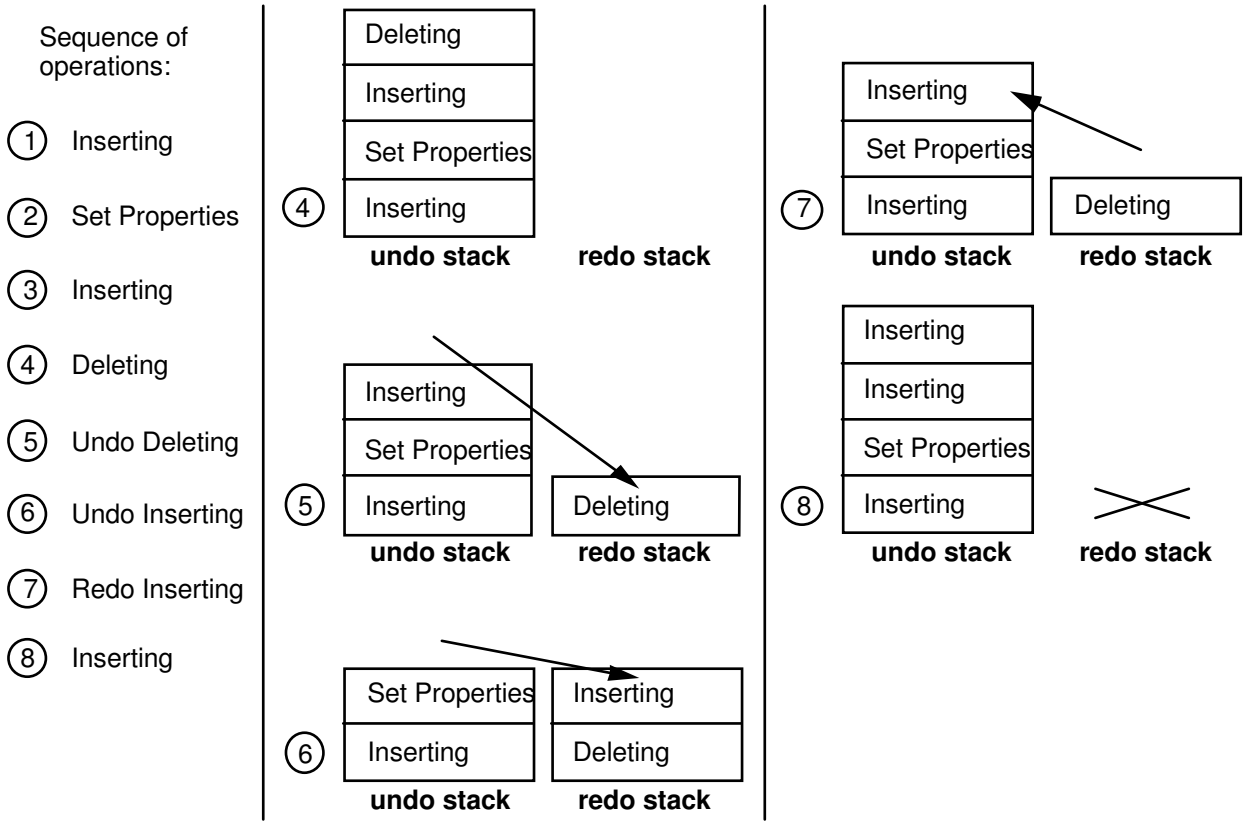

Рисунок 2-16. Последовательность операций и получившиеся изменения стеков отмены и

повторения.

На рисунке 2-16 показана последовательность операций от 1) до 8). Для последних пяти ситуаций показаны получившиеся стеки отмены и повторения. Например, после операции 5) стек отмены содержит операции Вставка, Установка свойств и Вставка (от вершины к дну стека), а стек повторения содержит Удаление.

Механизм отменить/повторить имеет отношение только к перситентному состоянию документа. Это состояние, которое можно сохранить в файле. Изменения временного состояния, такие как позиция прокрутки визуального представления, не записываются как операции, и, следовательно, их нельзя отменить (по крайней мере, для корневых вьюшек).

Отменяемые операции либо изменяют персистентное состояние вьюшки либо ее модели (состояние контроллера обычно является временным). Каркас знает, к каким документам принадлежит персистентный объект, потому что все персистеные объекты документа разделяют один и тот же домен. Домены будут обсуждаться в Части III, где механизм хранилища обсуждается более подробно.

Система Блэкбокс уникальна в том, что ее механизм отмены/повтора является компонентно-ориентированным: он позволяет осуществлять композицию отменяемых операций в *составные операции*, отменяемые как целое. Без этой способности вложенные операции записывались бы как плоская последовательность атомарных операций.

Рассмотрим, что бы это значило для конечного пользователя. Это бы значило, что пользователь мог выполнить команду меню, которая приводит к выполнению иерархии операций. До сих пор хорошо. Но, когда пользователь захочет отменить эту команду, он или она должен выполнить Правка->Отменить индивидуально для каждой отдельной операции команды вместо одной.

По этой причине BlackBox обеспечивает поддержку для произвольно вложенных операций: модули *Models* и *Views* оба экспортируют пару процедур *BeginScript / EndScript*. В этом контексте, "script" означает последовательность или иерархию атомарных операций, отменяемых как целое. Операции модели и вьюшки можно произвольно смешивать в скрипте. Абстрактные операции очень легкие. Они обеспечивают только одиночную процедуру без параметров, называемую *Do*. Эта процедура должна быть реализована обратимым образом, так что если она выполнена четное число раз, она не имеет эффекта. Если она выполнена нечетное число раз, она имеет тот же эффект, что и при выполнении один раз. В терминах проектных схем операция называется *командой.*

2.9 Элементы управления

Элементы управления — это «легкие» вьюшки, которые обеспечивают очень специфичную функциональность, которая приобретает смысл только в соединении с другими элементами управления и контейнером. Типичные элементы управления — командные кнопки и поля ввода текста. Они используются в диалоговых окнах или формах ввода данных. Параметры или другие данные, представляемые элементами управления, должны как-то обрабатываться соответствующей программой. Например, команда *Найти* берет строку поиска как вход, или данные, введенные в форму, должны посылаться реляционной базе данных. Могут существовать даже сложные взаимодействия между элементами управления и формой. Например, когда строка поиска пуста, кнопка *Найти* может быть недоступной. Если что-то набрано, кнопка доступна. Такие взаимодействия могут стать очень запутанными. В принципе, элемент управления — это просто наблюдатель своих данных. Это означает, что наиболее прямолинейно было бы реализовать данные (например, строку поиска) как модель.

При проектировании Блэкбокса было сочтено, что этот подход слишком сложен и неудобен. Вместо этого была осуществлена специальная реализация проектной схемы наблюдатель, в котором типичная программа никогда не нуждается в прямом доступе к объектам

элементов управления. Они совершенно абстрагированы. Программист имеет дело только с наблюдаемыми данными. Чтобы сделать определение и изменение этих данных как можно удобнее, они просто являются глобальными переменными языка Компонентный Паскаль, называемые *interactor*. Элемент управления имеет символическое имя (например, "TextCmds.find.ignoreCase"), которое позволяет ему найти свою переменную, используя встроенные возможности метапрограммирования BlackBox'a. В самом модуле интерактора нет ссылки на элемент управления.

Когда элемент управления редактируется, он прозрачно использует стандартный механизм распространения изменений Блэкбокса, т.е. он посылает сообщения отображения, которые уведомляют другие элементы управления, которые могут показывать отображать ту же переменную. Хотя модуль интерактора не имеет доступ к своим элементам управления, он все же может влиять на то, как показываются элементы управления (например, доступность/недоступность), и на способ их взаимодействия друг с другом. Это делается экспортом подходящих процедур *охранника* и *уведомителя*. Они присоединяются к элементу управления таким же образом, как и ссылка на интерактор, используя подходящий редактор свойств элементов управления. Важно, что программа не нуждается в прямом доступе к элементам управления, и нет необходимости программировать прямые взаимодействия элементов управления. Это значительно упрощает реализацию и поддержку сложных поведений пользовательского интерфейса. Этот механизм более подробно описан в Главе 4.

В принципе, каркас получает доступ к модулю интерактора (через метапрограммирование) как *одиночный посредник*.

3 Приемы проектирования

В предыдущих главах были описаны различные приемы и подходы к проектированию. Они помогают решить проблемы, которые возникают в составных интерактивных приложениях. В этой главе обсуждаются более общие аспекты, которые помогают создать компонентно-ориентированный каркас, расширяемый за счет динамически загружаемых блэкбоксовых компонентов.

Мы не пытаемся создать полный метод проектирования, который бы давал пошаговые инструкции, как проектировать новый каркас. Это нереально. Но мы хотим продемонстрировать и обосновать приемы, подходы и образцы, которым следовали при проектировании Компонентного Каркаса Блэкбокса. Некоторые из приемов проектирования даже привели к включению специфических "каркасных" функций в язык Компонентный Паскаль. Вначале обсуждается языковая поддержка проектирования каркаса, затем обеспечение компонент Компонентного Паскаля и правил, которые регулируют компонентное сотрудничество в Блэкбоксе.

3.1 Языковая поддержка

Каркас представляет собой коллекцию образцов проектирования, приведенную к нотации конкретного языка программирования. Поэтому выразительные возможности языка, в особенности его подмножества определения интерфейса, оказывают главное влияние на то, какая доля проекта каркаса может быть воплощена напрямую в коде. Выражение архитектурных решений и деталей проекта явно в языке важно для уменьшения рисков, связанных с переделками каркаса. Реконструкция каркаса и его компонентов расширения позволяет избежать превращения старых архитектур в хрупкие структуры, грозящие рухнуть под их собственным весом. Предупреждение архитектурной деградации является ключом к поддержанию программных систем продуктивными долгое время, особенно если ПО состоит из компонентов, которые меняются, добавляются или удаляются с течением времени по нарастающей.

Каркас представляет архитектуру для решения определенного класса проблем. Хорошая архитектура использует минимальное число типовых (базовых) элементов, где бы они ни

применялись. Последовательное применение типовых элементов упрощает документирование каркаса и облегчает его понимание.

Типовые элементы являются абстрактными решениями проблемы. Для построения каркаса они должны быть сформулированы в терминах конкретного языка программирования. Язык программирования оказывает большое влияние на то, как типовой элемент может быть представлен и как могут поддерживаться его логичность и постоянство.

Есть две основных "философии" построения языка программирования. Одна из них представлена языком Си. Си есть сжатый язык системного программирования, который позволяет легко и эффективно манипулировать структурой памяти данных. Другой подход представлен Паскалем. Паскаль - это читабельный язык прикладного программирования, который обеспечивает высокий уровень безопасности.

Подход Си особенно хорош для написания низкоуровневого кода, такого, как драйверы устройств. Когда Си стал все больше применяться для проектирования приложений, возникла идеология безошибочного программирования *(zero errors ideology)*, которая провозглашает, что, поскольку ошибки программирования не должны случаться, они не будут случаться. "Настоящий программист" не делает ошибок и поэтому нет нужды в средствах защиты, "давящих" на него со стороны языка. Функции безопасности, такие, как контроль переполнения индекса, нужны начинающим программистам, а для профессионалов они только помеха.

Однако современные психологические исследования показывают, что люди все время делают ошибки, независимо от того, пишут ли они программы, выполняют математические доказательства, пилотируют самолеты или оперируют больного. Ошибки допускаются независимо от того, признаются они или нет. Важно понимать, что большинство ошибок тем легче исправить, чем раньше их удается выявить. Обнаружение ошибок работает лучше в команде, взаимодействующей открыто, где член команды не боится двойного контроля его работы другими (выявляющими ошибки). Хорошие авиапредприятия отрабатывают коллективные действия своих пилотов в условиях стресса. Некоторые прогрессивно мыслящие клиники используют похожий тренинг для хирургов и их ассистентов. Они отбросили идеологию безошибочной работы в своих областях.

В мире программирования в последние десять лет преобладает обратная тенденция в силу того, что язык Си стал основным для всех видов программ. В терминах техники программирования и осознания безопасности это был огромный шаг назад от современного состояния. Большие компании пытаются ограничить потери навязыванием дорогостоящих инструментов, которые привносят лишь ограниченный уровень безопасности, вместо решения проблем там, где они (решения) стоят намного меньше, а именно, на уровне языка.

Но, в то время как программисту трудно признать, что делает ошибки, ему намного легче признать, что другие программисты делают ошибки. Это стало очевидно с появлением Интернета. Интернет позволяет легко, а иногда даже автоматически загрузить и выполнить маленькие программы (аплеты) с неизвестных сайтов. Этому коду вообще оказалось нельзя доверять. К счастью, это вынудило большие компании пересмотреть языки с точки зрения безопасности. Фактически, требования безопасности явились причиной выдвижения языка Java на первое место. Поверхностно Java похож на C++, но, в отличие от Си и C++, он является полностью защищенным по типам, как Компонентный Паскаль.

В Компонентном Паскале объекты и их классы (типы записей) могут быть полностью спрятаны в модуле или они могут быть полностью экспортированы, т.е. они и их части (поля записи / отдельные переменные) могут быть полностью видимы вне своего определяющего модуля. На практике полезно иметь еще больше контроля. По этой причине Компонентный Паскаль позволяет определить для каждого поля записи, является ли оно полностью экспортируемым, экспортируемым только для чтения или скрытым. Следующий пример (листинг 3-1) показывает, как звездочка ("*") применяется для экспорта и минус ("-") для более ограниченного экспорта "только для чтения":

```
MODULE ObxSample;

  TYPE
    File* = POINTER TO RECORD
      len: INTEGER   (* пример скрытой переменной *)
    END;

    Rider* = POINTER TO RECORD
      (* разрешается "садить" несколько бегунков на один файл *)
      file-: File;   (* пример переменной "только для чтения" *)
      eof*: BOOLEAN;   (* пример переменной полного экспорта *)
      pos: INTEGER   (* пример скрытой переменной *)
        (* Инвариант: (pos >= 0) & (pos < file.len) *)
    END;

  PROCEDURE (f: File) GetLength* (OUT length: INTEGER), NEW;
  BEGIN
    length := f.len
  END GetLength;

  PROCEDURE (rd: Rider) SetPos* (pos: INTEGER), NEW;
  BEGIN
    (* защита инвариантов, чтобы ошибки не могли проскочить сквозь компоненты *)
    ASSERT(pos >= 0); ASSERT(pos < rd.file.len);
    rd.pos := pos
  END SetPos;
  ...

END ObxSample.
```

Листинг 3-1. Образец модуля на Компонентном Паскале

Язык Java не поддерживает экспорт только для чтения, но поддерживает защищенные поля, которые являются видимыми только расширениям класса, но не нормальным клиентам. Это свойство уместно, только если наследование выполнения используется с пересечением модульных границ. По причинам, которые лежат за рамками этого текста, наследование выполнения не является хорошей идеей в блэкбоксовых абстракциях, таких, как компоненты, и поэтому не может нормально переноситься через границы компонент или в интерфейсах компонент каркаса. Защищенный экспорт поэтому не поддерживается в Компонентном Паскале.

Безопасность означает разные вещи для разных людей. С одной стороны, C++ может считаться безопаснее Си. С другой стороны, Интернет вызывает озабоченность, которая выходит за пределы просто безопасности; должна также предусматриваться защита (от несанкционированного доступа, криминальных атак и т.д.). Предметами охраны являются способы авторизации и обычно имеют дело на уровне операционной системы или оборудования. Функции защиты более высокого уровня являются предметом библиотечных определений и исполнения. Однако сопутствующих потерь из-за чрезмерного проникновения сквозь аппаратные механизмы защиты можно избежать, если строить защиту на строгих свойствах безопасности используемого языка(ов), что возвращает нас к вопросу о безопасности. Но что точно есть "безопасность"?

Если коротко, безопасность языка программирования означает, что его спецификация позволяет указывать инварианты, и что его выполнение гарантирует, что эти инварианты поддерживаются. (Возможность доверенного выполнения должна предполагаться, эта уверенность должна подтверждаться выживанием после атак.) Коротко: *безопасность есть "инварианты, принятые всерьез".*

Какие типы инвариантов мы рассматриваем? Большинство фундаментальных инвариантов есть инварианты памяти: память, занятая переменными, может использоваться только таким путем, который допускает язык (например, описание типа). На практике это означает, что произвольный выбор типа должен быть запрещен и что ручное удаление динамических структур данных (освобождение памяти) не должно допускаться, поскольку освобождение может случиться слишком рано и некоторая часть все еще используемой памяти может быть снова распределена и, таким образом, будет использоваться одновременно двумя независимыми переменными. Такие висячие указатели гибельны, но их можно полностью избежать, если вместо ручного освобождения используется автоматическая сборка мусора. Поэтому Java и Компонентный Паскаль используют автоматическую сборку мусора.

Инварианты памяти фундаментальны. Более специфичные для приложения инварианты обычно устанавливаются поверх нескольких взаимодействующих объектов. Например, объект, который представляет путь к файлу, всегда должен иметь текущую позицию, которая лежит внутри файла, где файл представляется вторым объектом. Этот инвариант связывает два объекта (и таким образом классы). Только это может гарантировать, что если язык программирования позволяет определить взаимодействие между двумя объектами, которые являются частными для него, то внешний код не может на него повлиять.

Заметим, что это отличается от "защищенного" отношения, рассмотренного выше. В отличие от установки отношения между классом и его неизвестными еще подклассами, потомок здесь является классом и который был построен совместно и всегда будет использоваться в связи. Модуль или пакет модулей является подходящим структурированным средством, которое позволяет определить такие свойства безопасности наивысшего уровня. Пакеты Java в этом отношении не идеальны. Поскольку пакеты Java являются *открытыми модулями*, новые классы могут быть добавлены в пакет во время исполнения (at run-time), и могут полностью нарушать инварианты, которые были установлены ранее. Такое дополнение "чужих" классов в пакет необходимо защищать средствами периода исполнения, которые не подчиняются языковым определениям. В Java это связано с концепцией уникального владения: каждый пакет, по идее, принадлежит его источнику и только этот единственный источник имеет право добавлять новые классы в пакет. В любом случае пакет Java является улучшением по сравнению с многими другими объектно-ориентированными языками, например, полностью неструктурированным подходом дружественных классов в C++, или полным отсутствием подходящего конструкта в стандартном Smalltalk.

В противоположность пакетам Java Компонентный Паскаль поддерживает *закрытые модули,* или, для краткости, модули. В Компонентном Паскале модуль является отдельной единицей компиляции, загрузки и сокрытия информации.

Применение строгой типизации и сокрытия информации облегчает обнаружение ошибок на возможно более ранней стадии, когда их легко и недорого исправить. Это ценно, поскольку позволяет повысить программную устойчивость.

Но преимущество типовой и модульной безопасности простирается дальше. Оно обеспечивает большую *гибкость*. На первый взгляд это кажется удивительным. Почему ограничения, такие как типы (которые ограничивают операции над переменными) и модули (которые ограничивают видимость) создают что-то еще, кроме уменьшения гибкости? Причина этого *парадокса переработки (refactoring paradoxon)* двоякая. С одной стороны, всё, что полностью скрыто в модуле, может быть изменено вмешательством в этот единственный модуль. Локальные изменения в скрытой части модуля никак не отдаются за его пределами. Даже не требуется перекомпиляция других модулей-клиентов. С другой стороны, когда строго типизированный интерфейс по какой-то причине меняется, простая перекомпиляция выявит несоответствие в используемом интерфейсе, например, изменение типа или имени метода. Компилятор, таким образом, реально может повысить уверенность в программной системе, которая была модифицирована некоторыми интерфейсными

изменениями.

Контролируя типизированный интерфейс на аккуратно выбранном уровне детализации (разбиения на модули), можно достичь баланса между бинарной совместимостью версий и выявлением определенных несоответствий. Компонентный Паскаль поддерживает строгую типизацию, позволяющую выявлять такие несоответствия. Например, вновь введенный метод должен быть помечен как *NEW*, и *VAR* параметры могут быть уточнены как *IN* или *OUT* параметры. В Java, например, невозможно различить новый метод, попытку перезагрузки или переопределения (overriding). Ошибка в имени метода, изменении базы или подкласса, или комбинация несовместимых версий может приводить к ошибкам, которые трудно локализовать.

Каркас обычно предопределяет интерфейсы для расширений каркаса. Каркас даже может содержать код, который использует эти интерфейсы, хотя собственно исполнение еще не существует. Вызов кода, расположенного «выше» в модульной иерархии, типично для объектно-ориентированных каркасов. Язык может поддерживать эту типичную для каркаса схему потока управления через определенную форму интерфейсов, которые являются абстрактными классами, которые можно применять где угодно. Компонентный Паскаль поддерживает записи абстрактных типов с единственным наследованием таким образом, что возможен широкий спектр от полностью абстрактных до полностью конкретных типов. Java похож на Компонентный Паскаль в этом отношении, за тем исключением, что он дополнительно поддерживает конструкт раздельный интерфейс. Интерфейс Java - это то же, что и полностью абстрактный класс, за исключением того, что он допускает множественное (интерфейсное) наследование.

```
MODULE TestViews;

  TYPE
    View* = POINTER TO ABSTRACT RECORD
      (* частично абстрактный тип  *)
      context-: Context;
      ... some hidden fields ...
    END;

    Context* = POINTER TO ABSTRACT RECORD END;
    (* полностью абстрактный тип *)

    PROCEDURE (v: View) Restore* (l, t, r, b: INTEGER), NEW, ABSTRACT;

    ...

END TestViews.
```

Листинг 3-2. Полуабстрактный и абстрактный типы записей в Компонентном Паскале

Инварианты являются свойствами, которые предположительно остаются неизменными. Тогда возникает вопрос, может ли подтип (подкласс / расширенный тип) произвольно менять поведение своего базового типа. В объектно-ориентированных языках такие модификации могут быть достигнуты поправками [?, overriding] унаследованных методов. Java позволяет делать классы или методы финальными [*final*], давать проектировщику каркаса возможность предотвратить любой вид нарушения инварианта через "заднее крыльцо" правок [?, overriding].Типы записей Компонентного Паскаля являются финальными по умолчанию, они могут быть явно помечены как расширяемые.

Компонентный Паскаль превосходит Java в нескольких других языковых конструктах. Один — *ограниченные [limited]* записями. Это записи, которые могут быть расширены и размещены только внутри описывающего их модуля. С точки зрения перспективы импорта

модулей ограниченные типы являются финальными и даже не размещаемыми. Это позволяет гарантировать, что все размещения происходят централизованно в определяющем (каркасе) модуле, что дает этому модулю полный контроль над инициализацией. Например, это позволяет предлагать фабричные [разработчиками] функции, которые размещают объект и инициализируют его разными способами, устанавливая инварианты перед тем, как объект попадет в клиентский модуль. Этот механизм гибче и проще, чем конструкторы, которые используются в Java.

Экспорт "только для исполнения" - это главный пример свойства, являющегося причиной типичных шаблонов проектирования каркаса. Методы типа запись могут быть экспортированы "только для исполнения" путем указания дефиса вместо звездочки. Метод "только для исполнения" может вызываться только внутри определяющего модуля. Но он может быть *исполнен* [*implemented*] вне определяющего модуля в исполняющем компоненте каркаса. Например, механизм сохранения Компонентного Каркаса Блэкбокса использует это свойство, чтобы защитить методы *загрузки* и *выгрузки* [*Internalize* и *Externalize*] от вызова их вне корректного контекста. По существу, каркас (в данном случае модуль *Stores*) использует методы "только для исполнения" всеми возможными законными способами (выполняет все законные способы применения) и экспортирует их только для целей исполнения.

Для методов, которые представляют интерфейсные варианты, поддерживается атрибутный метод *EMPTY (пустой).* Пустой метод есть полностью абстрактная ловушка, которая может быть использована в расширении, но в отличие от абстрактных методов ее не требуется использовать (?). Например, образ [вид, view] имеет пустой метод *HandleCtrlMsg*, который необходимо применять только в образах, которые реагируют на пользовательский ввод, например, через мышь или клавиатуру.

В книге Design Patterns (Шаблоны проектирования) ["Design Patterns, Elements of Reusable Object-Oriented Software"; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley, 1994; ISBN 0-201-63361-2] обозначен список общих проблем проектирования. Эти проблемы часто ведут к необходимости перепроектирования, поскольку ПО не дает достаточной свободы для внесения изменений. Некоторые из этих проблем решены в Компонентном Паскале:

- *Создание объекта указанием класса явно*

Модули позволяют скрывать классы. Скрытый класс не может быть прямо подтвержден клиентским модулем, поскольку он там невидим. Это позволяет навязать множество непрямых механизмов размещения. Абстрактные типы записей позволяют отделить интерфейсы от исполнения, так что абстрактные типы записей могут быть экспортированы без риска прямого размещения клиентами.

Как и абстрактные типы, ограниченные [limited] типы не могут быть напрямую размещены клиентами.

Экспорт "только для исполнения" ограничить последовательности размещения и инициализации определяющим модулем, что также предохраняет клиентов от обязанности самим уточнять классы прямым подтверждением их.

Эти свойства можно комбинировать для разработки безопасных исполнений всеми создающими шаблонами, описанными в *Design Patterns:* abstract factories, builders, factory methods, prototypes и singletons.

- *Зависимость от некоторых операций*

Переменные типа статическая запись могут использоваться как легковесные (размещенные на стеке) объекты сообщений, вместо использования жестко кодированных сигнатур методов (?) или тяжеловесных (размещенных на куче) объектов сообщений. Записи сообщений могут быть пересланы, отфильтрованы, опубликованы и так далее. Но они не могут нарушить безопасность типа, поскольку адресация статических записей не может

меняться небезопасным способом.

Записи сообщений могут использоваться для выполнения цепочки обязательств и шаблонов наблюдателя (?), которая развязывает отправителя сообщения и его получателя(ей).

- *Зависимость от оборудования и программных платформ*

- *Зависимость от представлений или исполнений объекта*

Переносимость - это одно из главных преимуществ использования настоящего языка высокого уровня. Компонентный Паскаль абстрагируется от обеспечивающего его оборудования, кроме того, его семантический интервал достаточно мал, чтобы мог быть сгенерирован очень эффективный машинный код. Как в Java, размеры типов определяются так, что перенос данных между машинами не создает проблем.

- *Алгоритмические зависимости*

Части алгоритма могут заменяться с использованием абстрактных методов, пустых (hook) методов или простых объектов. Экспорт "только для исполнения" делает возможным безопасное назначение обязанностей: последовательности корректных вызовов методов должны выполняться определением модуля, исполнение методов должно выполняться расширяющим программистом. Это свойство позволяет разрабатывать безопасные исполнения всех поведенческих шаблонов (строитель, внутренний итератор, стратегия, временный метод и так далее).

- *Плотное сопряжение*

Модули действуют как границы видимости. Поэтому плотное сопряжение возможно, где оно необходимо (внутри модульного исполнения), тогда как неплотное сопряжение возможно на модульных границах. Это упрощает безопасное исполнение в принципе все шаблонов проектирования.

- *Расширение функциональности подклассами*

Классы могут быть скрыты в модулях. Финальные или ограниченные классы и финальные методы могут экспортироваться без риска того, что могут сделать подклассами или исказить.

- *Неспособность удобно менять классы*

Строгая типизация, атрибуты записи и метода, такие, как NEW, и утверждения (assertions) периода исполнения повышают уверенность, что интерфейс может быть изменен таким образом, что все клиенты снова могут быть сделаны согласованными легко и надежно. Это основа программного проектирования и поэтому важна для всех видов проектов.

Все эти проблемы имеют одну общую тему: локальное изменение может вызвать эффект расходящихся волн, который отразится на программной системе в целом. Хорошие языки и приемы проектирования помогают проектировать с учетом возможности изменений в будущем:

Гарантировать, что возможные проектные изменения имеют только локальные эффекты.
Где это невозможно, гарантировать, что эффекты обнаруживаются компилятором.
Где это невозможно, гарантировать, что эффекты обнаруживаются во время исполнения как можно раньше.

Ясно, что еще остается существенный разрыв между современными языками, такими как Компонентный Паскаль, и языками полных спецификаций, которые также позволяют задать семантику программы. В будущем придется закрыть такие части этого разрыва, которые имеют достаточно хорошее отношение цена/качество, т.е. помогают изменить поведение

компонент контролируемым образом, без того, чтобы программы становились неудобоваримыми для средних программистов.

3.2 Модули и подсистемы

В этом разделе мы опишем более конкретно, как выглядит код Компонентного Паскаля и как он управляется Блэкбоксом. Для этой цели мы должны более детально рассмотреть особенности Блэкбокса, чем в других разделах этой книги.

Весь код, написанный на Компонентном Паскале, «живет» внутри модулей. Модуль — единица компиляции Компонентного Паскаля. Модуль имеет интерфейс и скрытую реализацию. Синтаксически это достигается тем, что некоторые элементы в модуле, например, некоторые типы и процедуры, помечаются как *экспортируемые*. Всё, что не экспортировано, невидимо и напрямую недоступно извне модуля. Если модулю нужен сервис, предоставляемый одним или несколькими другими модулями, он *импортирует* их. Поэтому модуль объявляет, что ему требуются описания интерфейсов импортированных модулей во время компиляции и подходящие реализации во время выполнения. Листинг 3-3 показывает возможную реализацию модуля, названного *ObxPhoneDB.* Он экспортирует три процедуры для поиска данных в телефонном справочнике: по индексу, по имени и по номеру. Обратите внимание на звездочки, которые помечают элементы как экспортируемые, в нашем случае это тип *String* и три процедуры поиска:

```
MODULE ObxPhoneDB;

  CONST
    maxLen = 32;   (* максимальная длина строк имя/число *)
    maxEntries = 5;   (* максимальное число элементов в базе данных *)

  TYPE
    String* = ARRAY maxLen OF CHAR;

    Entry = RECORD
      name, number: String
    END;

  VAR db: ARRAY maxEntries OF Entry;

  PROCEDURE LookupByIndex* (index: INTEGER; OUT name, number: String);
  BEGIN   (* дан индекс, возвращается соответствующая пара <name, number> *)
    ASSERT(index >= 0);
    IF index < maxEntries THEN
      name := db[index].name; number := db[index].number
    ELSE
      name := ""; number := ""
    END
  END LookupByIndex;

  PROCEDURE LookupByName* (name: String; OUT number: String);
    VAR i: INTEGER;
  BEGIN   (* дано имя, поиск соответствующего номера телефона *)
    i := 0; WHILE (i # maxEntries) & (db[i].name # name) DO INC(i) END;
    IF i # maxEntries THEN   (* имя найдено в db[i] *)
      number := db[i].number
    ELSE   (*имя не найдено в db[0..maxEntries-1] *)
      number := ""
    END
  END LookupByName;
```

```
  PROCEDURE LookupByNumber* (number: String; OUT name: String);
    VAR i: INTEGER;
  BEGIN  (* дан номер телефона, поиск соответствующего имени *)
    i := 0; WHILE (i # maxEntries) & (db[i].number # number) DO INC(i) END;
    IF i # maxEntries THEN  (* номер найден в db[i] *)
      name := db[i].name
    ELSE  (* номер не найден в db[0..maxEntries-1] *)
      name := ""
    END
  END LookupByNumber;

BEGIN  (* инициализация содержимого справочника *)
  db[0].name := "Даффи Дьюк"; db[0].number := "310-555-1212";
  db[1].name := "Виль Е. Койот"; db[1].number := "408-555-1212";
  db[2].name := "Скрудж МакДьюк"; db[2].number := "206-555-1212";
  db[3].name := "Хью Льюис"; db[3].number := "415-555-1212";
  db[4].name := "Томас Деви"; db[4].number := "617-555-1212"
END ObxPhoneDB.
```

```
Даффи Дьюк          310-555-1212
Виль Е. Койот       408-555-1212
Скрудж МакДьюк       206-555-1212
Хью Льюис           415-555-1212
Томас Деви          617-555-1212
```

Листинг 3-3. Исполнение of ObxPhoneDB

Перед тем, как модуль использовать, его необходимо загрузить с диска в оперативную память. Но перед тем, как он может быть загружен, его необходимо откомпилировать (команда *Разработка -> Компилировать*). Компилируя модуль, компилятор порождает кодовый файл и символьный файл. Кодовый файл содержит исполняемый код, который может быть загружен в память. Кодовый файл есть разновидность суперлегкой DLL (динамически загружаемой библиотеки). Компилятор также производит символьный файл, который содержит бинарное представление интерфейса модуля. Если модуль импортирует другие модули, компилятор читает символьные файлы этих модулей, чтобы проверить, что их интерфейсы используются корректно. Процесс компиляции можно изобразить так:

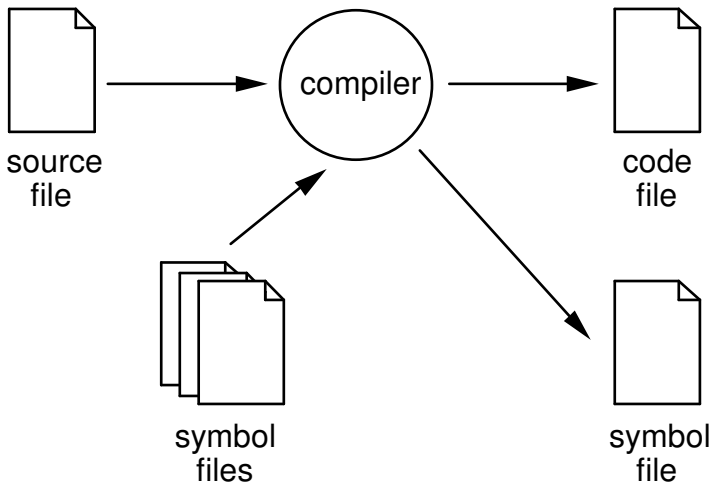

Рис. 3-4. Процесс компиляции

Когда вы компилируете модуль первый раз, генерируется новый символьный файл. В

рабочем журнале компилятор пишет примерно следующее:

```
компилируется "ObxPhoneDB"
  new symbol file   564   640
```

Первые два числа показывают, что машинный код в новом кодовом файле имеет длину 564 байта.

Второе число показывает, что модуль содержит 640 байт глобальных переменных (пять входов в переменной db; каждый вход состоит из двух строк по 32 элемента в каждой; каждый элемент — 2-байтная литера в кодировке Юникод). Если символьный файл точно для такого интерфейса уже существует, компилятор выдает более короткое сообщение:

```
компилируется "ObxPhoneDB"  564   640
```

Если интерфейс изменился, компилятор пишет новый символьный файл и указывает в рабочем журнале, какие изменения появились по сравнению со старой версией. Например, если вы только что изменили процедуру *LookupByNumber*, компилятор пишет:

```
компилируется "ObxPhoneDB"
  LookupByNumber is new in symbol file   964   640
```

Символьный файл используется только во время компиляции, он не используется во время выполнения. Чтобы загрузить модуль, достаточно его кодового файла. Модули загружаются динамически, т.е. нет отдельного шага линкования, как в большинстве статических языков. Чтобы увидеть список загруженных модулей, выполните *Инфо->Загруженные модули*. Как результат, откроется окно с содержимым, похожим на следующее:

имя модуля	*байт*	*клиенты*	*скомпилирован*	*загружен* Обновить
StdLinks	20639	1	2.7.1996 18:42:15	29.8.1996 14:31:14
StdFolds	20425	1	2.7.1996 18:41:33	29.8.1996 14:31:12
StdCmds	25066	7	2.7.1996 18:39:12	29.8.1996 14:31:00
Config	125	0	2.7.1996 18:38:21	29.8.1996 14:31:20
Init	682	0	2.7.1996 18:40:21	29.8.1996 14:31:05
Controls	78876	5	7.7.1996 14:14:58	29.8.1996 14:31:00
Services	1472	5	2.7.1996 18:37:14	29.8.1996 14:30:54
Containers	37348	40	2.7.1996 18:37:51	29.8.1996 14:30:52
Properties	8337	42	2.7.1996 18:37:40	29.8.1996 14:30:49
Controllers	6037	42	2.7.1996 18:37:36	29.8.1996 14:30:49
Views	31589	49	2.7.1996 18:37:33	29.8.1996 14:30:49
Models	4267	50	2.7.1996 18:37:27	29.8.1996 14:30:48
Converters	2189	51	14.7.1996 22:45:12	29.8.1996 14:30:48
Dialog	8979	54	2.7.1996 18:37:13	29.8.1996 14:30:48
Dates	3848	45	2.7.1996 18:37:07	29.8.1996 14:30:48
Meta	19275	11	2.7.1996 18:37:10	29.8.1996 14:30:48
Stores	22302	53	2.7.1996 18:37:22	29.8.1996 14:30:47
Strings	17547	15	2.7.1996 18:37:05	29.8.1996 14:30:47
Math	15408	2	3.7.1996 1:45:05	29.8.1996 14:30:47
Ports	10631	56	2.7.1996 18:37:17	29.8.1996 14:30:46
Fonts	1589	58	2.7.1996 18:37:15	29.8.1996 14:30:46
Files	3814	62	2.7.1996 18:36:28	linked
...				

Таблица 3-5. Список загруженных модулей

Список показывает все загруженные модули. Для каждого модуля показан его размер, сколько других модулей он импортирует, когда он был откомпилирован и когда он был загружен.

Легко получить список уже загруженных модулей, но как насчет еще не загруженных? Желание иметь доступ к большому числу уже готовых компонентов ставит некоторые организационные вопросы. Как точно определить, какие компоненты доступны? Как определить, какие из доступных компонентов делают то, что нужно?

Это предмет соглашений, документации и поддерживающего инструментария. В Блэкбоксе решено, что коллекция связанных компонентов, называемая *подсистема*, размещается в отдельным папках; все они находятся в корневой папке Блэкбокса. Там есть такие подсистемы, как System, Std, Host, Mac, Win, Text, Form, Dev или Obx. Полная коллекция подсистемы называется *депозитарий* (хранилище) Блэкбокса. Идея, лежащая в основе структуры депозитария, состоит в том, чтобы всё, что принадлежит компоненту (кодовые файлы, символьные файлы, документация, исходные тексты) сохранялось вместе в систематической и простой структуре папок в соответствии с правилом

Держать все составляющие компонента в одном месте.

Для компонентно-ориентированного ПО единственно подходящим является вариант, когда добавление и удаление компонентов может выполняться по частям, добавлением или удалением одной папки. Все виды механизмов централизованной инсталляции или регистрации, которые распространяют составляющие компонента, должны *быть исключены, поскольку* они неизбежно ведут к (ненужным) проблемам управления.

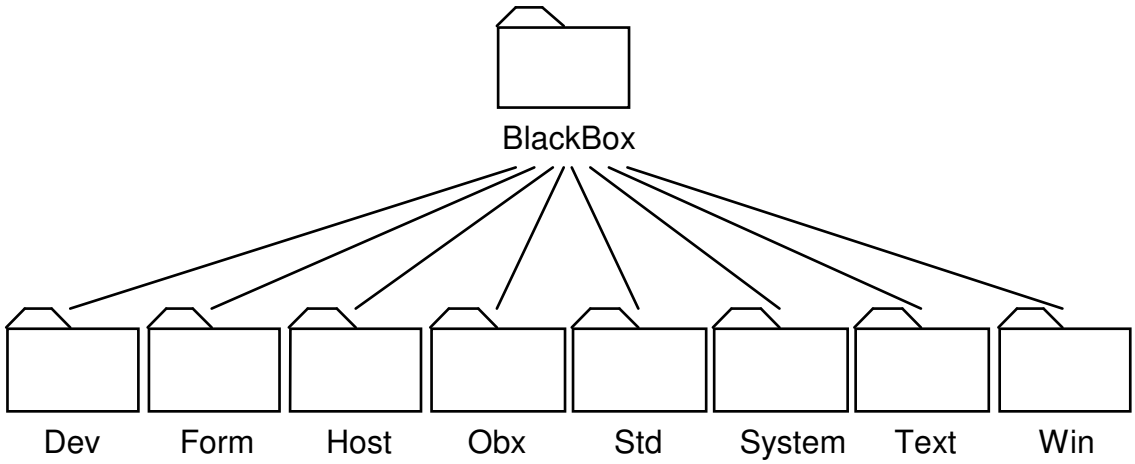

Рис. 3-6. Стандартные подсистемы Блэбокса

Каждая папка подсистемы, как Obx, может содержать следующие подпапки:

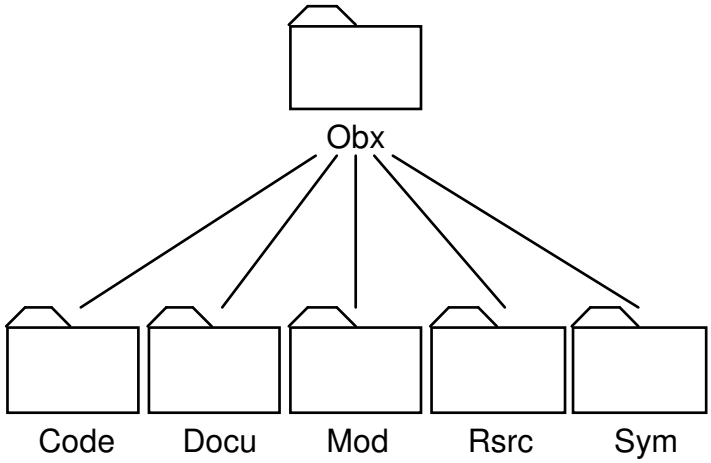

Рис. 3-7. Структура типичной папки подсистемы

Исходный модуль сохраняется в папке Mod подсистемы. Имя файла соответствует имени модуля, но без префикса подсистемы; например, модули ObxPhoneDB и ObxPhoneUI сохранены как Obx/Mod/PhoneDB и Obx/Mod/PhoneUI соответственно. Для каждого исходного файла может быть соответствующий символьный файл, например, Obx/Sym/PhoneDB; соответствующий кодовый файл Obx/Code/PhoneDB; и соответствующий файл документации, например, Obx/Docu/PhoneDB. Также может быть 0 или более ресурсных документов в подсистеме, например, Obx/Rsrc/PhoneUI. Причем нет необходимости в соотношении 1:1 между модулями и ресурсами, хотя обычно рекомендуется использовать имя модуля как часть имени ресурса для упрощения поддержки.

Модули, чьи имена имеют форму SubMod, такие, как *TextModels*, *FormViews* или *StdCmds*, сохраняются в их соответствующих подсистемах, заданных префиксами имен, например, *Text*, *Form* или *Std*. Имя подсистемы начинается с заглавной буквы и далее могут идти несколько заглавных и затем несколько строчных букв или цифр. Первая заглавная буква после этого обозначает отдельный модуль в подсистеме.

Модули, которые не принадлежат подсистеме, т.е. модули, имена которых не имеют формы SubMod, сохраняются в специальной подсистеме, названной *System*. Вся библиотека Блэкбокса и ядро каркаса принадлежат к этой категории, это модули *Math*, *Files*, *Stores*, *Models* и т.д.

Каждая папка подсистемы может содержать следующие вложенные папки:

Code Папка с исполняемыми кодовыми файлами, т.е. легкими DLL.
Например, для модуля "FormCmds" там есть файл "Form/Code/Cmds".
Модуль для связи с настоящими DLL ОС (Windows DLL или кодовые фрагменты Mac OS) не имеют кодового файла.

Docu Папка с полностью документированными интерфейсами и другими docu.
Например, для модуля "FormCmds" там есть файл "Form/Docu/Cmds".
Для модуля, который используется только внутренне (для служебных целей Блэбокса), его файл docu не поставляется.
Часто есть общие файлы документации, которые не относятся к отдельному модулю подсистемы. Такие файлы содержат один или более дефисов в их именах, например, "Dev/Docu/P-S-I".
Типичными файлами являются
"Sys-Map" (обзор гиперссылок на другие документы этой подсистемы)
"User-Man" (руководство пользователя)
"Dev-Man" (руководство разработчика)

Mod Папка с исходными модулями.
Например, для модуля "FormCmds" там есть файл "Form/Mod/Cmds".

	Для модуля, который не публикуется в исходном виде ("белый ящик" = "white box"), его исходный текст не поставляется пользователям.
Rsrc	Папка с ресурсными документами подсистемы. Например, для модуля "FormCmds" там есть файл "Form/Rsrc/Cmds". Может быть ноль, один или более ресурсных файлов для одного документа. Если есть несколько файлов, второй получает суффикс "1", третий суффикс "2" и так далее. Например, "Form/Rsrc/Cmds", "Form/Rsrc/Cmds1", "Form/Rsrc/Cmds2" и т.д. Часто есть добавочные файлы, которые не относятся к отдельному модулю подсистемы. Типичными файлами являются "Strings" (строковые ресурсы этой подсистемы) "Menus" (меню этой подсистемы)
Sym	Папка с символьными файлами. Например, для модуля "FormCmds" там есть файл "Form/Sym/Cmds". Для модуля, который используется только внутренне, его символьный файл не поставляется.

Таблица 3-8. Содержание стандартных папок подсистемы

Если вы что-то хотите найти в депозитарии, его подсистемах или их папках, вы можете выполнить команду *Инфо->Обзор подсистем.* Если вы хотите найти больше сведений о модуле, вы можете выделить имя модуля в тексте и затем выполнить *Инфо->Исходный текст,* или *Инфо->Документация.* Эти команды открывают модульные файлы Mod, Sym или Docu. Для этой цели *Инфо->Интерфейс* преобразует бинарное представление символьного файла в читабельное текстовое описание. Например, наберите строку "ObxPhoneDB" в рабочем журнале, выделите строку и затем выполните команду просмотра *Инфо->Интерфейс.* В результате в новом окне будет открыт следующий текст:

```
DEFINITION ObxPhoneDB;

  TYPE
    String = ARRAY 32 OF CHAR;

  PROCEDURE LookupByIndex (index: INTEGER; OUT name, number: String);
  PROCEDURE LookupByName (name: String; OUT number: String);
  PROCEDURE LookupByNumber (number: String; OUT name: String);

END ObxPhoneDB.
```

Листинг 3-9. Определение ObxPhoneDB

Модульное определение, сгенерированное браузером синтаксически, является подмножеством модульного исполнения, за тем исключением, что ключевое слово *MODULE* заменено на *DEFINITION.* Этот синтаксис можно назвать языком описания интерфейса (IDL) Компонентного Паскаля. Тексты в этом языке обычно создаются по полным исходным текстам модулей браузером или аналогичным инструментом, так что нет нужды писать их вручную или компилировать.

Поскольку команды браузера оперируют с символьным файлом модуля, он может быть использован, даже когда еще нет ни настоящей документации, ни кодового файла. Во время прототипирования, когда документация редко имеется, просмотр символьного файла очень удобен для быстрого получения деталей, таких, как сигнатура процедуры, или обзора интерфейса всего модуля. Когда доступна полная документация, может использоваться команда *Инфо->Документация.* Она открывает файл документации модуля,

который начинается с определения модульного интерфейса, точно так, как показано выше, но затем продолжается назначением модуля и детальным описанием различных элементов, экспортированных модулем, например:

```
    Модуль ObxPhoneDB обеспечивает доступ к телефонному справочнику. Доступ может
    быть по индексу, по имени или по номеру. Входные данные могут состоять из литерной
    цепочки, задающей имя или телефонный номер, которая не может быть пустой.
    Минимальный индекс есть 0, и все входы являются смежными.

    PROCEDURE LookupByIndex (index: INTEGER; OUT name, number: ARRAY OF CHAR)
    Возвращает пару <name, number> входа index. Если индекс слишком большой,
    возвращается <"", "">.
    Процедура выполняется за постоянное время.

    Pre
    index >= 0    20

    Post
    index правильный
       name # ""  &  number # ""
    index неправильный
       name = ""  &  number = ""

    PROCEDURE LookupByName (name: ARRAY OF CHAR; OUT number: ARRAY OF CHAR)
    Возвращает телефонный номер, связанный с name, или "", если не найден вход для
    name.
    Процедура выполняется за линейное время, в зависимости от размера базы данных
    (справочника).

    Post
    name найдено
       number # ""
    name не найдено
       number = ""

    PROCEDURE LookupByNumber (number: ARRAY OF CHAR; OUT name: ARRAY OF CHAR)
    Возвращает имя, связанное с number, или "", если не найден вход для number.
    Процедура выполняется за линейное время, в зависимости от размера базы данных
    (справочника).

    Post
    number найдено
       name # ""
    number не найдено
       name = ""
```

Листинг 3-10. Документация ObxPhoneDB

Отметим, что предусловия и постусловия задокументированы в полуформальной нотации. Их целью не является дать полную формальную спецификацию, но помочь сделать простой текст описания менее двусмысленным, где это возможно, без наворотов сложных (и потому нечитабельных) формальных условий. В Блэкбоксе используются следующие номера утверждений для контроля периода исполнения:

Свободны	0 .. 19
Предусловия	20 .. 59

Постусловия	60 .. 99
Инварианты	100 .. 120
Резервировано	121 .. 125
Еще не использовано	126
Резервировано	127

Листинг 3-11. Номера утверждений в Блэкбоксе

Хорошо известно, что определение ошибки тем более затруднено и дороже обходится, чем позже она обнаруживается, т.е. чем дальше разнесены источник и его эффекты. Это позволяет сформулировать правило:

Позволяйте ошибкам проявиться как можно раньше.

В компонентно-ориентированных системах дефекты всегда должны содержаться в их компонентах и не распространяться на другие компоненты. Другие компоненты даже могут быть черными ящиками и не иметь исходных текстов, что делает отладку на уровне исходных текстов невозможной. Более того, поток передачи управления в больших объектно-ориентированных программных системах настолько запутанный, что нереально, и это пустая трата времени, проследить его за пределами границ компонентов для целей отладки.

Единственный жизнеспособный отладочный подход есть проектирование всего, от языка программирования до библиотек, до компонентов и приложений, используя оборонительный стиль программирования. В частности, во входных точках компонент (вызовы процедур/методов) исполнение должно прекращаться, если их предусловия не обеспечены:

Никогда не позволяйте ошибкам проникать сквозь границы компонентов.

К счастью, большинство проверок предусловий недорого и поэтому их отключение во время исполнения программы не имеет смысла. Это важно, поскольку в компонентно-ориентированной системе проверки периода исполнения не могут быть выключены в готовой системе, поэтому разрабатываемая и готовая системы не различаются. На практике большинство компонентов уже отлажено в режиме черного ящика ("продукция"), а другие отлаживаются в режиме белого ящика. Готовые компоненты должны взаимодействовать, чтобы доказать приверженность сформулированному выше правилу, которое означает «никогда не отключать проверки периода исполнения».

3.3 Интерфейсы типа «бутылочное горло»

Один частный шаблон, который мы обсудим здесь, — это шаблон Носитель-Бегунок-Преобразователь (Carrier-Rider-Mapper). Этот шаблон используется несколькими способами в компонентном каркасе Блэкбокса: в текстовой подсистеме, в файловой абстракции, в рамочной абстракции, в абстракции контейнер/контекст и других. Рассмотрим в качестве примера текст.

Пусть текст будет носителем. Носитель — это объект, который несет (содержит) данные, в данном случае текстовые данные. По сути, текст можно рассматривать как линейную последовательность элементов, причем элементы имеют такие атрибуты, как шрифт, цвет, стиль и вертикальный сдвиг (для верхних и нижних индексов). Элементы являются литерами или вьюшками (views). Для любых практических целей вьюшку в тексте можно рассматривать просто как специальный символ.

Когда читается текст, удобно сделать так, чтобы не требовалось указывать позицию для каждого читаемого символа. Такое удобство может быть достигнуто поддержкой *текущей позиции*, т.е. текст сам знает, где читать следующий символ. Каждая операция чтения автоматически увеличивает текущую позицию. Некоторые клиентские объекты могут использовать текстовый носитель одновременно. Например, текстовой вьюшке нужно читать свой текст, когда она отрисовывает текст на экране, а команде меню может понадобиться прочитать текст, чтобы получить входные данные. Эти клиенты независимы

друг от друга, и им незачем друг о друге знать. Поэтому текущая позиция чтения не может сохраняться в самом текстовом носителе, поскольку тогда клиенты были бы в состоянии взаимодействовать друг с другом и потеряли бы независимость.

Чтобы избежать такого взаимодействия, носитель имеет особые процедуры-функции, которые возвращают так называемые *объекты доступа* или, короче, *бегунки* (rider). Бегунок — это независимый канал доступа к данным носителя. Каждый текстовый бегунок имеет собственную текущую позицию и может далее оставаться таковым как текущие атрибуты.

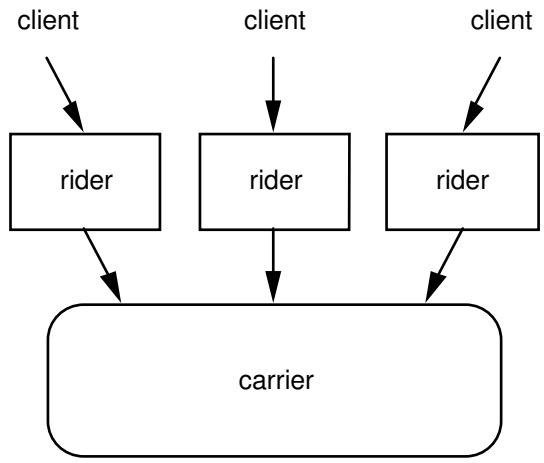

Рис. 3-12. Разделение Носитель-Объект доступа (бегунок) (Carrier-Rider).

Причиной, почему носитель и бегунок разделены в виде разных типов данных, является отношение один-ко-многим между сущностями: для одного носителя может быть произвольное число (включая ноль) бегунков.

Обычно требуется, чтобы носитель и бегунок исполнялись одним и тем же компонентом, поскольку бегунок должен иметь хорошее знание о внутренностях своего носителя. Например, текстовые бегунки и текстовые носители оба используются в модуле *TextModels.* Текстовый бегунок содержит скрытые указатели на внутреннюю структуру данных текстового носителя. Поскольку эта информация полностью скрыта за границами модуля *TextModels,* внешний клиент не может сделать бегунок несовместимым с его носителем. Этот тип инварианта, гарантированный модульной системой Компонентного Паскаля, является примером того, почему сокрытие информации за пределами отдельных классов является важным по соображениям безопасности.

В терминологии проектирования шаблонов текстовый бегунок есть *внутренний итератор*. Однако не все бегунки с необходимостью итераторы. Например, контекстный объект, управляемый контейнером, не является итератором.

Текстовые бегунки бывают двух видов: "анализаторы" (reader) и "синтезаторы" (writer). Анализаторы поддерживают чтение литер и вьюшек, тогда как синтезаторы поддерживают запись литер и вьюшек. Например, возможны такие вызовы:

```
reader.ReadView(view)
```

или

```
writer.WriteChar("X")
```

Часто программисту нужно читать или писать сложные последовательности литер, так что работа на уровне отдельных литер слишком неудобна. Желательны высокоуровневые абстракции для чтения и записи. Это является целью преобразователей (*mappers).* Преобразователь содержит бегунок, который он использует для обеспечения более проблемно-ориентированных операция для записи или для чтения. Поскольку существуют очень разные проблемы программирования, разные приложения могут использовать разных

преобразователей, возможно даже работающих с одним и тем же носителем.
Для текстовых манипуляций Блэкбокс предлагает модуль *TextMappers,* который определяет и исполняет два текстовых преобразователя: *форматер* для записи и *сканер* для чтения.

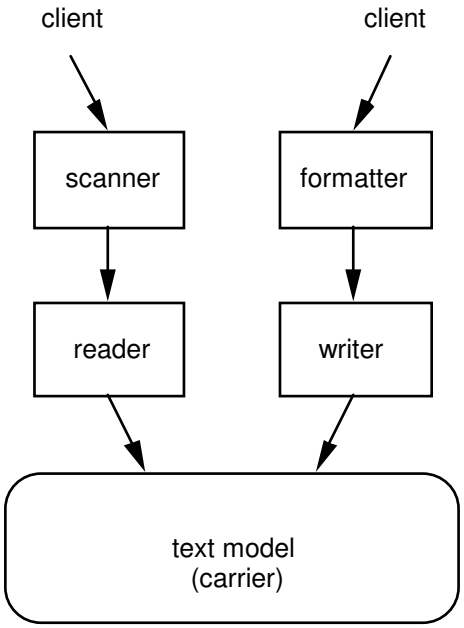

Рис. 3-13. разделение Carrier-Rider-Mapper

Как сканер, так и форматер работают на логическом уровне символов Компонентного Паскаля: с целыми числами, вещественными числами, цепочками литер и т.д. Например, следующие вызовы могут быть:

```
scanner.ReadInt(int)
```

или

```
formatter.WriteReal(3.14)
```

Если есть разные преобразователи для одного вида носителя, они все должны исполняться в терминах интерфейса бегунка. По этой причине интерфейс бегунок/носитель иногда называется интерфейсом узкого горла. Причина, по которой бегунки и преобразователи разделены, не только в отношении один-ко-многим как между носителем и бегунком, так в отношении один-ко-многим между преобразователем и бегунком, но и в независимой расширяемости: особые обстоятельства могут требовать особых преобразователей. Если носители и бегунки строго придерживаются хорошо спроектированного интерфейса доступа (типа узкое горло), можно добавлять новые преобразователи в любое время и использовать их с тем же носителем.

Даже лучше: интерфейс носитель/бегунок может быть абстрактным интерфейсом, так что могут быть разные его *реализации* . Например, файловый носитель может быть исполнен по-разному для файлов на гибком диске, файлов на жестком диске, файлах CD-ROM и сетевых файлов. Но если все они исполняют один и тот же интерфейс, то все файловые преобразователи могут работать со всеми файловыми реализациями. Сравните это с ситуацией, где вы вынуждены заново исполнить все преобразователи для всех носителей: вместо реализации n носителей плюс m преобразователей, вынуждены исполнить n носителей плюс n * m преобразователей! Там, где случается такое расширение в двух измерениях (носитель/бегунок и преобразователь), интерфейс узкого горла необходим, чтобы избежать так называемого *декартова произведения проблем,* т.е. взрыва [количества] реализаций.

Отметим, что интерфейс узкого горла сам не расширяемый, поскольку каждое расширение,

которое не может быть выполнено в терминах интерфейса узкого горла, сводит на нет все его существующие реализации. Например, если вы расширяете драйвера устройства для отображения черно-белой точечной картинки цветным расширением, то все существующие реализации драйвера устройства должны быть обновлены. Поскольку это может быть сделано разными компаниями, это может стать главной проблемой.

Проблема может быть снята, если новое расширение интерфейса определено как опция: тогда клиенты должны тестировать, где опция поддерживается. Если нет, то клиент либо должен сигнализировать, что он не может работать с этим оборудованием, либо он должен изящно деградировать, предлагая более ограниченную функциональность. Если опциональный интерфейс поддерживается и используется, он действует как вид конспирации между реализацией интерфейса и его клиентом. Это приемлемо, но, несомненно, уменьшает комбинации клиентов и реализаций, которые полностью функциональны. Это подчеркивает, насколько важны хорошие проекты узкого горла, чтобы исключить необходимость в последующих расширениях.

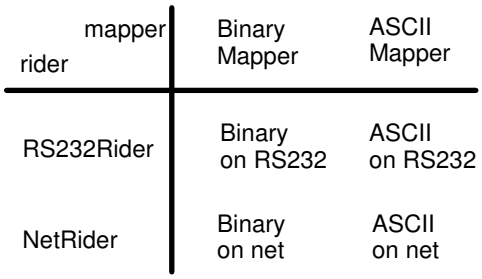

Рис. 3-14. Расширяемость в двух измерениях

В терминах проектных шаблонов, шаблон Носитель-Бегунок-Преобразователь (Carrier-Rider-Mapper) решает проблему гибкости и удобства доступа к носителю данных, однако он является исполняемым. Он может рассматриваться как состоящий из двух простых и наиболее важных шаблонов (схем, приемов, методов) проектирования: разделение одного объекта на несколько объектов, чтобы обеспечить отношение многие-к-одному; и разбиение одного объекта на два объекта для обеспечения независимой расширяемости:

Разбейте абстракцию на два интерфейса, если некоторые клиенты в отдельных случаях должны входить одновременно, и если независимое состояние должно осуществляться для каждого клиента.

Разбейте абстракцию на два интерфейса, если может потребоваться ее расширить независимо в двух различных измерениях.

Разделение Носитель-Бегунок-Преобразователь возвращает к исследовательскому проекту ["Insight ETHOS: On Object-Orientation in Operating Systems"; Clemens Szyperski; vdf, Zьrich, 1992, ISBN 3 7281 1948 2], предшествовавшему Блэкбоксу. Этот проект использовал некоторые шаблоны проектирования и правила проектирования (например, исключение наследования реализации), которые также описаны в *Design Patterns*. В терминологии *Design Patterns* комбинация Бегунок-Преобразователь (или Носитель-Преобразователь, если нет бегунка) формирует шаблон *мост* (*bridge*).

В Блэкбоксе бегунки часто создаются и управляются особым образом. Бегунок обычно находится под исключительным контролем одного клиентского объекта, Поскольку, кроме всего прочего, причиной, почему могут быть множественные бегунки на одного носителя (и, таким образом, почему два различных), является именно желание позволить нескольким клиентам иметь доступ к данным носителя через его частные пути доступа. Поскольку бегунки создаются в таких управляемых окружениях, Блэкбокс обычно создает бегунки следующим путем:

```
rider := carrier.NewRider(rider);
```

Идея в том, что если уже существует старый бегунок, который больше не используется, он может быть переработан. Переработка делается фабричным методом *NewRider* (см.следующий раздел), если возможно. Конечно, переработка законна только тогда, когда старый бегунок больше нигде не используется для чего-либо еще (возможно кем-то еще); вот почему так важно, что бегунок поддерживается в контролируемом окружении.

Обычно процедура *NewRider* применяется следующим образом:

```
PROCEDURE (o: Obj) NewRider (old: Rider): Rider;
   VAR r: RiderImplementation;
BEGIN
   IF (old # NIL) & (old IS RiderImplementation) THEN  (* использовать старый бегунок *)
      r := old(RiderImplementation)
   ELSE  (* разместить новый бегунок *)
      NEW(r)
   END;
   ... initialize r ...
   RETURN r
END NewRider;
```

Листинг 3-15. Повторное использование бегунка

Этот подход используется в Блэкбоксе для файлов, текстов и форм. Он может давать существенную разницу в эффективности в зависимости от типа приложения. Однако там, где эффективность несущественна, хорошее идеей будет избегать этот механизм, чтобы исключить любую возможность небрежного использования бегунков в разных контекстах одновременно.

3.4 Создание объекта

Компонентный Каркас Блэкбокса в основном спроектирован как черный ящик. Он строго отделяет интерфейс от реализации. Клиент может только импортировать и, таким образом, напрямую использовать или манипулировать интерфейсами объектов. Исполнения спрятаны внутри модулей. Это вынуждает модульных клиентов подчиняться следующему правилу проектирования (Gamma et.al.):

Нужно программировать, ориентируясь на интерфейс, а не на реализацию.

В Компонентном Паскале объектные интерфейсы представлены типами абстрактных записей. Реализацией записи абстрактного типа является ее конкретное расширение. Обычно реализации не экспортируются и поэтому не могут быть расширены ("subclassed") в других модулях, что означает невозможность наследования реализации. Это также позволяет сформулировать правило для программистов, занимающихся расширениями (Gamma et.al.):

Следует отдавать предпочтение композиции объектов перед наследованием классов.

Конкретные типы записей даже не могут быть прямо подтверждены: поскольку клиент не может импортировать объектное использование, чей тип не экспортирован, он не может разместить пример вызовом *NEW.* Это желательно, поскольку *NEW* будет привязывать клиентский код навсегда к одной частной реализации объекта; повторное использование кода для других реализаций станет невозможным где-либо еще. Но как код клиента может решить проблему размещения черного ящика, т.е. получить реализацию объекта без специфических знаний о нем?

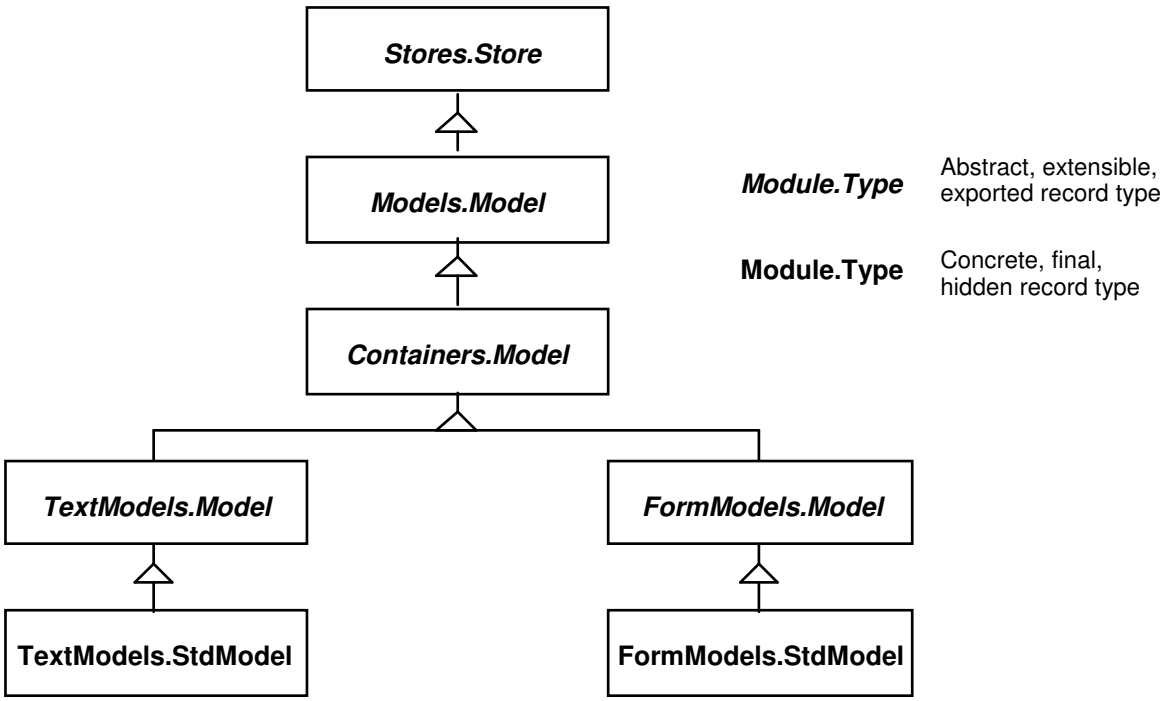

Рис. 3-16. Экспортированные интерфейсные записи и скрытые записи реализации

Вместо вызова *NEW* может быть возможен вызов функции размещения, так называемой фабричной функции. Например, модуль *TextModels* может экспортировать функцию *New*, которая размещает и возвращает текстовую модель. Внутренне она выполняет *NEW* над неэкспортируемым типом реализации и, возможно, выполняет некоторую инициализацию. Это лучше, чем прямой вызов клиентами *NEW*, поскольку модуль размещения может гарантировать корректную инициализацию и новая реализация (версия) модуля может выполнять размещение или инициализацию несколько иным путем, не влияющим на клиента. Однако, фабричные статичные функции являются слишком негибкими. Надлежащее решение проблемы размещения черного ящика требует уровня окольных путей. Есть несколько подходов, чтобы этого достигнуть. Они названы созидательными шаблонами. Блэкбокс использует 4 созидательных шаблона: прототипы, фабричные методы, каталожные объекты и фабричные управляющие. Они объясняются ниже один за другим.

Иногда требуется создать объект того же типа, что и некоторый уже существующий объект. В этом случае существующий объект называется *прототипом*, который дает фабричную функцию или метод инициализации. Модели Блэкбокса могут действовать как прототипы. Это дает уверенность, что вновь созданный объект имеет точно тот же конкретный тип, что и прототип, и позволяет прототипу и его клону(ам) совместно использовать буферы при необходимости. Например, текстовая модель Блэкбокса и ее клоны разделяют один вспомогательный файл, который они используют для размещения временных данных. Разделение предотвращает размножение открытых файлов.

Иногда существует объект, который может использоваться для создания объекта другого типа. Для этой цели существующий объект предлагает фабричную функцию, *фабричный метод*. Возможно, может предлагать разные фабричные методы, которые поддерживают разные стратегии инициализации или которые создают разные типы конкретных объектов.

В Блэкбоксе текстовая модель предлагает процедуры *NewReader* и *NewWriter*, которые создают новых анализаторов и синтезаторов. Они являются объектами доступа, которые представляют пути доступа к тексту.

Фабричные функции предназначены для объектов, которые исполняются одновременно ("*implementation covariance*"). Это всегда случай для бегунков и их носителей.

Но этот подход недостаточен для размещения самих текстовых моделей. Когда появляется

первая текстовая модель? Блэкбокс использует специальные объекты с фабричными методами, так называемые *фабричные объекты (factory objects)*. В Блэкбоксе фабричные объекты используются особым образом: они установлены в глобальные переменные и могут быть заменены во время исполнения без влияния на код клиента. По историческим причинам мы называем фабричные объекты, которые используются для конфигурационных целей, *директивными объектами (directory objects).*

Например, модуль *TextModels* экспортирует тип *Directory,* который содержит функцию *New.* Более того, он экспортирует две переменные *dir* и *stdDir* и процедуру *SetDir.* По умолчанию *TextModels.dir* используется кодом клиента для размещения новых пустых текстовых моделей. *TextModels.stdDir* содержит реализацию по умолчанию текстовой модели, что часто полезно при отладке реализации новой текстовой модели. Например, вы можете использовать старую папку при разработке реализации новой текстовой модели. Когда новая реализация готово, вы устанавливаете его объект-каталог вызовом *SetDir*, таким образом обновляя на лету текстовую подсистему под новую реализацию. Тексты со старой реализацией этим не затрагиваются. Если новая реализация содержит ошибки, реализация по умолчанию может быть переустановлена вызовом *TextModels.SetDir(TextModels.stdDir)*. Текстовая подсистема использует объекты типа каталог их обычным способом. Тот же шаблон проектирования может быть найден в других контейнерных модельных абстракциях; например, для моделей форм. В Блэкбоксе они используются для гораздо более простых моделей и образов; например, для *DevMarkers.*

Отметим, что объекты, размещенные через папки, часто являются устойчивыми и их реализации поэтому долгоживущими. По этой причине некоторые различные реализации появляются через время, которое должно использоваться одновременно. Обычно по умолчанию нормальные объекты папок подсистемы установлены автоматически, когда подсистема загружена. Если другие объекты папок должны быть установлены, используется модуль *Config* для установки конфигурации желаемых объектов папок при старте.

Имя "объект-каталог" ("directory object") пришло из другого использования шаблона проектирования каталога: в модуле *Files* файл может быть создан использованием объекта-каталога *Files.dir.* Но *Files.Directory* также предлагает средства для нахождения большей информации о текущей конфигурации файловой системы (например, процедур *Files.Directory.FileList*). Файловые каталоги также интересны, поскольку они показывают, что заменяемый каталог может иногда может нуждаться в продвижении к замененному объекту-каталогу (?). Например, если новый каталог обеспечивает специальные файлы памяти, которые отличаются именами, начинающимися на "M:\", тогда новый каталог должен контролировать каждый файл, проверяя, начинается ли его имя шаблоном, приведенным выше. Если так, то это файл памяти. Если нет, это нормальный файл и должен поддерживаться старым объектом-каталогом.

Может существовать одновременно несколько разных объектов-каталогов. Например, если сервис использует файлы особым образом, он может обеспечивать его собственный объект - файловую папку, которая по умолчанию пересылается стандартному объекту-каталогу файловой системы.

Множественные объекты-каталоги и связывающие объекты-каталоги являются частью шаблона проектирования каталога. Эта связующая часть шаблона является каскадируемой, если новый объект выходит на самый последний старый вместо *stdDir.*

Отметим, что, в отличие от типичных фабричных классов, каталоги редко расширяют.

client code
performs allocation
via *dir* variable

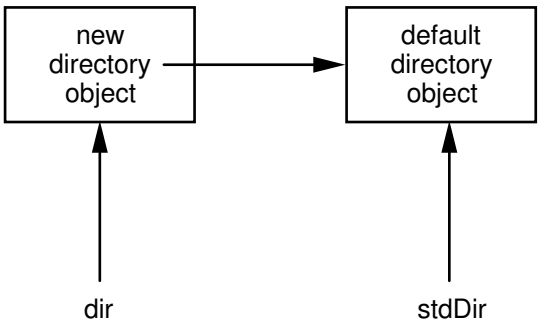

Рис. 3-17. Переход между двумя объектами-каталогами

Объекты-каталоги - очень подходящее решение, когда может ожидаться существование одного доминирующего реализации расширяемого типа черного ящика, без исключения существования других реализаций.

С другой стороны, также могут быть некоторые специальные объекты-каталоги для той же реализации, но для других целей. Например, может быть специальный объект текстовый каталог для текстов, которые имеют специальную линейку, полезную для программных текстов. Редактор программ предпочтет создавать новые программные тексты через этот каталог, а не через стандартный каталог текстовой подсистемы.

Registry (реестр) будет хорошим дополнением к объектам-каталогам. Реестр - это сохраняемая центральная база данных, в которой хранится информация о конфигурации, такую, как индивидуальные объекты-каталоги для установки при загрузке. Проблема с реестрами состоит в том, что они противоречат децентрализованной природе программных компонентов, что ведет к проблемам управления, таким как поиск реестровых входов, которые больше не адекватны.

Мы видели, что размещение через один обходной уровень есть ключ к проблеме размещения черного ящика. Объекты-каталоги являются решением, включающим глобальное состояние. Это желательно, только если это состояние меняется редко и не происходит параллельно иной активности (пересечений). Например, в серверных окружениях решение без учета состояния часто предпочтительнее. Это может быть достигнуто параметризацией клиента: клиент должен принять фабричный объект как параметр. Это делает возможным определить точную природу реализации на "вершине" иерархии процедурных вызовов, всегда пропуская фабричный объект "вниз". Это полезно, поскольку верхний уровень обычно гораздо меньше повторно используется, чем низкоуровневый код.

Как пример, диалоговая форма передачи файла может позволять задавать протокол передачи файлов вместе с такой информацией, как обратный адрес, скорость передачи и т.д. Протокол передачи файлов, который указывает реализацию коммуникационного объекта, может быть передан из диалоговой формы коммуникационному ПО, где он используется для размещения подходящего коммуникационного объекта (например, потоковый объект TCP).

Вместо передачи фабричного объекта может быть передано его символическое имя. "Внизу" сервис интерпретирует это имя; если необходимо, загружает модуль, который исполняет соответствующую фабричную функцию; и затем создает объект, используя фабричную функцию. Таким образом, сервис действует как *фабричный менеджер*.

Этот подход используется для некоторых сервисов Блэкбокса, в частности, для подсистем

Sql и *Comm*. В обоих случаях имя модуля может передаваться, когда размещается новый объект. Этот модуль, как ожидается, предоставит подходящие фабричные функции. Для *Sql* модули *SqlOdbc* и *DtfDriver* поставляют подходящие реализации объекта и фабричные функции. Для *Comm* модуль *CommTCP* поставляет подходящую реализацию объекта.

Фабричные менеджеры часто больше подходят для создания временных (non-persistent) объектов, тогда как объекты-каталоги больше подходят для создания постоянных (несохраняемых, persistent) объектов.

Часть II: Библиотека

В части I представлено введение в шаблоны проектирования, применяемые в Блэкбоксе. Знание этих шаблонов облегчает понимание и запоминание более детальных проектных решений в различных модулях Блэкбокса.

Часть II поясняет применение наиболее важных компонентов библиотек: элементов управления, форм и текстовых элементов.

В части III серия постепенно усложняющихся примеров показывает разработку объектов изображения (вьюшек).

4 Формы

В первой главе этой части мы сконцентрируемся на структуре формоориентированных элементов управления. "Формоориентированный" означает наличие редактора с графическим макетированием, иногда называемого "визуальным дизайнером" или "экранным художником", который позволяет вставлять, перемещать, изменять размеры, и удалять элементы управления. Элементы управления можно сделать активными, соединив их с фрагментом кода. Например, можно определить действие, которое будет выполняться при щелчке на кнопке.

Элементы управления могут иметь разный статус видимости, например, могут быть доступными или недоступными. Это способ показать пользователю какие действия могут быть доступны. Выставление статуса элементов управления может привести к значительным изменениям в приложении, дружественном к пользователю. К сожалению, такие особенности интерфейса часто требуют больше усилий от программиста, чем собственно логика работы приложения. Тем не менее, необходимо понимать лишь небольшое количество концепций способов построения подобных пользовательских интерфейсов. Эти концепции будут разъяснены с применением простых примеров.

4.1 Введение

Мы хотим как можно скорее начать с конкретных примеров, но небольшие вводные замечания все же необходимы. Ожидается, что читатель уже знаком с основами программирования, желательно на одном их диалектов Паскаля. Более того, ожидается, что он знаком с платформой (Windows или Macintosh) и основными принципами пользовательского интерфейса платформы.

Система Блэкбокс используется как средство образцово показать различные свойства компонентного программного обеспечения. Содержание книги может быть применено как для Windows, так и для Macintosh-версии системы Блэкбокс, за исключением экранных снимков, сделанных в основном с применением Windows 95. Для минимизации платформенной специфики в тексте, применяются следующие соглашения:

* Mac OS folders называются "каталогами",
* В путях применяется литера "/" как разделитель каталогов как в Unix и World-Wide Web,
* Наименования файлов и каталогов могут содержать как прописные, так и строчные буквы,
* Наименования файлов документов даются без расширения ".odc", применяемого в Windows. Так, наименование файла "Text/Rsrc/Find" соответствует "Text\Rsrc\Find.odc" в Windows 95 и "Text:Rsrc:Find" в MacOS,
* Клавиша-модификатор: в Windows - Ctrl, в MacOS - Option,
* Команды меню: M->I - сокращение для пункта I из меню M, например File->New

При работе в оболочке BlackBox полезно помнить следующее: практически все изменения в документах BlackBox могут быть отменены, что делает безопасным эксперименты с

различными тонкостями. В общем, доступно многократная отмена, то есть не только одна, но и несколько команд могут быть отменены, столько сколько позволяет объем памяти.

Эта книга не заменяет руководство пользователя по BlackBox. В ней задействовано минимальное число специфических средств BlackBox, поэтому требуется описание этих средств. Текст предполагает концентрацию на программировании в ущерб описанию средств. Там, где это необходимо, разъяснения особенностей средств даются при первом упоминании.

Для читателей, не владеющих языком Компонентный Паскаль свободно, в приложении B описываются различия между Паскалем и Компонентным Паскалем.

Окно справки системы Блэкбокс дает прямой или косвенный доступ к полной и обширной интерактивной документации, такой как руководство пользователя, все примеры ("Overview by example") и прочее. Во всемирной паутине дополнительные источники могут быть найдены по адресу http://www.oberon.ch.

4.2 Пример "Телефонная книга"

На протяжении всей части книги мы рассмотрим изменения и дополнения характерного примера. Этот пример - чрезвычайно упрощенная база данных. Идея не в представлении полноценного приложения со всеми возможными "примочками", а в минимальном примере, который бы не прятал объясняемые концепции за большими кусками кода. Тем не менее, идеи как добавить "примочки" компонент за компонентом видна все время.

Наша база данных телефонов содержит следующие фиксированные значения:

Daffy Duck	310-555-1212
Wile E. Coyote	408-555-1212
Scroodge McDuck	206-555-1212
Huey Lewis	415-555-1212
Thomas Dewey	617-555-1212

Таблица 4-1. Состав базы данных телефонов

В базе данных может быть найден телефонный номер по имени. Напротив, в базе данных можно отыскать имя по телефонному номеру, или содержимое базы данных может быть доступно по индексу. Мы уже видели возможную реализацию такой базы данных в разделе 3.2.

Наша задача - создать интерфейс для базы данных. Пользовательский интерфейс представляет собой диалоговое окно, содержащее два поля; одно под имя, второе под телефонный номер. У каждого поля есть заголовок, поясняющий назначение. Более того, диалоговое окно содержит переключатель, который позволяет определить, по какому из полей: имени или телефонному номеру, осуществляется поиск, и, наконец, кнопку, которая вызывает команду поиска. Следующий экранный снимок показывает, как будет выглядеть диалоговое окно окончательно:

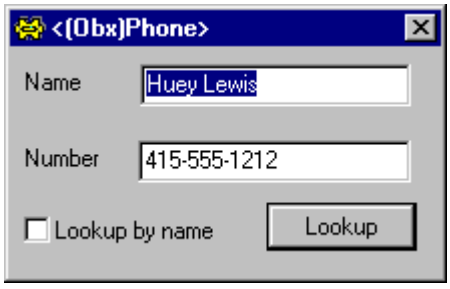

Рисунок 4-2. Образец окна поиска в базе данных

Для построения окна поиска начнем с создания пустой формы диалогового окна, используя команду Contorls->New Form. В результате будет вызвано диалоговое окно:

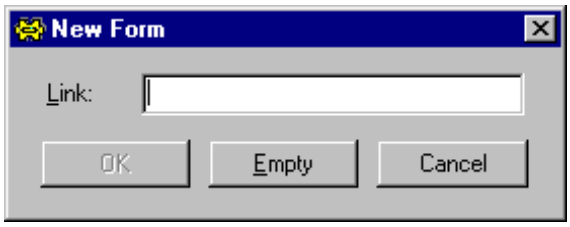

Рисунок 4-3. Диалог новой формы

Щелчок на кнопке "Empty" открывает пустую форму:

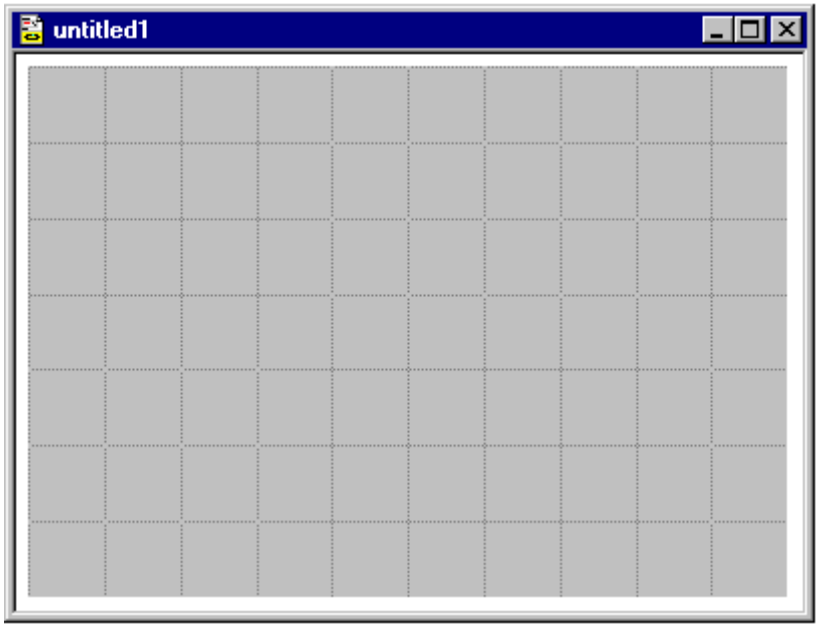

Рисунок 4-4. Пустая форма

Используя команды из меню Controls, мы вставляем те элементы управления, которые необходимы, а именно: два заголовка, два однострочных редактора, переключатель и кнопку. Вставленные элементы управления имеют обобщенные названия вроде "untitled" (безымянный) или "Caption" (заголовок). Для изменения этих визуальных свойств применяется инспектор свойств. Он открывается путем выбора элемента управления и затем: Edit->Object Properties... (Windows) или Edit->Part Info (Mac OS) соответственно. Отредактируйте поле "label" в на форме для изменения заголовка выбранного элемента управления и щелкните на кнопке по умолчанию для сохранения изменений.

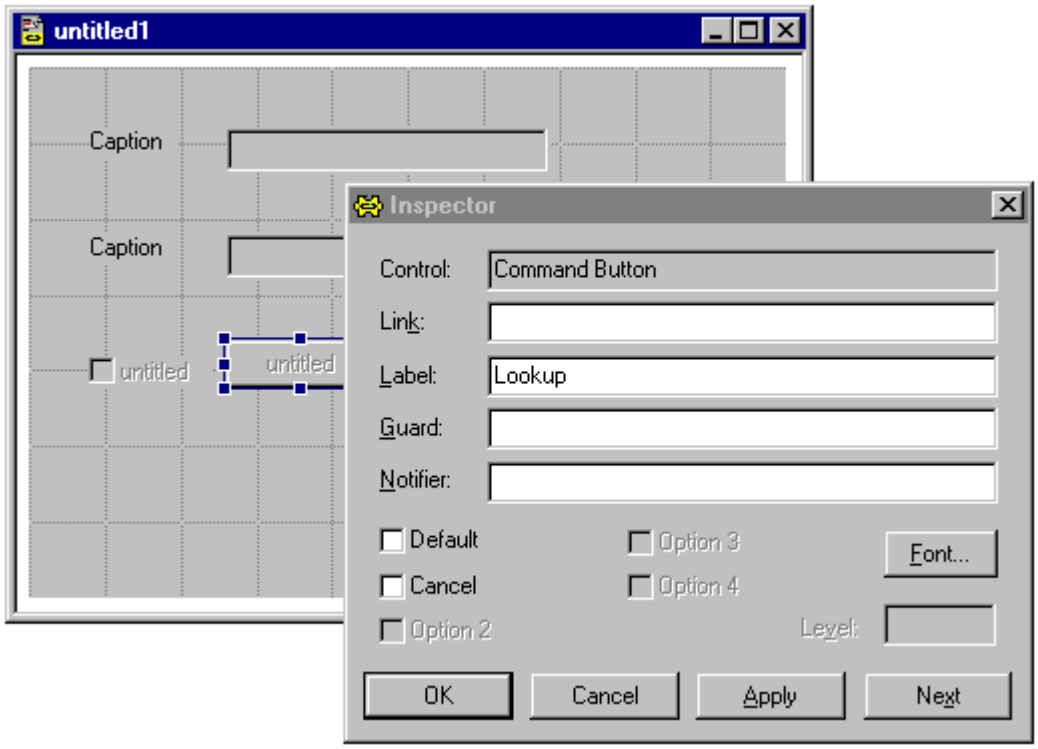

Рисунок 4-5. Редактор свойств компонента

Измените поле "label" для всех элементов управления так, чтобы в конце вы получили макет подобно рисунку 4-2. При переходе в порядке табуляции поля "label" будут следующими (слева-направо, сверху-вниз):

Тип элемента управления	Заголовок
Caption	Name
Text Field	
Caption	Number
Text Field	
Check Box	Lookup by Name
Command Button	Lookup

Таблица 4-6. Список элементов управления диалога телефонной книги

Элементы управления можно разместить по-другому, используя мышь или команды разметки из меню Layout. После изменения разметки макета, обязательно вызовите Layout->Sort Views; эта команда отсортирует элементы управления в том порядке, в котором они будут обходиться, перемещения по клавише табуляции между ними будут происходит в традиционном порядке, то есть слева-направо и сверху-вниз.

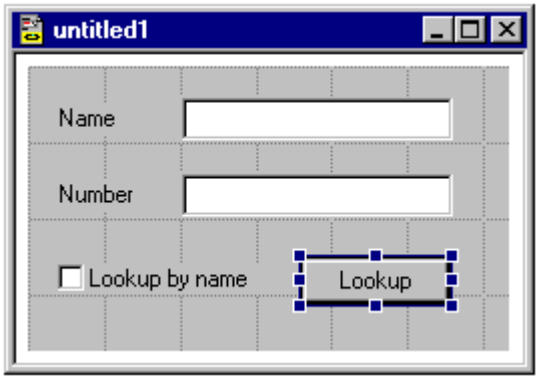

Рисунок 4-7. Окончательный макет диалога телефонной книги

Когда вы останетесь довольны макетом, следует сохранить диалоговое окно как и любой другой документа, используя команду File->Save. По соглашению, макеты диалоговых окон всех примеров сохранены в каталоге Obx/Rsrc. В нашем случае, мы сохраним диалоговое окно как Obx/Rsrc/PhoneUI.

Каталог Rsrc необходим для т. наз. ресурсов. Ресурсы — это документы, необходимые для работы программ. Например, макеты диалоговых окон и строковых ресурсов. Строковые ресурсы позволяют вынести строки за пределы программного кода в отдельный редактируемый документ.

Ресурсы могут быть отредактированы без перекомпиляции чего-либо. Например, вы можете изменить все заголовки в представленном выше диалоговом окне с английского на немецкий без необходимости доступа с источнику кода или инструментов разработки.

4.3 Интеракторы

Создание диалогового окна - это хорошо, но необходим еще способ добавить линию поведения для диалогового окна. В нашем случае пользовательский интерфейс диалогового окна должен обеспечивать операции поиска в базе данных телефонов. Чтобы это выполнить, нужно реализовать это для базы данных. Мы уже упоминали выше, что подходящая реализация уже существует. Не должно вызывать удивление, так как в данной части книги рассказывается о сборке компонентов, что имеется ввиду соединени имеющихся компонентов и подходящих экземпляров объектов, которые они реализуют.

В нашем примере, база данных телефонов представлена в модуле Component Pascal под именем ObxPhoneDB. Мы уже знакомы с этим модулем по главе 3. Сейчас мы хотим создать наш первый новый модуль, назначение которого состоит в построении моста между модулем базы данных ObxPhoneDB и диалоговым окном, созданным нами ранее. Новый модуль назовем ObxPhoneUI, где "UI" означает "Пользовательский Интерфейс". Это типичный модуль с программным кодом, основной целью которого является обеспечение работы составного документа, который может использоваться как представление логики приложения, такого, как в нашем случае, как простая база данных телефонов.

Новый модуль использует, то есть импортирует, два существующих модуля. Один - это ObxPhoneDB, другой - Dialog:

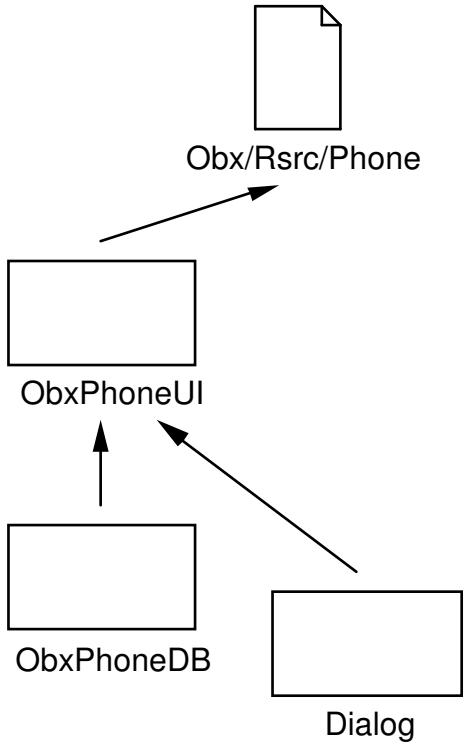

Рисунок 4-8. Важные связи между ObxPhoneUI и ObxPhoneDB

Dialog - часть основополагающего каркаса, имеющегося в BlackBox Component Builder. Этот модуль обеспечивает различные службы для поддержки пользовательского интерфейса. С наиболее важными мы уже встречались в этой главе.

```
MODULE ObxPhoneUI;

  IMPORT Dialog, ObxPhoneDB;

  VAR
    phone*: RECORD
      name*, number*: ObxPhoneDB.String;
      lookupByName*: BOOLEAN
    END;

  PROCEDURE Lookup*;
  BEGIN
    IF phone.lookupByName THEN
      ObxPhoneDB.LookupByName(phone.name, phone.number);
      IF phone.number = "" THEN phone.number := "not found" END
    ELSE
      ObxPhoneDB.LookupByNumber(phone.number, phone.name);
      IF phone.name = "" THEN phone.name := "not found" END
    END;
    Dialog.Update(phone)
  END Lookup;

END ObxPhoneUI.
```

Листинг 4-9. Первая версия ObxPhoneUI

ObxPhoneUI экспортирует глобальную запись phone, которая содержит два строковых и

одно логическое поля. В зависимости от текущего значения логического поля, процедура Lookup берет phone.name для поиска соответствующего номера или phone.number для поиска соответствующего имени. Если поиск неудачен, то есть возвращена пустая строка, то результат переключается на "не найдено". Оба результата помещаются в phone, то есть phone несет одновременно входной и выходной параметр поиска в базе данных.

Поскольку пользователь может интерактивно менять содержимое записи phone, манипулируя элементом управления, например, заполняя текстовые поля, глобальная переменная вроде phone называется интерактором. Элементы управления отображают содержимое полей интерактора и, если возможно, позволяют изменять их интерактивно. Для этого каждый элемент управления вначале должен быть присоединен к соответствующему полю интерактора. Это выполняется с помощью редактора свойств элементов управления, который мы видели ранее. Поле "Link" должна содержать имя поля, например ObxPhoneDB.phone.name или имя процедуры, например ObxPhoneDB.Lookup. Используйте инспектор свойств элемента управления для выставления связей согласно таблице 4-10:

Control type	Label	Link
Caption	Name	
Text Field		ObxPhoneUI.phone.name
Caption	Number	
Text Field		ObxPhoneUI.phone.number
Check Box	Lookup by Name	ObxPhoneUI.phone.lookupByName
Command Button	Lookup	ObxPhoneUI.Lookup

Таблица 4-10. Связи телефонной базы и диалогового окна.

Обратите внимание, что элемент управления бывший до этого недоступным сейчас стал доступным. Что это значит? Когда элемент управления связан, как это обычно случается при чтении формы из файла или, как в нашем случае, связь изменена из инспектора, модуль, связанный с этим элементом должен быть загружен. Если модуль уже загружен, ничего не требуется делать дополнительно. Если он еще не был загружен (это можно проверить по Info->Loaded Modules), загрузка происходит немедленно. Если загрузка оборвалась, например, код модуля еще не существует, связка элемента управления нарушается и он остается недоступным. Связывание так же оканчивается неудачей, когда элемент управления и поле не совместимы по типу, например кнопка независимой фиксации связана со строковым полем.

Для достижения такого уровня функциональности (и безопасности против неверного применения), BlackBox обеспечивает ряд улучшенных служб "метапрограммирования", а именно динамическая загрузка модулей по требованию и проверка типов переменных (typesafe lookup of variables). Последнее требование расширяет информацию о типе на этапе выполнения (RTTI), что сравнительно необычно для полностью компилирующих языков.

Связи всех элементов управления переопределяются каждый раз, когда модуль выгружается. Это гарантирует, что элементы управления никогда не ссылаются на незагруженные модули.

Одной из целей при проектировании системы Блэкбокс было обеспечить разделение деталей пользовательского интерфейса и программной логики. По этой причине модуль ObxPhoneUI ничего не знает об элементах управления, формах и т.п. Например, связка по кнопке "ObxPhoneUI.Lookup" говорит о необходимости активировать процедуру ObxPhoneUI.Lookup, когда кнопка будет нажата. В свою очередь, процедура ничего не знает о командной кнопке (или даже о нескольких), а только оповещает возможные элементы управления, вызывая

```
Dialog.Update(phone)
```

в конце процедуры. Dialog.Update инициирует обновление всех элементов управления, которые этого требуют. Например, если процедура Lookup записала значение "не найдено" в поле phone.number, то соответствующие текстовые поля и подобные элементы управления должны быть перерисованы соответствующим образом.

Как параметр для Dialog.Update должен быть передан интерактор (interactor). Вызов Dialog.Update необходим после программного изменения одного или нескольких полей интерактора. В случае модификации нескольких полей, Dialog.Update следует вызывать только одни раз из соображений эффективности. Необходимо отметить, что элементы управления вызывают Dialog.Update самостоятельно, когда пользователь изменяет поля интерактора, необходимость в принудительном вызове возникает только когда интерактор изменяется программным кодом.

Строгое разделение интерфейса пользователя и программной логики весьма необычно. Преимущество разделения - в простоте: как только мы узнаем как определять процедуры и как объявлять записи и глобальные переменные, вы уже можете создавать графический интерфейс пользователя для модуля. Это возможно даже для тех, кто только начал заниматься программированием. Другое преимущество: вам не требуется писать код для простого пользовательского интерфейса. Создание пользовательского интерфейса осуществляется посредством редактора форм (то есть установкой места и размера элементов управления) и инспектора (то есть, установкой выравнивания текста и собственно текста для строковых полей). Это позволяет легко изменять пользовательский интерфейс под различные требования без затрагивания логики приложения. Только если вы хотите получить больший контроль над пользовательским интерфейсом, например доступность элементов управления или реакция на особые события, такие как печать с клавиатуры, необходимо написать небольшой кусок кода, который может быть очень просто отделен от логики непосредственно приложения. Необходимые концепции, так же называемые охранники (guards) и уведомления (notifiers) будут обсуждаться в следующих двух разделах. Если вам все еще нужны другие элементы управления, можно изучить самостоятельно, как рассказано в разделе 4.9.

В настоящее время к недостаткам системы Блэкбокс можно отнести необходимость связывать элементы управления с глобальными переменными-интеракторами. Если есть несколько элементов управления для одного интерактора, все из них отображают одинаковое значение. Элементы управления не могут имеют своего независимого состояния.

Отметим интересную особенность Блэкбокса: если написан модуль, подобный ObxPhoneUI из листинга 4-9, то возможна автоматическая генерация формы с подходящими элементами управления по умолчанию. Это происходит при нажатии кнопки "Create" в диалоге "New Form" вместо кнопки "Empty". Эта особенность полезна при создании тестовых или отладочных интерфейсов в процессе разработки, когда нецелесообразны затраты времени на ручное построение форм.

Итак, мы имеем дизайн диалогового окна с элементами управления, связанными с модулем ObxPhoneUI и этот модуль импортирует механизм управления базой данных ObxPhoneDB. Сейчас все еще отсутствует способ применения диалогового окна вместо простого редактирования его дизайна. В процессе редактирования очень полезно сразу же попробовать диалоговое окно, даже если оно еще не совершенно. Для того, чтобы опробовать, убедитесь, что макет является самым верхним окном и затем выполните Controls->Open As Aux Dialog. Новое открывшееся окно содержит такой же диалог, но может быть отредактировано содержимое элементов управления вместо изменения разметки этих элементов. Окно ведет себя как маска ввода. Сейчас можно напечатать, например, "Huey Lewis" в качестве имени (Name), отметить флаг "Lookup by Name" и щелкнуть по кнопке "Lookup". Можно увидеть, что соответствующий номер появился в поле "Number".

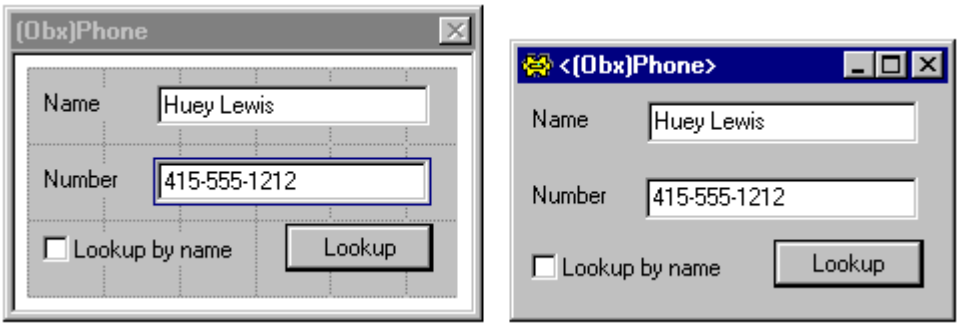

Рисунок 4-11. Дизайн (слева) и работа (справа) для одной и той же формы.

Следует отметить, что то же имя и номер так же отразились и в окне разметки. Более того, если вы измените разметку в дизайнере, например, подвинете кнопку независимой фиксации, можно увидеть, что изменения немедленно отразятся и в соседнем окне. Такое поведение так же называется Model-View-Controller, реализованной в BlackBox. В Части II книги этот шаблон проектирования уже детально обсуждался. Сейчас достаточно сказать, что несколько просмотров могут совместно использовать одни и те же данные, то есть дизайн и рабочий диалог могут отображать одну и ту же форму; и разметка, и диалог явления одного порядка, хотя и в разных формах. Переключение между двумя режимами осуществляется посредством команд: Dev->Layout Mode или Dev->Mask Mode.

Два режима работы различаются способом обработки выделения и активности. В режиме разметки можно выделить внедренный просмотр и отредактировать выделение, но нельзя сделать активным элемент просмотра. В режиме работы вы можете активизировать просмотр, но не можете выделить сам элемент управления (только его содержимое) и поэтому не можете изменить его разметку, то есть его размер, местоположение и т.п. Другими словами: режим разметки запрещает активацию, режим работы запрещает выделение.

Открытие второго окна формы в рабочем режиме удобно в процессе разработки, но не желательно, чтобы окно разметки открывалось в момент реальной работы программы. В таком случае следует открывать диалоговое окно в рабочем режиме применяя подходящие команды меню.

Новая команда меню может появиться после редактирования текста конфигурации. Можно открыть этот текст (расположенный в System/Rsrc/Menus) выполнив команду Info->Menus. Допишите следующий текст в конец файла:

```
MENU "Priv"
    "Open..."    ""    "StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phonebook')"    ""
END
```

Затем следует выполнить Info->Update Menus. Появится сообщение, что появилось новое меню "Priv". Выполните пункт меню "Open..." В результате будет выполнена команда

```
StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phonebook')
```

Она окрывает макет Obx/Rsrc/PhoneUI, переключает его в режим ввода и открывает его с заголовком "Phonebook". Если необходимо, чтобы изменения меню стали постоянными, сохраните текст меню перед его закрытием.

Отметим, что форма не сохранялась в режиме ввода (это было бы неудобно для последующего редактирования), она только временно переключается в режим ввода командой StdCmds.OpenAuxDialog.

4.4 Охранники

В рамках чистой функциональности мы узнали все необходимое о стандартных наборах

элементов управления. Позднее будет рассказано о палитре стандартных компонентов более подробно. Тем не менее вначале необходимо обсудить важные моменты стандартных элементов управления, которые, строго говоря, не добавляют функциональности приложению, а необходимы только для удобства пользователя, то есть для дружественности к пользователю.

В разделе 1.2 уже было обсуждено что значит дружественный к пользователю. Например, это означает уход от модальности везде, где это возможно. По этой причине BlackBox не поддерживает модальных диалоговых окон. Подобный способ действия необходим, если действия пользователя в одних случаях возможны, а в других случаях - нет. Например, если буфер обмена пуст, его содержимое не может быть вставлено в текст. Если он не пустой и содержит текст, то вставка возможна. С этим ничего не поделаешь, поэтому текущий статус должен быть виден или легко получен пользователем. Например, если буфер обмена пуст, команда меню Paste должна быть визуально отмечена недоступной. Видимые различия дают пользователю раннее предупреждение, что данная команда не доступна. Это гораздо лучше, чем позволить пользователю выполнить команду, а затем получить сообщение об ошибке. Следующий пример показывает диалоговое окно с двумя кнопками. Кнопка Empty доступна, только когда кнопка Create недоступна. Кнопка Create становится доступной только в случае, если что-то было напечатано в текстовом поле диалогового окна:

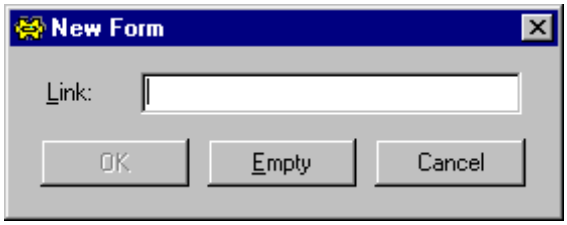

Рисунок 4-12. Доступные и недоступные кнопки.

Итак, хороший пользовательский интерфейс всегда позволяет пользователю выполнить любое осмысленное действие и дает визуальные подсказки о действиях, не имеющих смысла.

Таким образом, Блэкбокс должен предоставить способ обеспечить обратную связь в системе, особенно для текущей доступности или недоступности команд.

В этих целях должно быть возможным разрешать или запрещать элементы управления и пункты меню. Например, поиск телефонного номера возможен только в случае, если введено имя, то есть если текстовое поле имени не пусто. Для определения когда процедура команды, в нашем случае - процедура Lookup, может быть вызвана, то есть когда соответствующий элемент управления или пункт меню может быть доступен, должен быть определен подходящий охранник. Охранник - это процедура, вызываемая из среды каждый раз, когда требуется изменить статус элемента управления или пункта меню. Охранник проверяет состояния нескольких глобальных параметров системы, применяет эти состояния для определения доступности подопечных охраннику команд и выставляет соответствующим образом выходной параметр. Охранник имеет следующий вид:

```
PROCEDURE XyzGuard* (VAR par: Dialog.Par);
BEGIN
   par.disabled := ...some Boolean expression...
END XyzGuard;
```

Охранник описывается следующим типом:

```
   GuardProc = PROCEDURE (VAR par: Dialog.Par);
```

Для процедуры охранника Lookup из нашего примера мы дополним модуль ObxPhoneUI следующим образом:

```
MODULE ObxPhoneUI;

  IMPORT Dialog, ObxPhoneDB;

  VAR
    phone*: RECORD
      name*, number*: ObxPhoneDB.String;
      lookupByName*: BOOLEAN
    END;

  PROCEDURE Lookup*;
  BEGIN
    IF phone.lookupByName THEN
      ObxPhoneDB.LookupByName(phone.name, phone.number);
      IF phone.number = "" THEN phone.number := "not found" END
    ELSE
      ObxPhoneDB.LookupByNumber(phone.number, phone.name);
      IF phone.name = "" THEN phone.name := "not found" END
    END;
    Dialog.Update(phone)
  END Lookup;

  PROCEDURE LookupGuard* (VAR par: Dialog.Par);
  BEGIN  (* disable if input string is empty *)
    par.disabled := phone.lookupByName & (phone.name = "") OR
                    ~phone.lookupByName & (phone.number = "")
  END LookupGuard;

END ObxPhoneUI.
```

Листинг 4-13. ObxPhoneUI с LookupGuard

Что случится после компиляции приведенного выше модуля? Его символьный файл на диске заменится новой версией, так как интерфейс модуля поменялся по сравнению с предыдущей версией. Так как изменения заключаются в простом добавлении глобальной процедуры (то есть новая версия совместима со старой версией), возможные зависимые модули, импортирующие модуль ObxPhoneUI не теряют актуальности и не требуют перекомпиляции.

Компиляция так же породит новый файл кода на диске. Тем не менее, старая версия ObxPhoneUI по-прежнему остается загруженной в памяти! Другими словами: однажды будучи загруженным, модуль остается в памяти ("выполнить и оставаться в памяти"). Это не проблема, так как модули чрезвычайно малы и занимают мало памяти. Тем не менее программист конечно же должен иметь возможность выгрузит модули без необходимости выхода из оболочки BlackBox Component Builder для того, чтобы опробовать новый версии модулей. Для этой цели служит команда Dev->Unload, выполняющая выгрузку модуля, код которого в настоящее время активен.

Необходимо отметить, что компиляция не выгружает модули автоматически, что часто бывает неудобно. В частности, если ведется работа с несколькими взаимосвязанными модулями одновременно, выгрузка одного из них раньше перед тем как другие корректно обновятся может привести к тому, что весь набор модулей придет в противоречивое состояние.

Для таких случаев, когда важна немедленная выгрузка модулей после компиляции, как в простом обсуждаемом примере, существует команда Dev->Compile And Unload.

Ее можно использовать для проверки новой версии модуля ObxPhoneUI. Отметим, что кнопка Lookup недоступна, когда поле Name пусто (если выбран режим "Поиск по имени") или когда поле Number пусто (если выбран режим "Поиск по номеру"). Ввод каких-либо литер в поле делает кнопку доступной, удаление всех литер — недоступной. В конце этого раздела более подробно разъясняется, когда именно срабатывает охранник, сейчас достаточно знать, что охранник срабатывает после ввода каждой литеры в элемент управления.

Охранники чаще всего применяются для обеспечения доступности/недоступности элементов интерфейса, таких как элементы управления или пункты меню. Тем не менее, иногда охранники выступают в более общей роли. Например, элемент управления может быть не только недоступен, но еще и быть только для чтения или не определен.

"Только для чтения" означает, что элемент управления не может быть модифицирован. Например, следующий охранник выставляет "Только для чтения" для полей выходного параметра. Первый охранник выставляет "только для чтения", если поиск идет не по имени, то есть по номеру. Очевидно, в этом случае номер - входной, а имя - выходной параметры. Параметр для чистого вывода должен быть только для чтения. Таким образом, первый охранник может использоваться как охранник для поля Name. Другой охранник может быть применен для поля Number, так как выставляет только для чтения, если поиск возвращает номер.

```
PROCEDURE NameGuard* (VAR par: Dialog.Par);
BEGIN   (* make read-only if lookup is by number *)
   par.readOnly := ~phone.lookupByName
END NameGuard;

PROCEDURE NumberGuard* (VAR par: Dialog.Par);
BEGIN   (* make read-only if lookup is by name *)
   par.readOnly := phone.lookupByName
END NumberGuard;
```

Листинг 4-14. Охранники "Только для чтения" для ObxPhoneUI

Поля интерактора, экспортированные только для чтения, то есть с символом "-" вместо "*", всегда имеют статус "только для чтения" вне зависимости от действий охранника (в противном случае нарушилась бы безопасность модуля, так как элементы, экспортируемые "только для чтения" могут быть изменены только в определяемом модуле - это инвариант).

Статус "*не определено*" для элемента управления означает, что элемент не имеет значения вовсе. Это возможно, если элемент управления отражает статус разнородного выделения. Например, кнопка независимой фиксации может отображать состоит ли выделенный текст из всех прописных или строчных букв. Если часть выделения состоит из прописных букв, а часть - из строчных букв, то элемент управления не имеет определенного значения. Тем не менее, после щелчка на кнопке независимой фиксации, в выделении все буквы преобразовываются в прописные и вновь получает определенное значение. Неопределенное значение может быть установлено из охранника следующим выражением:

```
par.undef := TRUE
```

Неопределенное значение может быть приравнено к значению "только для записи", так как значение элемента управления не может быть прочитано пользователем, так как оно еще не определено, но может быть изменено, тем самым выставляется какое-то значение.

Из полей disabled, readOnly и undef, только одно может быть установлено охранником в TRUE. Это приводит к тому, что возможно четыре состояния элемента управления: одно из

указанных выше полей или ни одного имеет установленный статус. Когда охранник вызывается оболочкой, все три логических поля устанавливаются в FALSE. Это одно из общих правил в BlackBox: логические поля по умолчанию выставляются в FALSE.

В таблице 4-15 представлены подходящие охранники для макета с рисунка 4-2:

Control type	*Link*	*Guard*
Caption		
Text Field	ObxPhoneUI.phone.name	ObxPhoneUI.NameGuard
Caption		
Text Field	ObxPhoneUI.phone.number	ObxPhoneUI.NumberGuard
Check Box	ObxPhoneUI.phone.lookupByName	
Command Button	ObxPhoneUI.Lookup	ObxPhoneUI.LookupGuard

Таблица 4-15. Охранники диалогового окна базы данных телефонов

Охранник может применятся к процедуре, к полю интерактора или к нескольким процедурам или полям интерактора. Если он применяется в большей части к одной процедуре или к полю, то имя охранника формируется добавлением слова "Guard" к (с большой буквы) имени процедуры или поля. Например, охранник для процедуры Lookup называется LookupGuard, охранник для поля phone.name называется NameGuard и т.п. Такое простое соглашение о наименование делает более легким определение связи между охранником и подопечным элементом. Соглашение об именовании "мягкое", то есть не навязывается оболочкой, фактически оболочка не знает об этом вовсе.

Экспорт охранников — не просто соглашение, а необходимость. Доступ к охранникам осуществляется так называемым механизмом метапрограммирования Блэкбокса, который по соображениям безопасности оперирует только набором экспортируемых элементов, то есть интерфейсов, доступных извне. Если охранник не экспортируется, элемент управления не может его вызвать. Подобная ситуация с полями интерактора, которые так же должны быть экспортированы, если интерактор будет связан с ними.

Если заглянуть в описание типа Dialog.Par (применяя Info->Interface...), то можно увидеть что еще не описаны два поля: label и checked:

```
Par = RECORD
  disabled, checked, undef, readOnly: BOOLEAN;
  label: Dialog.String
END
```

Поле label позваляет изменить заголовок элемента управления. Например, вместо заголовка кнопки "Toggle" можно применить заголовки "Switch On" и "Switch Off" в зависимости от текущего состояния системы. Этого можно добиться подобной процедурой:

```
PROCEDURE ToggleGuard* (VAR par: Dialog.Par);
  BEGIN
    IF someInteractor.isOn THEN
      par.label := "Switch Off"
    ELSE
      par.label := "Switch On"
    END
  END ToggleGuard;
```

Листинг 4-16. Пример охранника заголовка

Следует отметить, что охранник перекрывает все заголовки, которые были установлены для элемента управления из инспектора. Это верно для всех элементов управления (и для

всех пунктов меню также, смотри ниже).

Настоятельно не рекомендуется размещать строковые константы в коде программы, как в примере выше, так как при изменении языка, например, с английского на немецкий (см. так же раздел 1.4) потребуется обязательная перекомпиляция. В Блэкбоксе строки, относящиеся к пользовательскому интерфейсу, такие как заголовки или сообщения чаще всего заносятся в отдельный файл параметров, так же называемый ресурсом строк. Для каждой из подсистем может существовать один ресурсный файл, например Text/Rsrc/Strings или Form/Rsrc/Strings. Ресурс строк представляет из себя текстовый документ Блэкбокса, начинающийся с ключевого слова STRING и нескольких пар вида <ключ, строка>. Ключ представляет из себя литерную цепочку, отделенную символом табуляции от собственно строки, которую заменяет. Каждая пара <ключ, строка> должна заканчиваться символом возврата каретки. Пары не должны быть упорядочены каким-либо образом, но полезно отсортировать их в алфавитном порядке ключей, так как тогда будет удобней искать обыкновенный ключ в процессе редактирования строкового ресурса.

Например, System/Rsrc/Strings начинается так:

```
STRINGS
About    About BlackBox
AlienAttributes    alien attributes
AlienCause    alien cause
AlienComponent    alien component
AlienControllerWarning    alien controller (warning)
```

Таблица 4-17. Ресурс строк System/Rsrc/Strings

Для использования строковых ресурсов в нашем примере, необходимо использовать специальный синтаксис, который показывает, что строка в действительности является ключем, который должен быть заменен подходящим значением из строкового ресурса. Таким образом, в подсистеме Obx в строковом ресурсе имеются ключи "On" и "Off", следующий код их использует:

```
PROCEDURE ToggleGuard* (VAR par: Dialog.Par);
  BEGIN
    IF someInteractor.isOn THEN
      par.label := "#Obx:Off"
    ELSE
      par.label := "#Obx:On"
    END
  END ToggleGuard;
```

Листинг 4-18. Охранник заголовка с применением карты строк

Начальная литера "#" указывает, что нужно выполнить подстановку литерных цепочек. Затем следует название подсистемы, в нашем случае "Obx". Затем следует двоеточие, после которого идет ключ для замены. Кнопка с таким охранником будет отображать заголовки "Switch Off" или "Switch On" в английской версии Блэкбокса, "Ausschalten" или "Einschalten" в немецкой версии и т.п.

Если нет подходящего строкового ресурса, ключ заменяется на собственное значение. Например, если не будет ключа "Off" в Obx/Rsrc/Strings, то "#Obx:Off" будет заменено на "Off".

Из Dialog.Par осталось нерассмотренным поле checked. На самом деле оно никогда не применяется для элементов управления в Блэкбоксе. Оно используется для пунктов меню. Пункты меню подобны элементам управления: они могут запускать какие-то действия, могут быть доступными/недоступными, могут иметь заголовки. По этой причине к ним

применяется такой же механизм охранников. Охранник для меню определяется в тексте Menus как строка после метки пункта меню и клавиатурного эквивалента. Например, следующий фрагмент представляет определение охранника меню Dev:

```
"Edit Mode"   ""   "StdCmds.SetEditMode"   "StdCmds.SetEditModeGuard"
```

Отличительная черта элемента меню состоит в том, что он может быть отмечен. Например, в следующей меню пункт Edit Mode отмечен:

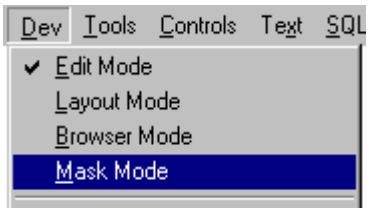

Рисунок 4-19. Пункт меню с отметкой

Отметка показывает, какой пункт был выбран самым последним, и какой статус был установлен по данному выделению. Состояние может быть изменено вовлечением других пунктов меню, например Mask Mode, как на рисунке выше. В общем, четыре пункта меню образут группу, из которых только один может быть выбран.

Процедура охранника для пункта меню может выставить значения disabled, checked и label для Dialog.Par. Другие поля для меню игнорируются.

Процедура-охранник может выставлять несколько полей для выходного параметра par одновременно, то есть может быть установлено поле disabled и label для кнопки. Тем не менее, охранник может выставлять как минимум одно логическое поле из par и никогда не должен изменять состояние вне рамок par, то есть поля интерактора или что-то подобное, то есть не должно быть побочных эффектов. Так же не должно быть вызовов других процедур, имеющих побочный эффект. Это сделано по причине того, что программа не может знать когда (а особенно когда "не") оболочка вызывает охранника. Охранник может использовать интерактор или несколько интеракторов как входные параметры, или текущее выделение или фокусировку. Последнее часто используется для охранников меню. Текущую фокусировку или выделение применяется в охранниках только для элементов управления так же называемых диалоговый панели инструментов, то есть диалоги, оперирующие содержимым нижеследующего документа. Диалоговое окно Find & Replace, работающее с текущим окном текста - типичный пример инструментального диалогового окна. Другие ока являются самодостаточными и называются вспомогательными диалоговыми окнами. Маска ввода данных - типичный пример вспомогательного диалогового окна.

Когда вызывается процедура-охранник? Существует четыре причины вызова охранника: когда установлена связь с элементом управления, когда изменяется содержимое интерактора, когда изменяется иерархия окна или когда пользователь щелкает на меню.

Связь элемента управления устанавливается после вставки в контейнер, после загрузки из файла, после выгрузки модуля или после того как связь была изменена в инспекторе свойств или другими средствами. Когда кусок кода изменяет содержимое интеркатора, необходимо вызвать Dialog.Update для этого интерактора. В результате каждый визуальный элемент управления получает уведомление (см. раздел 2.9). В ответ элементы управления сравнивают текущие состояния со значениями полей связанного интерактора. Если поля интерактора изменились, элемент управления соответствующим образом перерисовывается. После того, как все элементы управления обновили свое содержимое, вычисляются охранники всех визуальных элементов управления. По этой причине охранники должны быть оптимальными и не занимать много времени на обработку.

Охранники так же вычисляются, когда изменяется иерархия окон, то есть когда нижнее

окно перемещается наверх. Это необходимо, так как многие команды зависят от текущего фокуса и выделения, которые могут исчезнуть, когда другое окно перемещается наверх.

Другая причина вызова охранников - меню. Когда пользователь щелкает на меню, все охранники этого меню или охранник меню целиком пересчитывается.

Чаще всего охранник имеет форму:

```
PROCEDURE SomeGuard* (VAR par: Dialog.Par)
```

Другая форма охранника

```
PROCEDURE SomeGuard* (n: INTEGER; VAR par: Dialog.Par)
```

может применяться, когда требуется передать параметр для одного охранника нескольких связанных команд. Например, ниже приведены команды для выставления выделения в красный, зеленый или синий:

```
StdCmds.Color(00000FFH)
StdCmds.Color(000FF00H)
StdCmds.Color(0FF0000H)
```

Для этих команд могут быть применены следующие охранники

```
StdCmds.ColorGuard(00000FFH)
StdCmds.ColorGuard(000FF00H)
StdCmds.ColorGuard(0FF0000H)
```

Имеющаяся реализация заголовка StdCmds.ColorGuard:

```
PROCEDURE ColorGuard (color: INTEGER; VAR par: Dialog.Par)
```

4.5 Уведомления

Охранник элемента управления выставляет текущее состояние элемента, например: доступен или нет. Других действий не производится. Не изменяются какие-либо данные интерактора. Не добавляется какой-либо функциональности. Лишь добавляется обратная связь для элемента управления о текущей функциональности или ее возможности. Охранник пересчитывается только когда пользователь что-либо изменит в элементе управления. Такое положение может быть представлено как простая "косметическая" особенность, служащая исключительно для увеличения дружественности интерфейса.

Тем не менее, иногда действия пользователя должны инициировать больше, чем простое срабатывание охранника. В частности иногда необходимо изменить какое-либо состояние интерактора в ответ на действия пользователя. Например, изменение выделения в окне может повлечь необходимость обновления соответствующего счетчика, который показывает число выделенных элементов в окне. Или требуется реализация какой-либо последующей обработки пользовательского ввода, как будет показано на примере ниже. Это будет альтернативная реализация примера ObxPhoneUI. Вместо кнопки Lookup, новая версия имеет всего два текстовых поля. После каждого ввода литеры база данных выполняет поиск, пока не будет найден правильный ключ. Необходимо отметить, что неудачный поиск в ObxPhoneDB возвращает пустую строку.

```
MODULE ObxPhoneUI1;

  IMPORT Dialog, ObxPhoneDB;

  VAR
    phone*: RECORD
      name*, number*: ObxPhoneDB.String
    END;
```

```
  PROCEDURE NameNotifier* (op, from, to: INTEGER);
  BEGIN
    ObxPhoneDB.LookupByName(phone.name, phone.number);
    Dialog.Update(phone)
  END NameNotifier;

  PROCEDURE NumberNotifier* (op, from, to: INTEGER);
  BEGIN
    ObxPhoneDB.LookupByNumber(phone.number, phone.name);
    Dialog.Update(phone)
  END NumberNotifier;

END ObxPhoneUI1.
```

Листинг 4-20. Модуль ObxPhoneUI1 с уведомлениями.

Данное уведомление устанавливает зависимость между двумя полями интерактора: изменение одного поля может привести к изменению другого текстового поля, то есть определяется связь между двумя полями интерактора.

Уведомление вызывается сразу после взаимодействия, но до вычисления охранника. Уведомление имеет следующую форму:

```
  PROCEDURE XyzNotifier* (op, from, to: INTEGER);
  BEGIN
    ...
  END XyzNotifier;
```

Тип уведомления:

```
  NotifierProc = PROCEDURE (op, from, to: INTEGER);
```

Для простого уведомления параметры можно игнорировать. Параметры позволяют более точно определить какое именно изменение произошло. Какой из параметров определен и что именно он означает определено отдельно для каждого вида элементов управления. Для пояснения смотри следующий раздел.

4.6 Стандартные элементы управления

В данном разделе представлены различные элементы управления, принятые за стандарт в BlackBox Component Builder. Для каждого элемента управления приводится список возможных типов данных, с которыми можно связать элемент, и смысл и значения параметров уведомления.

Элементы управления могут иметь различные свойства, например заголовки или начертание заголовка стандартным системным шрифтом или более бледным, тонким. Тем не менее, подобные косметические свойства могут быть не реализованы на всех или части платформ, поэтому иногда просто игнорируются.

Заголовки могут иметь специальные метки (keyboard shortcut), которые позволяют на некоторых платформах перемещаться между элементами без применения мыши. Символ метки определяется по предшествующему знаку "&" (амперсанд). Две последовательных литеры "&" обозначают нормальный "&" без присвоения следующей литере признака метки. На один заголовок допустима только одна метка. Например: "Clear Text &Field" определяет "F" в качестве метки, в то время как "Find && &Replace" определяет "R" как клавиатурную метку. Подобное описание меток применяется и для команд меню.

Свойства заголовка для всех элементов управления могут быть изменены в процессе

выполнения посредством охранника, как было показано в листинге 4-16. Для элементов управления без видимого заголовка не будет видимого эффекта.

Два наиболее важных свойства элементов управления: охранник и уведомление - не обязательны. Если не определен охранник элемента, будет применяться значение по умолчанию (доступно, для чтения и записи, определено). Если не определено уведомление для элемента, ничего не будет вызываться.

В системе Блэкбокс сделана попытка минимизировать число различных свойств для облегчения их использования. Это согласуется с общими направлениями развития, как в Windows, так и в Mac OS, а так же в рамках общего развития элементов управления. Это означает, что для каждого отдельного элемента управления не следует специально назначать свойства цвета, шрифтов и т.п. Вместо этого пользователь должен назначить системные установки и конфигурацию этих свойств.

Тем не менее, если существуют весомые причины, можно назначит специальный шрифт для элементов управления простым выделением этого элемента и выполнением команд меню Font, Attributes или Characters. Если установлен шрифт по умолчанию, то будут выставлены общесистемные настройки. Это нормальное явление. Следует отметить, что различные атрибуты шрифтов, такие как стиль, жирность, размер, могут быть зафиксированы в некоторых платформах. Например, в Mac OS системный шрифт всегда 12 пунктов.

В нижеследующем тексте описаны стандартные элементы управления BlackBox в следующем порядке: кнопки (command buttons), редакторы (edit fields), кнопки независимой фиксации (check boxes), кнопки зависимой фиксации (radio buttons), поля ввода даты (date fields), поля ввода времени (time fields), поля ввода цвета (color fields), заголовки (captions), группы (groups), выпадающие списки (list boxes), списки (selection boxes), комбинированные списки (combo boxes). Для каждого элемента управления представлены его свойства, возможные типы переменных для связи, значения параметров op, from и to в уведомлениях.

Отметим, что не существует специального типа для ввода валюты, взамен следует использовать редактор, связанный с типом Dialog.Currency.

В приведенном описании применяются следующие сокращения:

pressed - Dialog.pressed
released - Dialog.released
changed - Dialog.changed
included - Dialog.included
excluded - Dialog.excluded
undefined означает, что параметр from или to уведомления не содержит определенного значения (в котором можно быть уверенным).
modifier означает 0 для простого щелчка и 1 для двойного щелчка
link, label, guard, notifier и level определены для соответствующего элемента управления в модуле Controls.

Может быть до 5 дополнительных логических свойств. В зависимости от элемента управления они могут называться по-разному, например: default font, default, cancel, sorted, left, right, multiLine, password.

Кнопка (Command Button)

Рисунок 4-21. Кнопка

Кнопка предназначена для выполнения непараметризированной команды Component Pascal

или целой последовательности команд. Кнопка по умолчанию отличается по внешнему виду от обычных кнопок; нажатие клавиши "Enter" соответствует щелчку мыши по кнопке по умолчанию. Для кнопки со свойством cancel, нажатие клавиши "ESC" соответствует щелчку мыши на данной кнопке. Кнопка не может быть одновременно "По умолчанию" и "Отмена". Следует отметить, что выставление свойств "По умолчанию" и "Отмена" выполняется для каждого элемента управления, то есть установка одной кнопки "по умолчанию" не превращает предыдущую кнопку со свойством "по умолчанию" в нормальную. Для диалогового окна следует выставлять не больше одной кнопки по умолчанию и не больше одной кнопки cancel. Кнопка отмены или любая другая кнопка, предназначенная для закрытия диалогового окна должна содержать команду StdCmds.CloseDialog.

properties: link, label, guard, notifier, font, default, cancel
linkable to: parameterless procedure, command sequence
op: pressed, released
from: undefined, modifier
to: undefined

Редактор (Text Field)

Рисунок 4-22. Редактор

Редактор отражает значение глобальной переменной, которая может иметь строковый, целый, дробный типы или различные типы Dialog.Currency или Dialog.Combo. Значение переменной может быть изменено в редакторе.

При любом изменении содержимого связанного поля интерактора вызывается уведомление. Нажатие клавиш, которые не изменяют состояние интерактора, таки как нажатие стрелок, ввод ведущих нулей для чиловых полей, не приводит к вызову уведомления. Изменение выделения содержимого поля так же не приводит к вызову уведомления.

Если изменение происходит для поля, связанного с типом строка, дробное число, валюта или переменной комбинированного списка, вызывается уведомление с параметрами (change, undefined, undefined). Если поле связано с целочисленным значением, вызывается уведомление с параметрами (change, oldvalue, newvalue).

Недопустимые литеры, например, буквы для полей связанных с числовыми типами, не вводятся.

Редактор может иметь заголовок, но он не показывается. Причина - метка клавиши может быть определена в заголовке, это может быть полезно для полей ввода.

Если значение свойства level 0 (по умолчанию) или элемент не связан с числовым полем или связан с типом Dialog.Currency, то свойство level не имеет эффекта. Если связь установлена с целочисленной переменной, level определяет масштаб для отображения числа, то есть связанное число отображается в разделенном на 10 в степени level. Например, если текущее значение 42 и level 2, то отображается значение 0,42. Для действительных чисел level показывает формат следующим образом:

level > 0: экспоненциальный (научный) формат, при этом не менее level цифр в мантиссе
level = 0: формат с фиксированной или плавающей точкой в зависимости от x
level < 0: фиксированный формат с level цифр после запятой.

Свойства left и right определяют режим выравнивания <выключки> содержимого поля. Возможны следующие комбинации:

left & ~right выравнивание по левому краю

left & right выравнивание по содержимому (может не срабатывать на некоторых платформах)
~left & ~right выравнивание по центру
~left & right выравнивание по правому краю

По умолчанию принято left & ~right (выравнивание по левому краю)

Свойство multiLine определяет возможность ввода символа возврата каретки и, соответственно, появление разрыва строки в указанное поле, которое должно быть связано со строковой переменной.

Свойство password определяет необходимость замены содержимого поля на литеры * при печати. Это свойство позволяет использовать строку для ввода паролей.

properties: link, label, guard, notifier, level, font, left, right, multiLine, password
linkable to: ARRAY const OF CHAR, BYTE, SHORTINT, INTEGER, LONGINT, SHORTREAL, REAL, Dialog.Currency, Dialog.Combo
op: pressed, released, changed
from: undefined, old value, modifier
to: undefined, new value

Кнопка независимой фиксации (Check Box)

Рисунок 4-23. Кнопка независимой фиксации

Кнопка независимой фиксации отображает значение глобальной переменной логического типа или типа элемента множества. Щелчок на элементе управления изменяет его статус.

Если элемент связан с логическим типом, то при изменении элемента вызываетя уведомление с параметрами (changed, undefined, undefined). Если связь установлена с элементом из множества, то вызываемые параметры будут (included, level, undefined) или (excluded, level, undefined) в зависимости от того, устанавливается или сбрасывается фиксация. Значение level соответствует элементу из множества, с которым элемент управления связан. Связь может быть определена из инспектора свойств элемента управления. Значение может находиться в пределах 0..31.

Только последнее и завершающее изменение статуса элемента управления приводит к вызову уведомления, возможные промежуточные изменения (перемещение мыши за пределы границ окна и обратно в область) не возымеют действия.

properties: link, label, guard, notifier, font, level
linkable to: BOOLEAN, SET
op: pressed, released, changed, included, excluded
from: undefined, level value, modifier
to: undefined

Кнопка с зависимой фиксацией (Radio Button)

Рисунок 4-24. Кнопка с зависимой фиксацией.

Кнопка зависимой фиксации становится активной при конкретном значении глобальной целочисленной или логической переменной. Обычно несколько кнопок с зависимой фиксацией связаны с одной переменной. Каждая кнопка установлена для разных значений, которые устанавливаются свойством level, то есть кнопка "включается", когда значение свойства level становится равно значению связанной переменной. Для логической переменной "установлено" соответствует значения TRUE, "сброшено" соответствует FALSE.

Только последнее и законченное изменение элемента управления приводит к вызову уведомления, возможные промежуточные состояния не имеют действия.

properties: link, label, guard, notifier, font, level
linkable to: BYTE, SHORTINT, INTEGER, LONGINT, BOOLEAN
op: pressed, released, changed
from: undefined, old value, modifier
to: undefined, new value = level value

Поле ввода даты (Date Field)

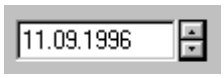

Рисунок 4-25. Поле ввода даты

Поле ввода даты отображает дату из глобальной переменной типа Dates.Date. При любом изменении поля связанного интерактора вызывается уведомление с параметрами (change, undefined, undefined). Нажатие клавиш, которые не изменяют значение интерактора, таких как стрелки влево, вправо, не приводят к вызову уведомления. Клавиши стрелок вверх, вниз изменяют дату. Изменение выделения не дает никакого эффекта. Неверная дата не может быть введена.

properties: link, label, guard, notifier, font
linkable to: Dates.Date
op: pressed, released, changed
from: undefined, modifier
to: undefined

Поле ввода времени (Time Field)

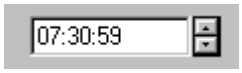

Рисунок 4-26. Поле ввода времени

Поле ввода времени отображает время из глобальной переменной типа Dates.Time. При любом изменении связанного интерактора вызывается уведомление с параметрами (change, undefined, undefined). Нажатия клавиш, которые не приводят к изменению значений интерактора, таких как стрелки влево, вправо, не не приводит в вызову уведомления. Стрелки вверх, вниз изменяют время. Изменения выделения в поле не оказывает влияния. Неверное значение времени не может быть введено.

properties: link, label, guard, notifier, font
linkable to: Dates.Time
op: pressed, released, changed
from: undefined, modifier

to: undefined

Поле выбора цвета (Color Field)

Figure 4-27. Поле выбора цвета

Поле выбора цвета отображает цвета. Оно может быть связано с переменной типа INTEGER или Dialog.Color. Тип Ports.Color является переопределением для INTEGER, поэтому так же может быть использован.

При выборе любого другого цвета вызывается уведомление с параметрами (change, oldval, newval). Старое и новое значение являются целыми числами, для переменной типа Dialog.Color имеется поле val.

properties: link, label, guard, notifier, font
linkable to: Dialog.Color, Ports.Color = INTEGER
op: pressed, released, changed
from: undefined, old value, modifier
to: undefined, new value

Промотка (Up/Down Field)

Рисунок 4-28. Промотка

Такое поле связано с целочисленной переменной. Значение так же может быть изменено с помощью клавиш со стрелками. При любом изменении связанного поля интерактора происходит вызов уведомления с параметрами (change, oldvalue, newvalue).

properties: link, label, guard, notifier, font
linkable to: BYTE, SHORTINT, INTEGER, LONGINT
op: pressed, released
from: undefined, old value, modifier
to: undefined, new value

Заголовок (Caption)

Рисунок 4-29. Заголовок

Заголовок обычно применяется в сочетании с редактором для описания его содержимого. Заголовок пассивен, то есть не может быть отредактирован, поэтому не имеет уведомления.

Заголовок может быть связан к тем же типам переменных, что и редактор, и может иметь охранника. Это полезно (в зависимости от платформы) при отображении заголовка с разными визуальными эффектами, если он (или в равной степени его соответствующий редактор) недоступен, только для чтения и т.п. Это означает, что заголовок может быть

недоступным вместе с соответствующим редактором, если их связать с одним и тем же полем интерактора. Охранник заголовка может изменить заголовок элемента управления, что актуально для элементов с видимым заголовком.

properties: link, label, guard, font, right, left
linkable to: ARRAY const OF CHAR,
BYTE, SHORTINT, INTEGER, LONGINT, SHORTREAL, REAL,
Dialog.Currency, Dialog.Combo

Свойства left и right определяют режим выравнивания заголовка. Возможны следующие комбинации:
left & ~right выравнивание по левому краю
left & right выравнивание по содержимому (может не срабатывать на некоторых платформах)
~left & ~right выравнивание по центру
~left & right выравнивание по правому краю

По умолчанию принято left & ~right (выравнивание по левому краю)

Группа (Group)

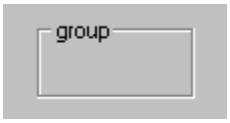

Рисунок 4-30. Группа

Группа применяется для визуального объединения связанных элементов, например: несколько кнопок зависимой фиксации. Группа - пассивный элемент, то есть не может быть отредактирована, а значит не имеет уведомления.

Группа может быть не связана с чем-либо, но может иметь охранника, делающего ее доступной или недоступной. Как и для всех элементов управления с видимым заголовком, охранник может менять менять заголовок группы.

properties: label, guard, font

Выпадающий список (List Box)

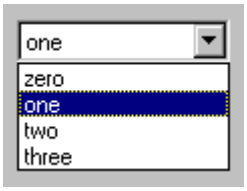

Рисунок 4-31. Выпадающий список (свернутая и распахнутая формы)

Выпадающий список позволяет выбрать одно значение из списка выбора. Он связан с переменной типа Dialog.List (смотри следующий раздел). Если высота списка не достаточна, отображается список с полосой прокрутки. Если и этого недостаточно, список сворачивается во всплывающее меню.

Выделение может быть изменено пользователем. Выделение может состоять из одного элемента списка или не содержать элементов. Когда пользователь изменяет выделение, вызывается уведомление с параметрами (changed, oldvalue, newvalue). Старое/новое

значение соответствует индексу выделенного элемента списка. Самый верхний элемент имеет индекс 0, следующий элемент - индекс 1 и т.п. Если ничего не выделено, индекс равен -1.

Свойство sorted определяет необходимость сортировки строк в алфавитном порядке (не действует в Mac OS).

properties: link, label, guard, notifier, font, sorted
linkable to: Dialog.List
op: pressed, released, changed
from: undefined, old value, modifier
to: undefined, new value

Список (Selection Box)

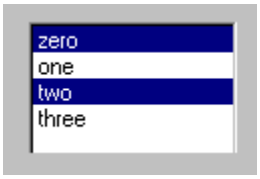

Рисунок 4-32. Список

Список позволяет выделить несколько элементов из предложенного набора выбора. Он связан с переменной типа Dialog.Selection (смотри следующий раздел).

Выделение может быть изменено пользователем. Каждый элемент может быть выделен отдельно. Когда пользователь изменяет выделение, вызывается уведомление следующими тремя возможными способами:

(included, from, to): диапазон from..to сечас выделен; не то, что выделено до этого
(excluded, from, to): диапазон from..to сейчас не выделено; было выделено до того
(set, from, to): диапазон from..to сейчас выделено; до этого не были выделены.

Три кода included, excluded, set выставляются на месте changed и применяются в большинстве других элементов управления. Уведомление вызывается по мере необходимости при включении и/или исключении всех необходимых элементов в диапазоны.

Значения from, to соответствуют индексам элементов. Первый элемент соответствует значению 0, следующий 1 и т.п.

Свойство sorted определяет необходимость сортировки строк в алфавитном порядке (не действует в Mac OS).

properties: link, label, guard, notifier, font, sorted
linkable to: Dialog.Selection
op: pressed, released, included, excluded, set
from: undefined, lowest element of range, modifier
to: undefined, highest element of range

Комбобокс (Combo Box)

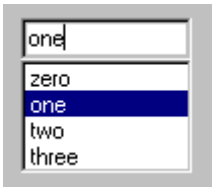

Рисунок 4-33. Комбобокс

Комбобокс - это текстовый редактор, содержащий еще и множество значений в стиле всплывающего меню. В отличие от настоящих всплывающих меню и списков, могут быть введены и значения не содержащиеся в списке всплывающего меню. Элемент управления связывается с переменной типа Dialog.Combo (смотри следующий раздел).

При изменении содержимого комбобокса вызывается уведомитель с параметрами (changed, undefined, undefined).

Свойство sorted определяет необходимость сортировки строк в алфавитном порядке (не действует в Mac OS).

properties: link, label, guard, notifier, font, sorted
linkable to: Dialog.Combo
op: pressed, released, changed
from: undefined, modifier
to: undefined

Каждый интерактивный элемент управления (за исключением заголовка или группы) вызывает уведомление (если оно есть) когда пользователь в первый раз щелкнет на элементе управления с параметром op = Dialog.pressed и затем с параметром op = Dialog.released, когда кнопка мыши освобождается. Эта особенность используется по большей части для отображения какой-либо строки в статусной строке диалогового окна. Это можно сделать с помощь вызова Dialog.ShowStatus или Dialog.ShowParamStatus. Например, следующее уведомление показывает функцию кнопки пользователю:

```
PROCEDURE ButtonNotifier* (op, from, to: LONGINT);
BEGIN
  IF op = Dialog.pressed THEN
    Dialog.ShowStatus("This button causes the disk to spin down")
  ELSIF op = Dialog.released THEN
    Dialog.ShowStatus("")   (* clear the status message again *)
  END
END ButtonNotifier;
```

Листинг 4-34. Уведомление для отображения данных в статусной строке.

На некоторых платформах, таких как Mac OS, нет статусной строки, поэтому указанный выше код не дает результата.

Часто бывает полезным определить двойной щелчок, например, двойной щелчок на списке может выделить элемент и запустить кнопку по умолчанию. Двойной щелчок распознается уведомлением следующим образом:

```
IF (op = Dialog.pressed) & (from = 1) THEN ...   (* double-click *)
```

Для упомянутого выше случая, в котором реакция на двойной щелчок сводится к простому выполнению команды кнопки по умолчанию, доступно более подходящее стандартное уведомление:

```
StdCmds.DefaultOnDoubleClick
```

Как уже упоминалось ранее, в BlackBox Component Builder обеспечивается возможность подмены строк с помощью карты ключей к собственно строкам, с применением ресурсного файла. Эта особенность так же поддерживается Dialog.ShowStatus. Например, вызов

```
Dialog.ShowParamStatus("This ^0 causes the ^1 to ^2", control, object, verb)
```

позволяет обеспечить разные строки для символов-заполнителей ^0, ^1, ^2. В

действительности будут использованы строки из трех дополнительных параметров control, object и verb. Следует отметить, что строки подменяются перед их сцеплением в первую строку. Особенность подмены строк можно использовать исключительно в вызове процедуры Dialog.MapParamString.

4.7 Составные элементы управления и интеракторы

Выпадающий список имеет содержание 2-х видов: элемент в строке, который взят из списка и текущее выделение. Как правило, список элементов более постоянен, чем выделение, но и он так же может меняться в процессе использования. В случает списка или выделения приложению малоинтересен сам список элементов, так как он является только подсказкой для пользователя. Для приложения интересно выделение, установленное пользователем. Для списка выделение определяется индексом выделенного элемента. Для списка с множественным выбором выделение определяется множеством индексов выделенных элементов. Для комбобокса допустимо не только выделение, но и введенная строка.

Для всех трех типов списочных структур список элементов должен быть как-то построен. Для этой цели в модуле Dialog определяется подходящий интерактор: Dialog.List для списка, Dialog.Selection для списка, Dialog.Combo для комбобокса. Эти типы определяются следующим способом:

```
List = RECORD
   index: INTEGER;   (* индекс текущего выбранного элемента *)
   len-: INTEGER;   (* число элементов списка *)
   PROCEDURE (VAR l: List) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR l: List) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR l: List) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR l: List) SetResources (IN key: ARRAY OF CHAR), NEW
END;

Selection = RECORD
   len-: INTEGER;   (* число выделенных элементов *)
   PROCEDURE (VAR s: Selection) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR s: Selection) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR s: Selection) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR s: Selection) SetResources (IN key: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR s: Selection) Incl (from, to: INTEGER), NEW;
                                    (* выбрать диапазон [от..до] *)
   PROCEDURE (VAR s: Selection) Excl (from, to: INTEGER), NEW;
                                    (* отменить выделение диапазона [от..до] *)
   PROCEDURE (VAR s: Selection) In (index: INTEGER): BOOLEAN, NEW
                                    (* проверить, выделен ли элемент с номером index *)
END;

Combo = RECORD
   item: String;   (* текущая введенная или выделенная строка *)
   len-: INTEGER;   (* число элементов *)
   PROCEDURE (VAR c: Combo) SetLen (len: INTEGER), NEW;
   PROCEDURE (VAR c: Combo) SetItem (index: INTEGER; IN item: ARRAY OF CHAR), NEW;
   PROCEDURE (VAR c: Combo) GetItem (index: INTEGER; OUT item: String), NEW;
   PROCEDURE (VAR c: Combo) SetResources (IN key: ARRAY OF CHAR), NEW
END;
```

Листинг 4-35. Определения для списка, выделения и комбобокса

Перед использованием переменных любого из указанных типов, необходимо

проинициализировать отдельные элементы. В процессе использования элементы могут быть изменены. Например, следующий фрагмент кода строит список:

```
list.SetLen(5);   (* определить длину списка *)
list.SetItem(0, "Daffy Duck");
list.SetItem(1, "Wile E. Coyote");
list.SetItem(2, "Scrooge Mc Duck");
list.SetItem(3, "Huey Lewis");
list.SetItem(4, "Thomas Dewey");
Dialog.UpdateList(list);   (* должен быть вызван после любых изменений элемента списка *)
```

Листинг 4-36. Явное построение списка

В случае применения спискоориентированной структуры (выпадающий список, список, комбобокс), Dialog.Update обновляет только выделение или текстовый ввод элемента управления, но не содержимое списка. Если изменяется структура, то есть элементы списка, то требуется вызывать процедуру Dialog.UpdateList вместо Dialog.Update.

Для фиксированных списков рекомендуется сохранять отдельные строки в ресурсах. Это облегчается применением процедур SetResources. Они ищут строки в ресурсном файле. Например, для полной замены вышеприведенного фрагмента может служить выражение:

```
list.SetResources("#Obx:list")
```

которое читает файл Obx/Rsrc/Strings и ищет вхождения ключей подобно "list[index]", например:

```
list[0]   Daffy Duck
list[1]   Wile E. Coyote
list[2]   Scrooge Mc Duck
list[3]   Huey Lewis
list[4]   Thomas Dewey
```

Таблица 4-37. Ресурс списка

Индексы должны начинаться с 0 и быть последовательными (без пропусков). Процедура SetLen не обязательна. Ее рекомендуется использовать, когда заранее известно число элементов в списке. Для других случаев ее можно опускать. Списки становятся больше по мере роста индексов элементов. SetLen необходима, когда требуется сократить имеющийся список или когда его необходимо целиком очистить.

4.8 Контроль ввода

Контроль правильности ввода и невведения чего-нибудь противоестественного - один из аспектов немодального пользовательского интерфейса. Между прочим, немодальность означает также, что пользователь не должен вмешиваться в режим, зависящий от текущего положения курсора и проверять, правильно ли введены текущие данные или нет.

В частности это означает, что нельзя заставлять пользователя вводить каки-либо данные, прежде чем разрешить дальнейшие действия.

Существует две основные стратегии проверки правильности введенных данных: ранний и поздний. Ранняя проверка проводится в момент работы с элементом управления. Поздняя проверка проводится после полного ввода данных и попытке совершить над ними какое-либо действие, например: ввод новых данных в базу данных.

Поздняя проверка в большинстве случаев подходит при проверке глобально определенных

инвариантов, например: во все ли необходимые поля введены данные. Ранняя проверка больше подходит для локальных, использующих особенности элемента управления, инвариантов, например: исправление синтаксиса введенной строки.

Приведем примеры как ранней, так и поздней проверки для модуля ObxPhoneUI. Примем, что телефонный номер всегда имеет формат: 310-555-1212. Для поздней проверки будет дополнена процедура Lookup (выделенный шрифт) и добавлены несколько вспомогательных процедур. Необходимо отметить, что оператор "$", который используется с LEN возвращает длину строки, а не длину массива.

```
PROCEDURE Valid (IN s: ARRAY OF CHAR): BOOLEAN;

  PROCEDURE Digits (IN s: ARRAY OF CHAR; from, to: INTEGER): BOOLEAN;
  BEGIN  (* проверить, содержит ли s в диапазоне [от..до] только цифры *)
    WHILE (from <= to) & (s[from] >= "0") & (s[from] <= "9") DO INC(from) END;
    RETURN from > to   (* нет найденных нецифр в диапазоне *)
  END Digits;

BEGIN  (*  проверить синтаксис телефонного номера *)
  RETURN (LEN(s$) = 12) & Digits(s, 0, 2) & (s[3] = "-") & Digits(s, 4, 6) &
            (s[7] = "-") & Digits(s, 8, 11)
END Valid;

PROCEDURE ShowErrorMessage;
BEGIN
  phone.name := "illegal syntax of number"
END ShowErrorMessage;

PROCEDURE Lookup*;
BEGIN
  IF phone.lookupByName THEN
    ObxPhoneDB.LookupByName(phone.name, phone.number);
    IF phone.number = "" THEN phone.number := "not found" END
  ELSE
    IF Valid(phone.number) THEN
      ObxPhoneDB.LookupByNumber(phone.number, phone.name);
      IF phone.name = "" THEN phone.name := "not found" END
    ELSE
      ShowErrorMessage
    END
  END;
  Dialog.Update(phone)
END Lookup;
```

Листинг 4-38. Поздняя проверка правильности ввода.

Грубо говоря, напоминание показывает сообщение об ошибке. Если открыто окно рабочего журнала, то сообщение выводится в него, а само это окно перемещается на передний план. Если журнал не используется, ShowErrorMessage выполняет следующее выражение:

```
Dialog.ShowMsg("Please correct the phone number")
```

Рассмотрим альтернативное решение для проверки ввода: ранняя проверка. Для ее реализации применяется уведомление для проверки вода телефонного номера посимвольно на предмет разрешен ли символ или нет. В отличие от поздней проверки, в ранней необходимо оперировать с частично введенными телефонными номерами. Когда обнаружен неверный ввод, строка просто обрезается до легального значения.

```
  PROCEDURE Correct (VAR s: ObxPhoneDB.String);

    PROCEDURE CheckMinus (VAR s: ObxPhoneDB.String; at: INTEGER);
    BEGIN
      IF s[at] # "-" THEN s[at] := 0X END   (* clip string *)
    END CheckMinus;

    PROCEDURE CheckDigits (VAR s: ARRAY OF CHAR; from, to: INTEGER);
    BEGIN
      WHILE from <= to DO
        IF (s[from] < "0") OR (s[from] > "9") THEN
          s[from] := 0X; from := to   (* clip string and terminate loop *)
        END;
        INC(from)
      END
    END CheckDigits;

  BEGIN   (* обрезать строку до легального символа, если необходимо *)
    CheckDigits(s, 0, 2);
    CheckMinus(s, 3);
    CheckDigits(s, 4, 6);
    CheckMinus(s, 7);
    CheckDigits(s, 8, 11)
  END Correct;

  PROCEDURE NumberNotifier (op, from, to: INTEGER);
  BEGIN
    Correct(phone.number)
    (* Dialog.Update не вызывается, потому что он может быть вызван после с помощью
уведомления *)
  END NumberNotifier;
```

Листинг 4-39. Ранняя проверка ввода.

Отметим, что активное выделение, то есть элемент управления не может предотвратить перевод фокуса ввода на другое выделение. Выделение может только просто изменять собственное отображение, содержание или состояние. Для этой цели выделение получает Contorllers.MarkMsg при изменении фокуса ввода.

4.9 Прямой доступ к элементам управления

В большинстве случаев механизм охранников и уведомлений предоставляет достаточный контроль над поведением элементов управления в стиле "Посмотри и почувствуй" (Look&Feel). Тем не менее, иногда необходим прямой доступ к элементам управления на форме. Его описание и представлено в этом разделе. Например, предположим, что необходимо написать свою собственную команду выравнивания и добавить ее в команды меню Layout. Для выполнения этого необходим доступ к форме представления в текущем редактируемом окне. Форма представления является контейнером для элементов управления, которыми хотим манипулировать. Функция FormControllers.Focus предоставляет ссылку на текущую форму с фокусом ввода в редакторе. Таким образом типичный код доступа выглядит следующим образом (Листинг 4-40):

```
    VAR c: FormControllers.Controller;
  BEGIN
    c := FormControllers.Focus();
    IF c # NIL THEN
```

```
    ...
```

Листинг 4-40. Доступ к элементу управления формы

Форма содержит элементы управления и возможно некоторые другие визуальные элементы. При переводе формы в режим разметки, ее можно редактировать. Обычно элемент в начале выделяется, а затем над ним выполняются команды. Следующий пример смещает все выделенные элементы на один сантиметр вправо (листинг 4-41):

```
MODULE ObxControlShifter;

  IMPORT Ports, Views, FormModels, FormControllers;

  PROCEDURE Shift*;
    VAR c: FormControllers.Controller; sel: FormControllers.List;
  BEGIN
    c := FormControllers.Focus();
    IF (c # NIL) & c.HasSelection() THEN
      sel := c.GetSelection();   (* создается список со ссылками на выделенные вьюшки *)
      WHILE sel # NIL DO
        c.form.Move(sel.view, 10 * Ports.mm, 0);   (* сдвинуть вправо *)
        sel := sel.next
      END
    END
  END Shift;

END ObxControlShifter.
```

Листинг 4-41. Смещение всех выделенных элементов вправо.

Этот код работает с выделенными элементами (вне зависимости от того являются ли они управляющими). Иногда требуется оперировать с элементами, которые не выделены. В таком случае необходим считыватель форм (FormModels.Reader) для выполнения последовательных действий над всеми элементами формы с текущим фокусом ввода. Следующий пример кода показывает как команды оперируют со всеми элементами из формы (листинг 4-42):

```
    VAR c: FormControllers.Controller; rd: FormModels.Reader;
  BEGIN
    c := FormControllers.Focus();
    IF c # NIL THEN
      rd := c.form.NewReader(NIL);
      rd.ReadView(v);   (* прочитать первую вьюшку *)
      WHILE v # NIL DO
        ...
        rd.ReadView(v)   (* прочитать следующую вьюшку *)
      END;
      ...
```

Листинг 4-42. Последовательный перебор всех элементов на форме

Элементы управления - это специальные элементы (типа Contorls.Control). Элементы управления могут быть связаны с глобальными переменными. Следующий пример показывает как получить в список все заголовки элементов управления на форме (листинг 4-43):

```
MODULE ObxLabelLister;

  IMPORT Views, Controls, FormModels, FormControllers, StdLog;

  PROCEDURE List*;
    VAR c: FormControllers.Controller; rd: FormModels.Reader; v: Views.View;
  BEGIN
    c := FormControllers.Focus();
    IF c # NIL THEN
      rd := c.form.NewReader(NIL);
      rd.ReadView(v);   (* прочитать первую вьюшку *)
      WHILE v # NIL DO
        IF v IS Controls.Control THEN
          StdLog.String(v(Controls.Control).label); StdLog.Ln
        END;
        rd.ReadView(v)   (* прочитать следующую вьюшку *)
      END
    END
  END List;

END ObxLabelLister.
```

Листинг 4-43. Создание списка заголовков всех элементов управления формы

Подобные операции могут быть проведены с любыми другими элементами на форме, не только с управляющими элементами. Например, допустим, что есть форма с кнопкой "Clear", которая приводит к очистке точечного просмотра. Кнопка - стандартный элемент управления, точечный просмотр - специальный тип. Команда, сопоставленная кнопке сперва должна найти точечный просмотр в той же форме, что и кнопка. Ниже приведен пример кода, демонстрирующего реализацию этой команды (листинг 4-44):

```
    VAR c: FormControllers.Controller; rd: FormModels.Reader; v: Views.View;
  BEGIN
    c := FormControllers.Focus();
    IF c # NIL THEN
      rd := c.form.NewReader(NIL);
      rd.ReadView(v);   (* прочитать первую вьюшку *)
      WHILE v # NIL DO
        IF v IS MyPlotterView THEN
          (* очичстить v(MyPlotterView) *)
        END;
        rd.ReadView(v)   (* прочитать следующую вьюшку *)
      END
    END
```

Листинг 4-44. Поиск обычного просмотра на форме

Имеется одна потенциальная проблема в приведенном выше коде. Допустим, что имеется диалоговое окно, содержащее кнопку с командой ObxLabelLister.List. Под диалоговым окном имеется окно с фокусом ввода. Теперь, если щелкнуть на кнопке диалогового окна, какие заголовки будут собраны в список? Заголовки собственно с диалогового окна или заголовки с окна с фокусом ввода, расположенного под диалоговым? Другими словами: FormControllers.Focus обрабатывает окно, содержащее кнопку по которой только что щелкнули или форму, имеющую элемент управления с фокусом ввода? В зависимости от того, что делает команда по кнопке, оба варианта могут иметь место. Для команды редактирования формы подобных ObxContorlShifter.Shift будет возвращены данные с формы

с фокусом ввода. Это самое верхнее документное окно, только перекрытое диалоговым окном. Наоборот: если требуется найти точечный просмотр, возвращена будет форма диалогового окна. В данном случае ищутся прямые "ближайшие соседи" кнопки.

Решение для поиска точечного просмотра будет начало поиска с контекста только что нажатой кнопки. Это можно сделать как указано в нижеследующем примере (листинг 4-45):

```
    VAR button: Controls.Control; m: Models.Model; rd: FormModels.Reader; v: Views.View;
  BEGIN
    button := Controls.par; (* во время клика на кнопку, эта переменная содержит ссылку на кнопку *)
    m := button.context.ThisModel();  (* получить модель вьюшки-контейнера, который содержит кнопку*)
    IF m IS FormModels.Model THEN   (* контейнер - это форма *)
      rd := m(FormModels.Model).NewReader(NIL);
      rd.ReadView(v);   (* прочитать первую вьюшку *)
      WHILE v # NIL DO
        IF v IS MyPlotterView THEN
          (* очистить v(MyPlotterView) *)
        END;
        rd.ReadView(v)   (* прочитать следующую вьюшку *)
      END
    END
```

Листинг 4-45. Поиск обычного просмотра в том же окне, что и кнопка

В этом коде предполагается, что команда будет выполнятся с кнопки. Она не будет работать из меню. В порядке лучшего разделения пользовательского интерфейса и программной логики этого предположения следует избегать. В BlackBox данная проблема решена предоставлением программисту исключительного контроля над поведением FormControllers.Focus. Хитрость состоит в том, что BlackBox поддерживает два вида окна диалоговых форм: инструментальное окно (tool windows) и вспомогательное окно (auxilary windows) в дополнение к обычному окну документа. Если кнопка расположена на инструментальном окне, FormControllers.Focus обрабатывает форму, являющуюся самым верхним окном документа или возвращает NIL, если такого окна документа не имеется или если фокус ввода самого верхнего окна документа не находится на форме. Если кнопка расположена на вспомогательном окне, FormContorllers.Focus обрабатывает форму диалогового окна, на которой эта кнопка расположена. В следующих абзацах приведено дальнейшее разъяснение различий между двумя видами окон.

Диалоговое окно в виде инструментального окна не является самостоятельным, оно обеспечивает параметры и элементы управления для операций в нижележащем документе. Типичный пример: диалог "Find / Replace", который оперирует с содержимым нижележащего документа. Кнопка диалогового окна запускает команду и эта команда выполняется в представлении, с которым она оперирует. В примере "Find / Replace" - это окно с текстом, имеющим фокус ввода. Под Windows инструментальное окно всегда является самым верхним окном, то есть никогда не перекрывается окнами документов или вспомогательными окнами, только другими инструментальными окнами. В отличие от других окон, инструментальные окна не могут менять размеры или минимизироваться, но могут перемещаться за пределы окна приложения. В Mac OS инструментальные окна частично похожи на обычные, за исключением отсутствия полос прокрутки и возможности изменения размеров.

С другой стороны, вспомогательные окна самодостаточны. Они содержат и оперируют со своими собственными данными. Как минимум, они знают как искать данные для своих операций (например в базе данных). Диалог "Phone Database" - типичное вспомогательное

окно, как и большинство форм ввода данных. Над вспомогательными окнами можно выполнять все операции как и над обычными окнами, они так же могут перекрываться другими обычными или вспомогательными окнами.

Оба вида диалоговых окон хранятся как документы в определенном RSRC каталоге. Но для их открытия в модуле StdCmds определено две разных команды: OpenToolDialog и OpenAuxDialog. Каждая из них принимает переносимое имя файла ресурса как первый параметр и заголовок диалогового окна как второй параметр. Например, могут быть использованы следующие команды меню:

"Find / Replace..." "" "StdCmds.OpenToolDialog('Text/Rsrc/Cmds', 'Find / Replace')" ""

"Phone Database..." "" "StdCmds.OpenAuxDialog('Obx/Rsrc/PhoneUI', 'Phone Database')" ""

Функция FormControllers.Focus возвращает разный результат, в зависимости от того является ли форма инструментальным или вспомогательным окном.

Дальнейшие примеры операция с формами можно найти в модуле FormCmds. Он доступен в исходных кодах и основан на подсистеме Form.

4.10 Резюме

В этой главе обсуждались важные аспекты реализации графического пользовательского интерфейса и его воплощении в виде кодов в BlackBox Component Builder. Имеется три части приложения: логика приложения, логика интерфейса и ресурсы, то есть документы. Четкое разделение этих частей делает программы более понятными, поддерживаемыми и расширяемыми.

В качестве примеров применения системы форм и элементов управления смотри встроенные примеры ObxAddress0, ObxAddress1, ObxAddress2, ObxOrders, ObxControld, ObxDialog и ObxUnitConv. Для более опытных программистов могут быть интересны исходные коды всей подсистемы Form.

За дополнительной информацией об использовании среды разработки Блэкбокс можно обратиться к документации следующих модулей:

DevCompiler, DevDebug, DevBrowser, DevInspector, DevReferences, DevMarkers, DevCmds, StdCmds, StdMenuTool, StdLog, FormCmds, TextCmds.

Чтобы узнать больше о модуле, выделите его имя и выполните команду Info->Documentation. В целях простоты изложения мы описали только наиболее важные команды. Тем не менее, имеются другие полезные команды в меню Info, Dev, Controls, Tools и Layout. Например, несколько "мастеров" для создания основы нового кода. Обратитесь к файлам System/Docu/User-Man, Text/Docu/User-Man, Form/Docu/User-Man и Dev/Docu/User-Man за исчерпывающим руководством пользователя по структуре, редактору (подсистема Text), визуальному проектированию (подсистема Form) и средствам разработки (подсистема Dev).

5 Тексты

Графические интерфейсы пользователя ввели графические элементы в повседневную компьютерную жизнь. Это значительный шаг вперед в сравнении с модальным текстовым интерфейсом, но простой текст нисколько не устарел. Даже в отчетах с большим количеством иллюстраций текст служит необходимым связующим звеном.

В типичном приложении баз данных не выполняются операции над графической информацией. Данные, которые вводятся и выбираются, в большинстве своем являются текстом, и отчеты обычно печатаются в виде текста, оформленного в таблицы.

В виду важности текста в этой главе описывается как можно использовать текстовые абстракции системы Блэкбокс для различных целей. В нескольких примерах будет

использован модуль простейшей базы данных, представленный в предыдущей главе.

Работа будет осуществляться "снизу-вверх" (bottom-up), на примерах будет показано вначале как записать новый текст, затем прочитать имеющийся текст и в конце как изменить имеющийся текст. В процессе обсуждения примеров встретятся текстовые модели/носители, бегунки, отображатели и дополнительные абстракции, такие как шрифты, атрибуты, линейки и др.

5.1 Запись текста

В BlackBox Component Builder'e тексты используются в рамках схемы проектирования носитель-бегунок-отображатель (см. главу 3). Они обеспечивает текстовые модели в качестве носителей для текста, текстовых записывателей и считывателей в качестве бегунков для текстовых моделей. Оба типа описаны в центральном модуле TextModels подсистемы Text. Записыватель текста хранит текущую позицию и текущие атрибуты, которые будут использованы при записи очередного элемента в текст. Ниже приведено полное определение записывателя:

```
Writer = POINTER TO ABSTRACT RECORD
   attr-: TextModels.Attributes;
   (wr: Writer) Base (): TextModels.Model, NEW, ABSTRACT;
   (wr: Writer) Pos (): INTEGER, NEW, ABSTRACT;
   (wr: Writer) SetPos (pos: INTEGER), NEW, ABSTRACT;
   (wr: Writer) SetAttr (attr: TextModels.Attributes), NEW;
   (wr: Writer) WriteChar (ch: CHAR), NEW, ABSTRACT;
   (wr: Writer) WriteView (view: Views.View; w, h: INTEGER), NEW, ABSTRACT
END;
```

Листинг 5-1. Определение типа TextModels.Writer

Наиболее важной процедурой записывателя является WriteChar, она позволяет записать новую литеру в текущую позицию. Если позиция находится внутри текста (т.е. в диапазоне 0..text.Length() - 1), то литера вставляется в эту позицию в тексте. (Здесь имеется отличие от работы с файлами в системе Блэкбокс, где в аналогичном случае старые данные будут переписаны. В случае текстов объекты записи *всегда* вставляют новые литеры, но не перезаписывают.) Если записыватель указывает на позицию в конце текста (т.е. text.Length()), литера дописывается к тексту, и текст становится на один элемент длиннее. После выполнения процедуры WriteChar позиция записывателя в тексте увеличивается на единицу. Текущую позицию записывателя можно узнать, вызвав функцию записывателя *Pos*, и ее можно изменить, вызвав процедуру *SetPos*.

Так как текстовая модель может содержать вьюшки (view) наряду с обычными литерами, имеется процедура WriteView, позволяющая записать вьюшку с конкретными указанными размерами (w для ширины, h для высоты). Как и большинство геометрических размеров в Блэкбоксе, ширина и высота вьюшки измеряется в так называемых *универсальных единицах*, составляющих 1/36000 мм. Это значение выбрано для исключения ошибок округления для большинства разрешений экрана и принтера. Например, дюйм, 1/300 часть дюйма, миллиметр и полиграфический пункт могут быть выражены в универсальных единицах без ошибок округления. Если не указано иное, предполагается, что размеры выражены в универсальных единицах.

Если не требуется указывать определенный размер для вьюшки, то передаются значение Views.undefined для ширины и/или высоты, что позволяет тексту выбрать подходящий размер для записываемой вьюшки.

Как создается объект записи? Как и все бегунки, объекты записи создаются с помощью фабричной функции их носителей. Для этой цели текстовая модель предоставляет процедуру *NewWriter*, которая создает объект записи и соединяет его с текстом. Объект записи

может вернуть текст посредством функции *Base*. Вновь созданный объект записи ставится на конец текста.

Элемент текста содержит не только собственно литеру, но также и ее атрибуты. В поле *attr* объекта записи содержатся атрибуты, которые будут применяться при записи следующей литеры или вьюшки. Текущие атрибуты могут быть изменены вызовом процедуры *SetAttr*, ассоциированной с объектом записи.
В качестве атрибутов текстовая подсистема Блэкбокса поддерживает цвет и вертикальное смещение. Цвета закодированы целыми числами (Ports.Color). Модуль Ports содержит предопределенные константы *black* (черный), *grey6* (серый 6), *grey12* (серый 12), *grey25*, *grey50*, *grey75*, *white* (белый), *red* (красный), *green* (зеленый) и *blue* (синий). "grey6" означает смесь 6% белого и 94% черного, т.е. почти черный. Другие значения серого постепенно становятся все светлее. Вертикальное смещение измеряется в универсальных единицах.

Помимо цвета и вертикального смещения текстовые элементы также имеют атрибут шрифта. Шрифт — это набор изображений, представляющих собой визуальное представление кода литеры. Разные шрифты могут содержать различные изображения для одной и той же литеры. Например, Helvetica выглядит иначе, чем Times или Futiger. Шрифты могут поддерживать не все коды литер, допустимых в Компонентном Паскале. Это не удивительно, так как тип *CHAR* в Компонентном Паскале соответствует стандарту Unicode. Unicode — 16-ти битный стандарт, что означает возможность определения более 65000 различных литер. Первые 256 литер — это литеры набора Latin-1. Первые 128 литер набора Latin-1 — это литеры набора ASCII.

Если в шрифте не определено какое-либо изображение, Блэкбокс выдаст изображение этой литеры из какого-нибудь другого шрифта, либо символ "отсутствующее изображение", т.е. маленький пустой прямоугольник. Объекты, представляющие шрифты, определены в модуле *Fonts*. Здесь нет необходимости более подробно рассматривать этот модуль, достаточно сказать, что указатель типа *Fonts.Font* идентифицирует шрифт с какими-то конкретными значениями начертания, стиля написания, жирности и размера.

Начертание (typeface)
Начертание определяется группой шрифтов, к которой он принадлежит, оно определяет внешний вид для всей группы в целом. Примеры начертаний: Helvetica, Times, Courier. Наименования начертаний представлены в виде литерного массива типа *Fonts.Typeface*.

Стиль написания (Style)
Штифт дополнительно может быть написан наклонно, подчеркнут, зачеркнут или содержать все эти комбинации. Стили написания представлены в виде множества. На данный момент определены элементы Fonts.italic, Fonts.underline и Fonts.strikeout.

Жирность (weight)
Шрифты могут быть разной жирности, то есть в диапазоне от светлого к нормальному и от полужирного к жирному. Обычно шрифт бывает нормальным или полужирным. Жирность определяется как целое число. Имеются предопределенные значения *Fonts.normal* и *Fonts.bold*.

Размер (size)
Шрифты могут отображаться различными размерами, в зависимости от разрешения применяемого устройства вывода. Часто размер задается в пунктах. Пункт — 1/72 часть дюйма. В системе Блэкбокс размер шрифта измеряется в универсальных единицах, которые подобраны так, что пункты (Ports.point) и другие полиграфические единицы могут быть представлены без ошибок округления. Обычные размеры шрифта: 10 * Ports.point или 12 * Ports.point.

В принципе, начертание, стиль, жирность и размер не зависят друг от друга. Это означает, что данное начертание может быть произвольного стиля написания, жирности и размера. Тем не менее, с точки зрения полиграфистов, некоторые комбинации более

предпочтительны, чем другие и могут быть оптимизированы разработчиком шрифта или механизмами операционной системой, связанными со шрифтом.

Атрибуты текста: шрифт, цвет, вертикальное смещение — собраны в объекте типа *TextModels.Attributes*. Слегка упрощая, этот тип определен как:

```
Attributes = POINTER TO RECORD (Stores.Store)
   color-: Ports.Color;
   font-: Fonts.Font;
   offset-: INTEGER
END;
```

Листинг 5-2. Упрощенное определение TextModels.Attributes

Это - тип поля *attr* записывателя. Он является расширением типа *Stores.Store*. Это означает, что подобный объект можно сохранять в файл. Другие примеры хранимых объектов: TextModels.Model и TextViews.View, но не TextModels.Writer. Более детальное описание приведено в части III.

Мы рассмотрели возможности объекта записи для текстов, включая атрибуты текста, которые поддерживается. Типичное использование писателей текста приведено в ниже-следующем программном примере. Образец кода - выдержка из процедуры, показывающий один или несколько важных аспектов обсуждаемых объектов. Подобный фрагмент кода является рецептом, который должен знать владеющий этой темой программист.

```
   VAR t: TextModels.Model; wr: TextModels.Writer; ch: CHAR;
BEGIN
   t := TextModels.dir.New();   (* разместим новую пустую текстовую модель*)
   wr := t.NewWriter(NIL);
   ... produce ch ...
   WHILE condition DO
      wr.WriteChar(ch);
      ... produce ch ...
   END;
```

Листинг 5-3. Образец кода из TextModels.Writer

Фрагмент кода не очень интересен, потому что не показывает, каким образом сгенерированный текст можно отобразить на экране. Объект текста представляет лишь текст с его атрибутами. Он не знает о том, как изобразить этот текст, даже не знает, как его форматировать; т.е. не знает каким образом разбить текст на строки и страницы так, чтобы они уместились в заданном прямоугольнике (отображении, листе бумаги и т.п.). Когда необходимо отобразить текст, нужно обеспечить отображение текста для текстового объекта. Объекты данных, которые могут быть представлены в нескольких вьюшках, обычно называются моделями (models). Отделение модели от отображения — важная проектная схема, которая обсуждалась в главе 2.

Модуль TextModels только экспортирует абстрактный тип записей для текстовых моделей без конкретной реализации. Взамен он экспортирует объект директория (TextModels.dir), который обеспечивает необходимую фабричную функцию (TextModels.dir.New). Этот косвенный способ создания объекта был обоснован в главе 2.

Работа непосредственно с записывателем текста неудобна, так как он не поддерживает запись строк. По этой причине нижеследующий пример использует форматеры для записи текстов. Форматер — это отображатель (смотри главу 3), который содержит текстовый объект записи и обеспечивает отображение высокоуровневых символов, таких как симво-лы языка Компонентный Паскаль в поток простых литер, которые могут быть переданы для работы объекту записи. В следующем примере применяется форматер для записи текста,

очень простой "отчет". Текст представляет собой содержимое базы данных телефонов, по одной строке на запись. Это первая версия отчета по базе данных телефонов реализована в модуле ObxPDBrep0:

```
MODULE ObxPDBRep0;

   IMPORT Views, TextModels, TextMappers, TextViews, ObxPhoneDB;

   PROCEDURE GenReport*;
      VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
         i: INTEGER; name, number: ObxPhoneDB.String;
   BEGIN
      t := TextModels.dir.New();   (* создание пустой текстовой модели *)
      f.ConnectTo(t);              (* соединение фоматера к тексту *)
      i := 0;
      ObxPhoneDB.LookupByIndex(i, name, number);
      WHILE name # "" DO
         f.WriteString(name);      (* первая литерная цепочка *)
         f.WriteTab;               (* символ табуляции *)
         f.WriteString(number);    (* вторая цепочка *)
         f.WriteLn;                (* возврат цепочка *)
         INC(i);
         ObxPhoneDB.LookupByIndex(i, name, number)
      END;
      v := TextViews.dir.New(t); (* создание текстовой вьюшки для созданного текста *)
      Views.OpenView(v)          (* открытие вьюшки в собственном окне *)
   END GenReport;

END ObxPDBRep0.
```

Листинг 5-4. Запись базы данных телефонов с применением форматера.

Следует отметить, что имена и телефоны разделены символом табуляции (09X). Так как позиции табуляции не определены для данного текста, то символ табуляции воспринимается как широкий пробел фиксированного размера (ширина которого равна нескольким ширинам символа обычного пробела). Позже будет разъяснено, каким образом можно определить позиции табуляции.

Результат выполнения команды ObxPDBRep0 приведен на следующем рисунке:

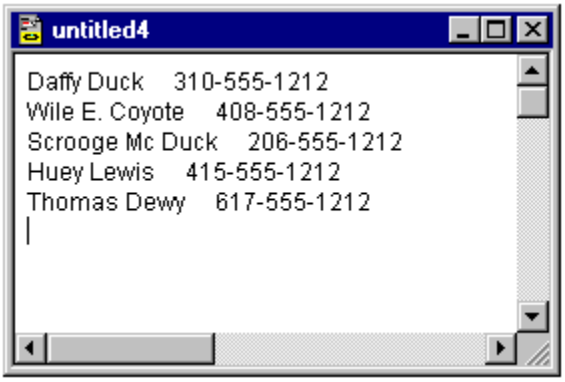

Рисунок 5-5. Окно текстового редактора, содержащее базу данных телефонов.

Текст во вновь открытом окне представляет собой полноценно редактируемую текстовую вьюшку. Ее можно редактировать, сохранить как документ BlackBox, распечатать и т.п. Если посмотреть на исходный код, то можно выделить следующий шаблон:

```
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: Views.View;
  BEGIN
    t := TextModels.dir.New();
    f.ConnectTo(t);
    ... использование процедур форматера для создания текста ...
    v := TextViews.dir.New(t);
    Views.OpenView(v)
```

Листинг 5-6. Шаблон кода для TextMappers.Formatter и для открытия текстовой вьюшки.

Этот шаблон кода встречается во многих командах Блэкбокса, например, DevDebug.ShowLoadedModules. Он удобен во всех случаях, когда нужно написать табличные отчеты различной длины, возможно разделенный на несколько печатных страниц. Так же четко видно, что текст вобрал в себя несколько шаблонов проектирования: объект текста является моделью, которая наблюдается визуальным объектом отображения (проектная схема Наблюдатель), носителем (проектная схема Носитель-Бегунок-Отображатель) и контейнером (проектная схема Компоновщик).

Интересная особенность приведенного выше фрагмента кода: вначале создается текстовая модель t, затем с помощью форматера формируется ее содержимое, и только затем для t создается вьюшка v. Наконец, вьюшка открывается в собственном окне.

В Блэкбоксе важно, что модель может подвергаться изменениям до того, как для него будут созданы какая-либо вьюшки. На самом деле намного эффективнее создавать текст до открытия вьюшек для него, потому что в таком случае не требуется ни обновлений экрана, ни внутренних ресурсов для обеспечения механизма отката, что заметно сказывается на скорости.

В Блэкбоксе текстовая модель представлена в виде последовательности текстовых элементов. Текстовыми элементами могут быть литеры из набора Latin-1 (SHORTCHAR), литеры набора Unicode (a CHAR > 0FFX) или вьюшки (расширения типа Views.View). Текст может иметь атрибуты шрифта, цвета и вертикального смещения. Поскольку модель текста может содержать вьюшки, то это — контейнер. В следующем примере показано, как всё это можно применять при создании нового текста. База данных телефонов будет использоваться как источник данных для вывода в отчет, при этом будут применяться различные представления для текста в различных примерах.

Следующий пример показывает каким образом можно генерировать текст из нашей тестовой базы данных. В этом тексте наименования выводятся зеленым цветом. Различия с модулем ObxPDBRep0 выделены подчеркнутым текстом.

```
MODULE ObxPDBRep1;

  IMPORT Ports, Views, TextModels, TextMappers, TextViews, ObxPhoneDB;

  PROCEDURE GenReport*;
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
      i: INTEGER; name, number: ObxPhoneDB.String;
      default, green: TextModels.Attributes;
  BEGIN
    t := TextModels.dir.New();   (* создать пустую текстовую модель *)
    f.ConnectTo(t);              (* соединение форматера с текстом*)
    default := f.rider.attr;  (* сохранение старых атрибутов для дальнейшего
использования *)
    green := TextModels.NewColor(default, Ports.green);   (* используем зеленый цвет *)
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
```

```
    WHILE name # "" DO
      f.rider.SetAttr(green);   (* поменять текущие атрибуты объекта форматирования *)
      f.WriteString(name);      (* первая литерная цепочка *)
      f.rider.SetAttr(default);  (* поменять текущие атрибуты объекта форматирования *)
      f.WriteTab;               (* символ табуляции *)
      f.WriteString(number);    (* вторая цепочка *)
      f.WriteLn;                (* перевод цепочка *)
      INC(i);
      ObxPhoneDB.LookupByIndex(i, name, number)
    END;
    v := TextViews.dir.New(t); (* создать текстовую вьюшку для созданного текста *)
    Views.OpenView(v)          (* открыть вьюшку в своем собственном окне *)
  END GenReport;

END ObxPDBRep1.
```

Листинг 5-7. Вывод базы данных телефонов в зеленом цвете

Следует отметить, что объект форматирования содержит текущий набор атрибутов (TextModels.Writer.attr). Он включает в себя текущий цвет и доступен только для чтения; текущий набор может быть установлен с помощью процедуры записывателя SetAttr.

Наши первые примеры — ObxPDBRep0 и ObxPDBRep1 — не выводят красивых отчетов, потому что телефонные номера не выровнены по вертикали один под другим в табличном стиле.

ObxPDBRep2 исправляет этот дефект, вставив линейку в начало текста. Эта линейка определяет позиции табуляции, которая выровняет все телефонные номера, отделенные от имен знаком табуляции. Отличия от модуля ObxPDBRep0 помечены подчеркнутым текстом.

```
MODULE ObxPDBRep2;

  IMPORT Ports, Views, TextModels, TextMappers, TextViews, TextRulers, ObxPhoneDB;

  PROCEDURE WriteRuler (VAR f: TextMappers.Formatter);
    CONST cm = 10 * Ports.mm;   (* универсальные определения *)
    VAR ruler: TextRulers.Ruler;
  BEGIN
    ruler := TextRulers.dir.New(NIL);
    TextRulers.AddTab(ruler, 4 * cm);   (* определить остановку табуляции, 4 cm от левого
отступа *)
    TextRulers.SetRight(ruler, 12 * cm);   (* определить правый отступ *)
    f.WriteView(ruler)                  (* линейка - это вьюшка, ее можно записать в текст *)
  END WriteRuler;

  PROCEDURE GenReport*;
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
      i: INTEGER; name, number: ObxPhoneDB.String;
  BEGIN
    t := TextModels.dir.New();   (* создать пустую текстовую модель *)
    f.ConnectTo(t);              (* соединить форматер с текстом *)
    WriteRuler(f);
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
    WHILE name # "" DO
      f.WriteString(name);      (* первая литерная цепочка *)
      f.WriteTab;               (* символ табуляции *)
```

```
        f.WriteString(number);   (* вторая цепочка *)
        f.WriteLn;               (* возврат каретки *)
        INC(i);
        ObxPhoneDB.LookupByIndex(i, name, number)
      END;
      v := TextViews.dir.New(t);   (* создать текстовую вьюшку для созданного текста *)
      Views.OpenView(v)        (* открыть вьюшку в своем собственном окне *)
    END GenReport;

END ObxPDBRep2.
```

Листинг 5-8. Запись телефонной базы данных с применением форматера и линейки

Выполнение ObxPDBRep2.GenReport приводит к появлению окна с улучшенным представлением телефонной базы данных. Выполнение Text->Show Marks делает линейку видимой:

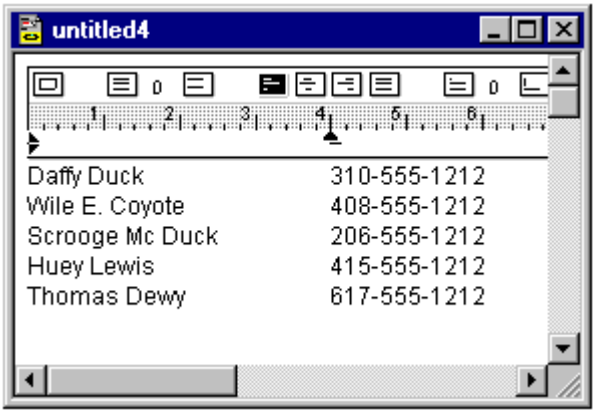

Рисунок 5-9. Текстовый редактор содержащий таблицу базы данных

Линейки — это некоторые вьюшки. Сами по себе они не имеют смысла. Как и любые элементы управления линейки работают только внутри контейнера. В отличие от большинства других элементов управления, линейки являются примером элементов, которые работают правильно только внутри определенного контейнера, в данном случае, внутри текстовой модели. Можно скопировать линейку на форму или другой контейнер, но она останется пассивной и не даст никакого полезного эффекта. Если она внедрена в текстовую модель, которая отображается в визуальном представлении текста, то линейка может повлиять на способ форматирования следующего за ней текста. Например, линейка может определять позиции табуляции, отступы и тип выравнивания.

Параметры линейки действуют в определенной области. Она начинается от начала линейки и заканчивается перед следующей линейкой или доходит до конца текста, если других линеек дальше нет. Линейки определяют новый абзац, возможно со своим собственным набором атрибутов. Вставка символа начала абзаца (0EX) вместо линейки так же начинает новый абзац, но все его параметры наследуются от последней линейки. Символ абзаца — обычный способ начать новый абзац, линейки необходимы только при необходимости изменения атрибутов абзаца. Следует отметить, что символ возврата каретки (0DX) не начинает новый абзац.

Текстовая вьюшка содержит невидимую линейку по умолчанию, которая определяет форматирование текста до первой линейки или всего текста в случае, если линейка не была вставлена вовсе. Линейку по умолчанию, а также настройки по умолчанию можно изменить интерактивно командой меню Text->Make Default Ruler и Text->Make Default Attributes.

У линейки есть много атрибутов, которые можно изменить. В отличие от параметров символов, которые можно применить к произвольному куску текста, атрибуты линейки

применяются к линейке или ко всем абзацам в ее области действия.

Мы обсудим эти атрибуты, так что вы узнаете о функциональности линеек, когда вам это нужно. Однако мы не будем обсуждать линейки более подробно, т.к. вы можете изучить нужные детали, когда вам потребуется более продвинутые возможности линеек (см. документацию к модулю TextRulers).

Стили

Атрибуты линеек не хранятся во вьюшках-линейках напрямую. Вместо этого они привязаны к отдельному объекту типа TextRulers.Attributes, который в свою очередь, содержится внутри объекта типа TextRulers.Style. Стиль - модель одной или нескольких вьюшек-линеек. Если несколько вьюшек связаны с одной и той же моделью, то пользователь может изменить атрибуты всех линеек одновременно. Например, изменение левого отступа вступит в силу для всех линеек, основанных на данном стиле. Тем не менее, модель не может распределяться на несколько документов, что означает, что все границы линеек стилей должны быть внедрены в тот же документ.

TextRulers.Attributes представляют собой коллекцию атрибутов линеек, подобно тому как TextModels.Attributes представляют собой коллекцию атрибутов символов.

```
Attributes = POINTER TO EXTENSIBLE RECORD (Stores.Store)
    first-, left-, right-, lead-, asc-, dsc-, grid-: INTEGER;
    opts-: SET;
    tabs-: TextRulers.TabArray
  END;
```

Листинг 5-10. Упрощенное описание TextRules.Attributes

Начиная со следующего абзаца будут описаны индивидуальные атрибуты. Для работы с этими атрибутами имеются вспомогательные процедуры в модуле TextRulers, которое облегчают установку линеек текста желаемым образом: создается новая линейка при помощи процедуры TextRulers.dir.New(NIL), а затем применяются вспомогательные процедуры для достижения требуемых атрибутов. Пример имеется в приведенной выше процедуре WriteRuler модуля ObxPdBRep2. Эти вспомогательные процедуры значительно облегчают установку текстовых линеек. Например, они прячут то, что атрибуты принадлежат не непосредственно линейкам, а специальному объекту атрибутов, который затем используется линейками. В терминологии шаблонов проектирования [GHJV94] обеспечение облегченного доступа к составным свойствам называется фасадом. Этот механизм простой, потому что фасад состоит из простых процедур, не требуется создавать и управлять какой-то дополнительный объект фасада.

Отступы табуляции

Поле tabs типа TextRulers.TabArray содержит информацию об отступах табуляции. Число отступов табуляции задано в tabs.len и не может быть больше TextRulers.maxTabs (текущее значение 32). Если пользователь вводит больше, чем maxTabs позиций отступов табуляции, или больше, чем заданое len, то лишние позиции будут заменены на пробелы фиксированной ширины. Фиксированная ширина вычисляется как умножение ширины пробела на целое число текущего шрифта, в общем, приблизительно 4 мм.

```
Tab = RECORD
    stop: INTEGER;   (* отступ >= 0 *)
    type: SET   (* тип IN {TextRulers.centerTab, TextRulers.rightTab, TextRulers.barTab} *)
  END;

  TabArray = RECORD
    len: INTEGER;   (* 0 <= len <= TextRulers.maxTabs *)
```

```
    tab: ARRAY TextRulers.maxTabs OF TextRulers.Tab
      (* tab[0 .. len-1] отсортированы в возрастающем порядке без дубликатов *)
  END
```

Листинг 5-11. Определение TextRulers.Tab и TabArray

В tabs.tab.stop задается расстояние от левого края текстовой вьюшки (конечно же, в универсальных единицах). Отступы табуляции в массиве tabs.tab должны быть отсортированы в порядке возрастания и не должны содержать повторяющихся значений.

Кроме позиции отступа, хранящейся в поле stop, в поле type может храниться и набор дополнительных возможностей. В частности, эти возможности позволяют центрировать или выравнивать по правому краю текст внутри абзацного отступа (по умолчанию он выравнивается по левой границе):

tab type tabs.tab[i].type
left adjusted {}
right adjusted {TextRulers.rightTab}
centered {TextRulers.centerTab}

В дополнение к вышеперечисленным типам есть возможность дополнительно определить будет ли отрисовываться вертикальная прографка (vertical bar). Прографка определяется включением в набор type значения TextRulers.barTab. Так как прографка занимает всю высоту строки, ее удобно применять для простых таблиц.

После вызова TextRulers.dir.New(NIL), линейка не содержит отступов табуляции.
Следующие вспомогательные процедуры обеспечивают изменения линеек:

```
PROCEDURE AddTab (r: Ruler; x: INTEGER)
PROCEDURE MakeRightTab (r: Ruler)
PROCEDURE MakeCenterTab (r: Ruler)
PROCEDURE MakeLineTab (r: Ruler)
```

AddTab добавляет нормальный левовыровненный отступ табуляции. Он может быть изменен на правовыровненный или центральный вызовом MakeRight или MakeCenterTab. Вне зависимости от режима выравнивания, отступ табуляции может быть переведен в отступ с прографкой вызовом MakeLineTab.

Поля

Текст представляет из себя набор с левым и правым полем. Эти поля можно определить как расстояние от левого края текстовой вьюшки. Левое поле должно быть меньше правого поля.

```
left-, right-: INTEGER   (* (left >= 0)  &  (right >= left) *)
```

Иногда необходимо уведомить разметчика текста о необходимости игнорировать правое поле и использовать для отображения текста символ разрыва строки. Фактически, это поведение по умолчанию. Это означает, что при изменении размера вьюшки могут так же измениться разрывы строки. Даже если правое поле игнорируется, оно все равно полезно: текстовая вьюшка получает примерную оценку насколько широко следует ее открыть.

Разрывы строк при фиксированном правом поле необходимы в очень редких обстоятельствах, например, когда следующий абзац необходимо уместить рядом с предыдущим. Если необходимо, чтобы строки разрывались только по правому полю, то необходимо включить TextRulers.rightFixed в набор opts (смотри ниже).

Отметим, что команда Tools->Document Size... позволяет гибко назначить ширину (и высо-

ту) первоначальной вьюшки документа, так же называемой корневой вьюшкой (root view). Ширина может быть назначена фиксированным размером, что полезно для приложений с разметкой страниц.

Тем не менее, чаще всего ширина определяется настройками страницы, которая наследует ширину и высоту отображения для текущей выбранной страницы, урезанные на пользовательские поля. Это отвечает требованиям таких документов как письма, программный код и т.п.

Третьей возможностью может быть ограничение ширины корневой вьюшки рамками размеров окна (как упоминалось выше, когда отображение использует размер правого поля из линейки как подсказку, вне зависимости от того фиксирована ли правое поле или нет). При любом изменении размеров окна, корневая вьюшка изменяет размер соответствующим образом, чтобы в окне уместилось столько информации, сколько возможно. Это удобно для типичного окна встроенной документации, в котором информация должна занимать столько места на экране, сколько для этого позволит пользователь, текст должен адаптироваться по ширине вне зависимости от изменения размеров окна. Примером подобного стиля является форматирование по умолчанию для Веб-страницы.

Следует отметить, что эта особенность не зависит от типа корневой вьюшки. Она работает подобным образом для всех вьюшек, способных менять размеры, таких как текстовые вьюшки, формы и большинство других (вьюшек-)редакторов.

При создании при помощи TextRulers.dir.New(NIL) линейка получает левое поле равное 0 и правое нефиксированное поле больше 0 в зависимости от специфики реализации.

Следующие вспомогательные процедуры обеспечивают изменение линейки:

```
PROCEDURE SetLeft (r: Ruler; x: INTEGER)
PROCEDURE SetRight (r: Ruler; x: INTEGER)
PROCEDURE SetFixedRight (r: Ruler; x: INTEGER)
```

SetLeft и SetRight устанавливают левое и правое поле. SetFixedRight выставляет правое поле и делает его фиксированным.

Отступ первой строки

Линейки задают отступ первой строки, то есть насколько первая строка после линейки будет сдвинута от левой границы текстовой вьюшки. Следует отметить, что можно задать отступ первой строки меньше, чем левое поле (first < left). Это возможно только в случае, если текстовая вьюшка выводит текст при определенном значении левого поля.

```
first-: INTEGER;   (* first >= 0 *)
```

Отступ первой строки применяется для всех первый строк каждого абзаца в пределах действия линейки.

При создании линейки с помощью TextRulers.dir.New(NIL) линейка не имеет отступа для первой строки (first = 0). Следующая вспомогательная процедура обеспечивает изменение для линейки:

```
PROCEDURE SetFirst (r: Ruler; x: INTEGER)
```

Над- и подстрочные элементы

При при отрисовке символа его изображение зависит от атрибутов шрифта, таких как гарнитура (Helvetica или Times?), стиль (обычное или наклонное?), начертание (нормальное или жирное) и размер. При отрисовке строки целиком все символы размещаются на обычной базовой линии. Надстрочный элемент - максимальный выступ символа над базовой линией. Некоторые символы имеют выступ под базовую линию, например "y", "p". Подстрочный элемент - максимальный выступ любого символа под базовую линию (см. рис.

5-12). Разметчик текста должен вычислять расстояние между базовыми линиями для предотвращения перекрытия надстрочных и подстрочных элементов соседних строк во избежание наложения символов. Во избежание вырожденных случаев, таких как пустая строка или строка с экстремальными размерами шрифтов, линейки могут задать минимальные значения для надстрочных (asc) и подстрочных (des) элементов.

```
asc-, dsc-: INTEGER;   (* (asc >= 0)  &  (dsc >= 0) *)
```

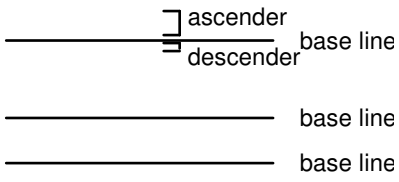

Рисунок 5-12. Над-, подстрочные элементы и базовая линия

При создании линейки с помощью TextRulers.dir.New(NIL) для надстрочных и подстрочных элементов используются значения текущего шрифта по умолчанию (Fonts.dir.Default()) в качестве начальных значений.

Следующие вспомогательные процедуры обеспечивают изменение линейки:

```
PROCEDURE SetAsc (r: Ruler; h: INTEGER)
PROCEDURE SetDsc (r: Ruler; h: INTEGER)
```

Отбивка абзаца

Отбивка абзаца определяет дополнительное расстояние по вертикали, добавляемое между последней строкой предыдущего абзаца и первой строкой следующего. При использовании разметки сетки строк (смотри ниже), считается хорошим стилем выставлять отбивку кратной строке, чаще всего половине строки. Отбивка позволяет визуально отличить новый абзац от простого перевода каретки.

```
lead-: INTEGER;   (* lead >= 0 *)
```

При создании линейки с помощью TextRulers.dir.New(NIL) отбивка устанавливается в 0.
Следующая вспомогательная процедура обеспечивает изменение линейки:

```
PROCEDURE SetLead (r: Ruler; h: INTEGER)
```

Сетка линеек

Если значение grid равно нулю, то линии располагаются настолько близко, насколько это возможно, чтобы не оставалось свободного месте между надстрочными и подстрочными элементами соседних строк. Разные строки могут иметь разные надстрочные и подстрочные элементы, в зависимости от текста и шрифта содержимого. По этой причине расстояние между базовыми линиями необходимо делать нерегулярным. Из соображений типографики, нерегулярное расстояние между строками часто нежелательно.

Для закрепления регулярного расположения сетки линеек возможно установка сетки линеек путем присвоения значению grid значения больше нуля. Это значение определяет вертикальную сетку, согласно которой принудительно выставляются базовые линии. В случае, если строка слишком высокая для размещения на следующем участке сетки, то она располагается на одну или несколько позиций сетки ниже (см рисунок 5-14).

```
grid-: INTEGER;   (* grid >= 0 *)
```

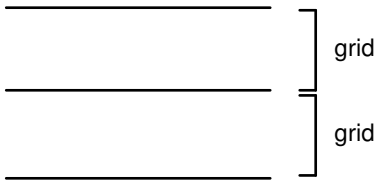

Рисунок 5-13. Сетка строк

В момент создания с помощью TextRulers.dir.New(NIL) линейка имеет значение линий сетки равное 1. Следующая вспомогательная процедура обеспечивает изменение линейки:

PROCEDURE SetGrid (r: Ruler; h: INTEGER)

Режимы выравнивания

Для длинных участков текста бывает необходимо разрывать строки между полями отображения. В зависимости от настроек, поведение текста может быть различным. Текст может быть выровнен по левому краю (левая выключка), по правому краю (правая выключка), центрирован или выровнен одновременно по левому и правому полям (полная выключка). Эти четыре режима выравнивания контролируются двумя флагами: TextRulers.leftAdjust и TextRulers.rightAdjust

leftAdjust	**rightAdjust**	**adjustment mode**
FALSE	FALSE	centered
FALSE	TRUE	right adjusted
TRUE	FALSE	left adjusted (default)
TRUE	TRUE	fully adjusted

Таблица 5-14. Режимы выравнивания

Режимы выравнивания могут быть включены или выключены исчерпывающим набором, оформленным в виде множества:

opts-: SET *(* opts это подмножество {leftAdjust, rightAdjust, pageBreak, rightFixed, noBreakInside, parJoin; другое множество элементов зарезервировано *)*

В момент создания с помощью TextRulers.dir.New(NIL) значение набора выравнивания установлено в {leftAdjust}.

Следующие вспомогательные процедуры обеспечивают модификации линейки:

PROCEDURE SetLeftFlush (r: Ruler)
PROCEDURE SetCentered (r: Ruler)
PROCEDURE SetRightFlush (r: Ruler)
PROCEDURE SetJustified (r: Ruler)

Разрывы страницы

Линейки могут выполнять принудительный переход на новую страницу путем включения TextRulers.pageBreak в набор opts. Элемент TextRulers.noBreakInside действует на всей области видимости или на абзац. Это означает принудительный перенос начала абзаца на новую страницу, кроме случая, когда страница начинается сразу после конца абзаца.

Элемент TextRulers.parJoin подобен noBreakInside за исключением дополнительной провер-

ки, что как минимум первая строка следующего абзаца будет на той же странице.

Когда линейка создается с помощью TextRulers.dir.New(NIL), у ней нет никаких из вышеперечисленных специальных атрибутов. Следующие вспомогательные процедуры обеспечивают модификации линеек:

```
PROCEDURE SetPageBreak (r: Ruler)
PROCEDURE SetNoBreakInside (r: Ruler)
PROCEDURE SetParJoin (r: Ruler)
```

На этом описание атрибутов линеек завершено. Атрибуты линеек применяются ко всем линейкам или параметрам в пределах видимости. Атрибуты символов напротив применяются только для конкретного участка текста. Текстовая модель Блэкбокса поддерживает все атрибуты текста плюс цвет и вертикальное смещение.

Линейки — это вьюшки, которые могут быть вставлены в текст, поскольку текст является контейнером. Линейки по определению знают, что такое текстовая система, но текстовой вьюшке необходимо знать о параметрах абзацев, выставляемых с помощью линеек. Тем не менее, несколько других отображений, которые ничего не знают о текстовой системе, могут быть вставлены в текст. В частности интересные эффекты могут быть достигнуты, если вьюшки знают о тексте. Закладки (Fold views) и ссылки (link views) — примеры подобных текстоориентированных вьюшек.

Следующий пример показывает, как можно создать закладку. Закладки - стандартный элемент системы Блэкбокс; они реализована в модуле StdFolds с применением только средств Блэкбокса и текстовой подсистемы. Закладки идут парами: левая закладка и правая закладка. Левая закладка содержит скрытый текст. Когда пользователь щелкает на одном из отображений закладки, текст между значками заменяется на скрытый текст. Обычно закладки применяются для подмены большого участка текста его короткой характеристикой, например, глава книги может быть спрятана, если виден только заголовок главы. Это один из способов увеличения абстракции документа путем сокрытия деталей. По этой причине одно состояние вьюшки-закладки называется свернутым, другое — развернутым состоянием. Нет технических причин для того, чтобы скрытый текст был короче, чем видимый; отображение наименования лишь пример наиболее типичного использования закладок.

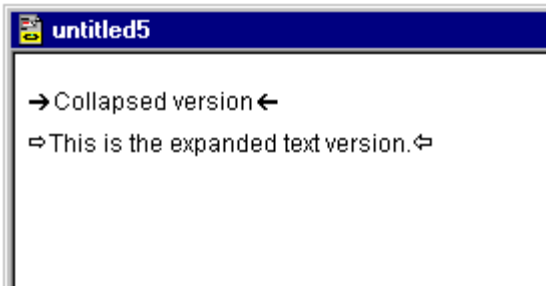

Рисунок 5-15. Закладка в свернутом и развернутом состояниях

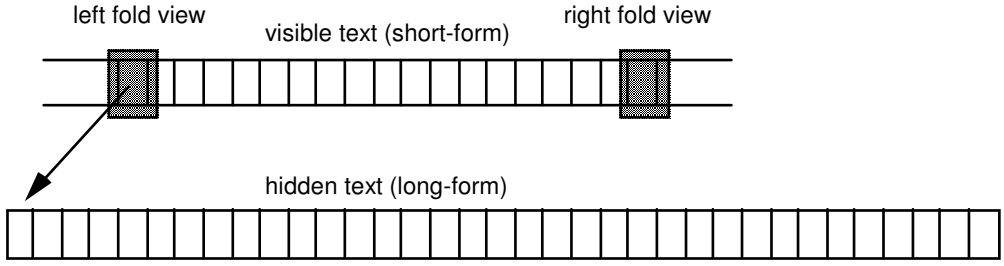

Рисунок 5-16. Структура текстовой закладки

```
MODULE ObxPDBRep3;
```

```
  IMPORT Views, TextModels, TextMappers, TextViews, StdFolds, ObxPhoneDB;
  PROCEDURE WriteOpenFold (VAR f: TextMappers.Formatter;
                           IN shortForm: ARRAY OF CHAR);
    VAR fold: StdFolds.Fold; t: TextModels.Model;
  BEGIN
    t := TextModels.dir.NewFromString(shortForm); (* преобразование цепочки в текстовую
модель *)
    fold := StdFolds.dir.New(StdFolds.expanded, "", t);
    f.WriteView(fold)
  END WriteOpenFold;

  PROCEDURE WriteCloseFold (VAR f: TextMappers.Formatter);
    VAR fold: StdFolds.Fold; len: INTEGER;
  BEGIN
    fold := StdFolds.dir.New(StdFolds.expanded, "", NIL);
    f.WriteView(fold);
    fold.Flip;  (* замена длинного текста, теперь между двумя вьюшками-закладками, с
коротким текстом *)
    len := f.rider.Base().Length();  (* определить, что носитель имеет новую длину *)
    f.SetPos(len)  (* установить позицию форматера в конец текста *)
  END WriteCloseFold;

  PROCEDURE GenReport*;
    VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
      i: INTEGER; name, number: ObxPhoneDB.String;
  BEGIN
    t := TextModels.dir.New();  (* создать пустую текстовую модель *)
    f.ConnectTo(t);             (* соединить форматер с текстом *)
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
    WHILE name # "" DO
      WriteOpenFold(f, name$);  (* записать левую закладку-вьюшку в текст, с ее именем
в качестве короткого текста *)
      (* теперь записать длинный текст *)
      f.WriteString(name);      (* первая строка *)
      f.WriteTab;               (* символ табуляции *)
      f.WriteString(number);    (* вторая строка *)
      WriteCloseFold(f);        (* записать закрывающую вьюшку, заменить длинный и
короткий текст *)
      f.WriteLn;                (* возврат каретки *)
      INC(i);
      ObxPhoneDB.LookupByIndex(i, name, number)
    END;
    v := TextViews.dir.New(t);  (* создать текстовую вьюшку для созданного текста *)
    Views.OpenView(v)           (* открыть вьюшку в своем собственном окне*)
  END GenReport;

END ObxPDBRep3.
```

Листинг 5-17. Запись телефонной базы данных с применением закладок

Процедура WriteCloseFold имеет две интересные особенности. Во-первых, закладка записывается в развернутом состоянии. На это есть причина: часто развернутая форма более сложна, чем свернутая, поэтому более удобно использовать WriteOpenFold (следует отметить удобную процедуру TextModels.dir.NewFromString). Если свернутая форма более сложная, удобней воспользоваться уже имеющимся форматером. После записи левой части,

развернутого текста и правой части закладки, текст может быть свернут процедурой StdFolds.Flip.

Это приводит ко второму интересному моменту: свертывание текста приводит к изменению текстовой модели. Это приводит к устареванию всех бегунков, и, как результат, всех форматеров, базирующихся на тексте. Например, форматер f всегда должен находится в конце созданного текста, но когда текст свернули, его принадлежность к концу текста не соответствует действительности. По этой причине необходимо заново выставить его позицию.

Это является общим поведением при работе с переменными носителями: при любом изменении статуса носителя (вне зависимости от того, кто это сделал) необходимо обновить некоторые статусы состояний бегунков. В данном примере изменяется длина текста, что приводит к необходимости обновления позиции форматера.

В последнем примере из текущей серии будет показано как создать текст с использованием ссылочных вьюшек. Ссылки - стандартный элемент BlackBox Component Builder; они реализованы в модуле StdLinks только на основе базовой подсистемы и подсистемы Text. Подобно закладкам, ссылки используются парами. Их реализация применяет специальный интерфейс текстовой подсистемы (TextControllers.FilterPollCursorMsg,

TextControllers.FilterTrackMsg), который заменяет стандартный указатель мыши на изображение пальца (hand cursor) при наведении его на текст между вьюшками ссылки. Такое поведения известно по типичным гипертекстовым системам, подобным броузерам, когда перемещение указателя мыши по тексту символизирует гиперссылку. Прямое назначение ссылки следует из его названия. Тем не менее имеется более общее назначение, чем простая разметка. Гиперссылка содержит пассивную связь с другим документом или с другим местом этого же документа. Например, ссылка на документ может быть сохранена в виде строки: "http://www.oberon.ch" или "BlackBox/System/Rsrc/About"

В отличие от команд Component Pascal, ссылка дополнительно может иметь один или два строковых параметра. Команда выполняется после щелчка мыши пользователя при нахождении ее указателя на тексте между двумя вьюшками ссылки.

Команда сохранена в левой части пары вьюшек ссылки. Типичная команда:

```
DevDebug.ShowLoadedModules

StdCmds.OpenDoc('System/Rsrc/Menus')

StdCmds.OpenAux('System/Rsrc/About', 'About BlackBox')
```

В нашем примере необходимо создать вьюшки ссылки со следующей примерной командой:

```
ObxPDBRep4.Log('Daffy Duck    310-555-1212')
```

Замечание: необходимо проконсультироваться по модулю StdInterpret, так как могут иметься некоторые ограничения по поддержке возможного списка параметров или команд.

Процедура Log в приведенном примере просто записывает строку, переданную как параметр, в журнал. В более реалистичном примере параметр может применяться для набора соответствующего номера.

```
MODULE ObxPDBRep4;

   IMPORT Views, TextModels, TextMappers, TextViews, StdLinks, StdLog, ObxPhoneDB;

   CONST cmdStart = "ObxPDBRep4.Log('"; cmdEnd = "')";

   PROCEDURE GenReport*;
      VAR t: TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
         i: INTEGER; name, number: ObxPhoneDB.String; link: StdLinks.Link;
```

```
  BEGIN
    t := TextModels.dir.New();   (* создать пустую текстовую модель *)
    f.ConnectTo(t);              (* соединить форматер с текстом*)
    i := 0;
    ObxPhoneDB.LookupByIndex(i, name, number);
    WHILE name # "" DO
      link := StdLinks.dir.NewLink(cmdStart + name + "    " + number + cmdEnd);
      f.WriteView(link);
      f.WriteString(name);  (* строка показывается между двумя ссылочными вьюшками *)
      link := StdLinks.dir.NewLink("");
      f.WriteView(link);
      f.WriteLn;                 (* возврат каретки *)
      INC(i);
      ObxPhoneDB.LookupByIndex(i, name, number)
    END;
    v := TextViews.dir.New(t);   (* создать текстовую вьюшку для созданного текста *)
    Views.OpenView(v)         (*открыть вьюшку в своем собственном окне *)
  END GenReport;

  PROCEDURE Log* (param: ARRAY OF CHAR);
  BEGIN
    StdLog.String(param); StdLog.Ln
  END Log;

END ObxPDBRep4.
```

Листинг 5-18. Запись телефонной БД с применением ссылок.

Следует отметить, что команды являются вызовами обычной процедуры; как и все вызовы процедур за пределами модуля, возможен только вызов процедуры объявленной как экспортируемая. Процедура Log из представленного модуля должна быть экспортирована, в противном случае программа не будет работать.

Другим интересным свойством ссылочных вьюшек является разный способ отрисовки в зависимости от того заказал пользователь отображать или прятать метки для текстовой вьюшки. При наличии нескольких текстовых вьюшек одной модели пользователь может независимо переключать отметки для отображения или для скрытия, а так же деактивировать мягкие переносы линеек или вьюшек ссылки. Текстовая подсистема обеспечивает интерфейс, который позволяет вьюшкам различать два состояния (TextSetters.Pref). В частности, вьюшка может принять решение для своего сокрытия, когда не отображаются метки, выставляя собственную ширину в ноль. Это то, что и делает ссылочная вьюшка: когда текстовые отметки выключены, они невидимы, а виден только текст между ними. Обычно текст отображается подчеркнутым и синим цветом, для обозначения ссылки. Линейки, напротив, всегда занимают все пространство между левым и правым полем абзаца, но всегда выставляют свою ширину в ноль при выключенном отображении метки. О взаимодействии контейнеров и внедренных вьюшек для достижения подобных эффектов будет рассказано дальше в части III этого учебника.

В приведенном выше примере показано, что ссылочные вьюшки могут вести себя разными путями при щелчке пользователя между ними. В общем, это мощное средство, но его нужно использовать с осторожностью. Команды в документе представляют собой вид "безусловного импорта" взаимосвязей между документами и командными модулями и поэтому порождают зависимости. Это не является фундаментальным отличием от случая зависимостей между внедренным в документ вьюшками, они так же функционируют правильно только при доступности задействованного компонента(ов). Подобно всем зависимостям, как в случае ссылки, внедрения кнопки на элемент управления, внедрения редактора в документ и других, их необходимо отслеживать и обновлять по мере

необходимости.

5.2 Шаблон Модель-Отображение-Контроллер в применении к текстам

Следующий основной пример показывает, как можно прочитать имеющийся текст. Предполагается, что обрабатываемый текст видимый, в самом верхнем окне. По доступу к тестам, сохраненным в файле обращайтесь к встроенной документации Obx на примерах ObxOpen0, ObxOpen1 и ObxAscii. Перед началом рассказа о специфике чтения текстов необходимо рассказать каким образом можно получить доступ к содержимому самого верхнего окна с помощью команд Component Pascal.

По существу это делается вызовом Controllers.FocusView(). Функция возвращает NIL если нет открытого документа. В других случаях возвращается содержимое самого верхнего окна, то есть вьюшка.

Ранее было сказано, что вьюшки, основанные на общем типе Views.View есть центральная абстракция Блэкбоксе для интерактивных компонентов, таких как редакторы и элементы управления. Все вращается вокруг вьюшек. В простых случаях, таких как элементы управления, реализация вьюшек представляет собой расширения Views.View и все. В более сложных случаях данные отображаются пользователю в отображениях, разделенных на несколько моделей. Это подробно обсуждалось в главе 2.

Так как реализация контейнеров является одной из самых сложных задач программирования с применением BlackBox, среда разработки обеспечивает помощь в методике построения контейнеров. Абстрактный проект контейнера приведен в модуле Containers. В модуле определены расширения типов Views.View, Models.Model и Controllers.Controller, которые представляют собой миниоболочку для построения контейнеров.

Обычно реализация контейнера содержит как минимум четыре модуля: один для модели, один для вьюшки, один для контроллеров (controller) и один для стандартных команд. Например, текстовая подсистема состоит из модулей TextModels, TextViews, TextControllers, TextCmds и дополнительных модулей TextMappers, TextRulers и TextSetters. Подсистема форм состоит из модулей FormModels, FormViews, FormControllers и FormCmds.

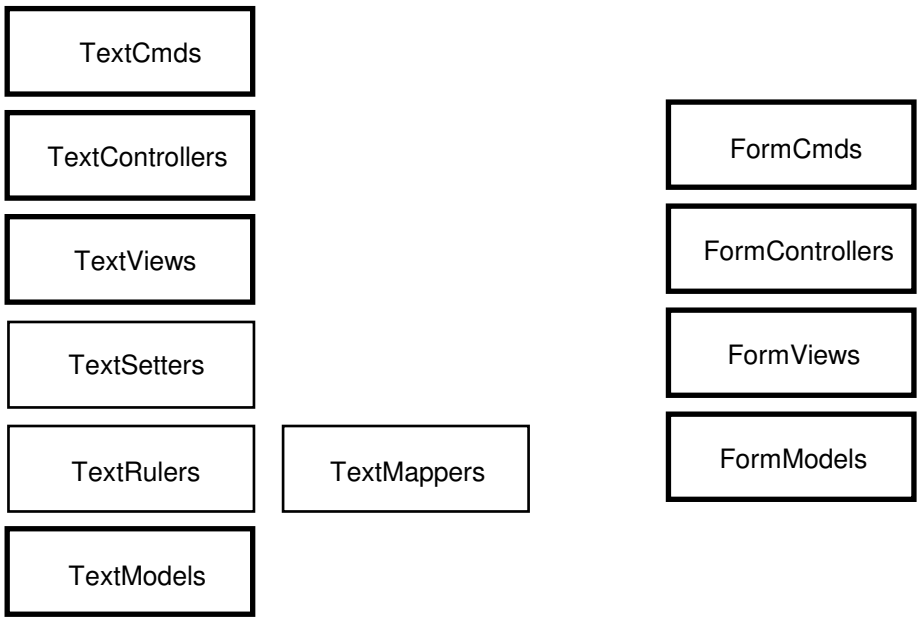

Рисунок 5-19. Варианты декомпозиции основных контейнеров

Сейчас не рассматриваются эти типы детально; но необходимо знать, что объекты контейнеров не только разделены на модели или несколько вьюшек, но так же и то, что вьюшки разделяются в дальнейшем на вьюшки и так называемые контроллеры. Со временем будет уделено внимание управлению выделением и положению фокуса ввода,

так как основная функциональность сосредоточена в контроллере. Поэтому сейчас заостряется внимание на контроллерах, потому что последующие примеры оперируют с текущим выделением.

Если имеется контроллер, то извлечь его вьюшку можно вызовом процедуры ThisView. Другие процедуры позволяют получить или выставить текущее выделение.

Может быть несколько вьюшек, показываемых в одной модели (отношение один-ко-многим), но всегда только один контроллер на одну вьюшку (отношение один-к-одному). (Именно поэтому вьюшки произведены от Containers.View, другие вьюшки могут содержать, а могут и не содержать отдельные контроллеры). Можно получить вьюшку контроллера с помощью процедуры ThisView, и можно получить контроллер для вьюшки с помощью процедуры ThisController.

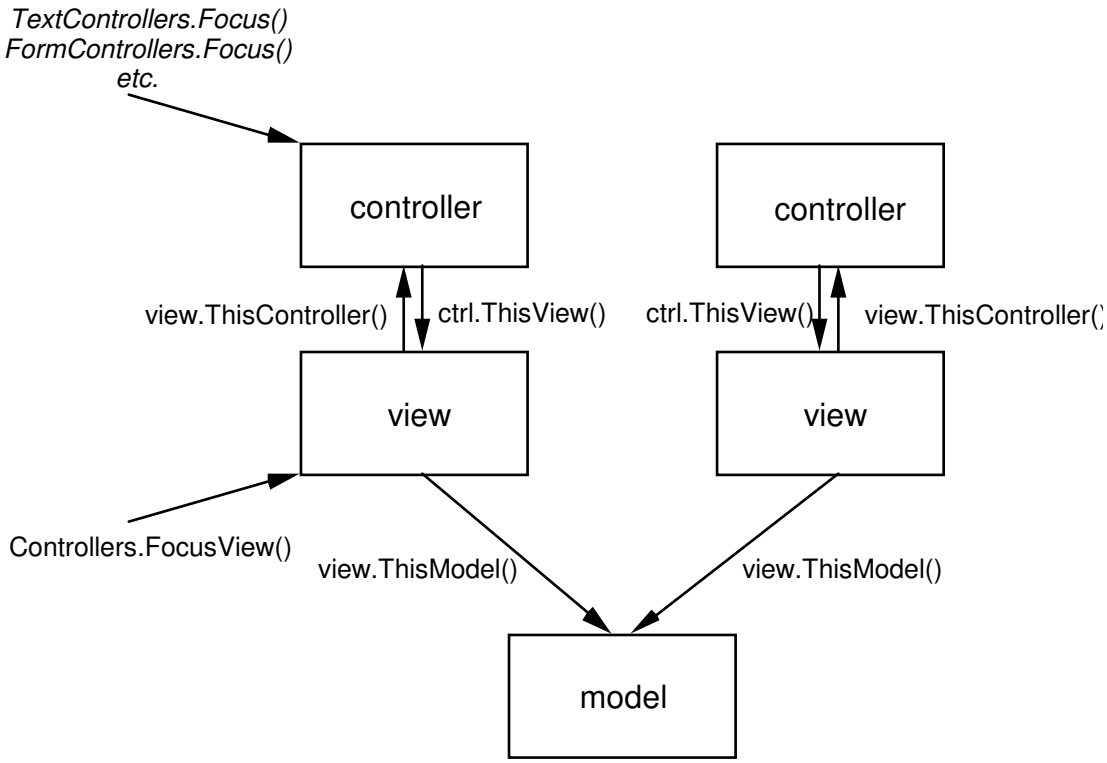

Рисунок 5-20. Строение абстракции Модель-Отображение-Контроллер. (Model-View-Controller separation)

Для более специфических абстракций контейнеров обычно предоставляется подходящая функция Focus, которая извлекает контроллер для вьюшки, владеющей фокусом в данный момент, если и когда она имеет подходящий тип.

Теперь у нас достаточно знаний о том, что такое контейнеры, чтобы описать способы доступа к вьюшкам, моделям и контроллерам, находящимся в фокусе.

Предположим, что сделаны следующие объявления:

```
VAR v: Views.View; m: Models.Model; c: Controllers.Controller;
```

Простая вьюшка (без модели):

```
v := Controllers.FocusView();
```

Вьюшка с моделью:

```
v := Controllers.FocusView();
IF v # NIL THEN m := v.ThisModel() ELSE m := NIL END;
```

Вьюшка, являющаяся расширением типа *Containers.View*:

```
    v := Controllers.FocusView();
    IF v # NIL THEN m := v.ThisModel() ELSE m := NIL END;
    IF (v # NIL) & (v IS Containers.View) THEN
      c := v(Containers.View).ThisController()
    ELSE
      c := NIL
    END
```

Листинг 5-21. Доступ к вьюшке, модели и контроллеру, владеющими фокусом

Типичная абстракция контейнера, подобная текстовой подсистеме или подсистеме форм, использует указанный выше механизм, чтобы обеспечить более удобные функции доступа, основанные на особенностях типа контейнера. Например, модуль TextControllers экспортирует функцию Focus, которая возвращает фокус контроллера вьюшки когда и если вьюшка с фокусом является текстовой вьюшкой. В других случаях возвращается NIL. Этот подход оправдан, так как если имеется контроллер контейнера, необходим легкий доступ к его вьюшке, модели, информации о выделении или положении фокуса ввода и т.п. Типичный шаблон кода является подмножеством нижеприведенного, в зависимости от того какая часть контролера необходима:

```
PROCEDURE SomeCommand*;
  VAR c: TextControllers.Controller;
    v: TextViews.View; t: TextModels.Model; from, to, pos: INTEGER;
    ...
BEGIN
  c := TextControllers.Focus();
  IF c # NIL THEN
    ...
    v := c.ThisView();   (*если нужна текстовая вьюшка... *)
    ...
    t := v.ThisModel();   (*или нужна текстовая модель... *)
    ...
    IF c.HasSelection() THEN   (* или нужен диапазон выделения... *)
      c.GetSelection(from, to);
      ...
    END;
    ...
    IF c.HasCaret() THEN   (* или нужна позиция знака вставки... *)
      pos := c.CaretPos();
      ...
    END;
    ...
  ELSE   (* не открывается окно документа, или фокус имеет неверный тип *)
    (* не обрабатывать ошибку необходимо если эта команда охраняется подходящей
командой охранника *)
  END
END SomeCommand;
```

Листинг 5-22. Шаблон кода для TextControllers.Controller

Команда меню может использовать охранника TextCmds.FocusGuard для уверенности, что команда должна быть доступна только при наличии фокуса на текстовой вьюшке. TextCmds.SelectionGuard дополнительно проверяет наличие (непустого) выделения текста.

5.3 Чтение текста

Зная, как получить доступ к контроллеру вьюшки, находящейся в фокусе, напишем команду для чтения текста из вьюшки в фокусе. Команда подсчитает количество символов из наборов Latin-1 (1 байт), Unicode (над Latine-1, 2 байта) и количество внедренных вьюшек.

```
MODULE ObxCount0;

  IMPORT TextModels, TextControllers, StdLog;

  PROCEDURE Do*;
  (** используйте TextCmds.SelectionGuard в качестве охранника для этой команды **)
    VAR c: TextControllers.Controller; from, to, schars, chars, views: INTEGER;
      rd: TextModels.Reader;
  BEGIN
    c := TextControllers.Focus();
    IF (c # NIL) & c.HasSelection() THEN
      c.GetSelection(from, to);   (* взять диапазон выделения; от < до *)
      rd := c.text.NewReader(NIL);   (* создать новый объект чтения для текстовой модели
*)
      rd.SetPos(from);   (* установить считывателя на начало выделения *)
      rd.Read;           (* считать первый элемент текстового выделения *)
      schars := 0; chars := 0; views := 0;   (* переменные для подсчета *)
      WHILE rd.Pos() # to DO   (* прочесть все элементы выделения *)
        IF rd.view # NIL THEN   (* элемент — вьюшка *)
          INC(views)
        ELSIF rd.char < 100X THEN   (* элемент — литера набора Latin-1 *)
          INC(schars)
        ELSE   (* элемент — литера набора Unicode *)
          INC(chars)
        END;
        rd.Read      (* считать следующий элемент *)
      END;
      StdLog.String("Latin-1 characters: "); StdLog.Int(schars); StdLog.Ln;
      StdLog.String("Unicode characters: "); StdLog.Int(chars); StdLog.Ln;
      StdLog.String("Views: "); StdLog.Int(views); StdLog.Ln;
      StdLog.Ln
    END
  END Do;

END ObxCount0.
```

Листинг 5-23. Посчет числа литер наборов Latin-1, Unicode и вьюшек в тексте

В приведенном выше примере используется TextModels.Reader, который является бегунком для операции чтения. Каждый считыватель сохраняет свою текущую позицию в его базовом тексте; может быть несколько независимых считывателей для одного и того же текста одновременно. Считыватель текста имеет следующее определение:

```
TYPE
  Reader = POINTER TO ABSTRACT RECORD
    eot: BOOLEAN;
    attr: TextModels.Attributes;
    char: CHAR;
    view: Views.View;
    w, h: INTEGER;
    (rd: Reader) Base (): TextModels.Model, NEW, ABSTRACT;
    (rd: Reader) Pos (): INTEGER, NEW, ABSTRACT;
```

```
    (rd: Reader) SetPos (pos: INTEGER), NEW, ABSTRACT;
    (rd: Reader) Read, NEW, ABSTRACT;
    (rd: Reader) ReadChar (OUT ch: CHAR), NEW, ABSTRACT;
    (rd: Reader) ReadView (OUT v: Views.View), NEW, ABSTRACT;
    (rd: Reader) ReadRun (OUT attr: TextModels.Attributes), NEW, ABSTRACT;
    (rd: Reader) ReadPrev, NEW, ABSTRACT;
    (rd: Reader) ReadPrevChar (OUT ch: CHAR), NEW, ABSTRACT;
    (rd: Reader) ReadPrevView (OUT v: Views.View), NEW, ABSTRACT;
    (rd: Reader) ReadPrevRun (OUT attr: TextModels.Attributes), NEW, ABSTRACT
  END;
```

Листинг 5-24. Определение TextModels.Reader

Для данной текстовой модели text может быть создан считыватель вызовом rd := text.NewReader(NIL). Вновь размещенный считыватель позиционируется на начало текста, то есть на позицию rd.Pos() = 0. Позиция может быть изменена при необходимости с помощью вызова rd.SetPos(pos), где pos находится в диапазоне 0..text.Length(). Текстовая модель для считывателя может быть получена вызовом процедуры Base.

Чтение следующего элемента текста осуществляется процедурой считывателя Read, которая увеличивает текущую позицию считывателя, пока не достигнут конец текста. В этом случае устанавливается поле считывателя eot. После вызова Read прочитанная литера доступна из поля char. Если значение литеры меньше, чем 100X, литера попадает в диапазон Latin-1 (1-байтная литера), в противном случае это 2-байтная литера набора Unicode. Для удобства поддерживается вспомогательная процедура ReadChar, которая выполняет Read и возвращает содержимое char.

Что случится, если будет прочитана вьюшка? В большинстве случаев поле char выставится в предопределенное значение TextModels.viewcode (2X). Но в этом нет необходимости. Текстоориентированные вьюшки могут предоставлять несколько кодов литер с применением специального сообщения предпочтений (TextModels.Pref). Если нужно определить факт прочтения вьюшки, необходимо проверить поле view. Оно возвращает только что прочитанную вьюшку или NIL, если прочитана не вьюшка.

Понятно, почему для текстовой модели реализован оптимизированный доступ к вьюшкам. Для этой цели служит процедура ReadView, которая читает следующую вьюшку после текущей позиции, при необходимости пропуская все элементы, не являющиеся вьюшками.

После прочтения элемента (литеры или вьюшки) его текстовые атрибуты возвращаются в поле attr объекта чтения. Текстовые атрибуты состоят из атрибутов шрифта, цвета и вертикального смещения и описаны выше. По соображениям эффективности и удобства поддерживается вспомогательная процедура ReadRun, которая перемещает позицию считывателя до смены атрибутов в сравнении с текущей позицией. Фрагмент текста, имеющий одинаковые атрибуты может быть отрисован на экране одной процедурой (Port.Frame.DrawString), что значительно быстрее отрисовки литер одна за другой.

Обычно текст читается вперед от меньшей позиции к большей. Ассоциированные с объектом чтения процедуры ReadPrev, ReadPrevView и ReadPrevRun симметричны своим аналогам Read, ReadView и ReadRun, но чтение осуществляется в обратном порядке (назад). В частности, ReadPrevView применяется для текстовых вьюшек, так же называемых установщиками текста для поиска наиболее близкой вьюшки-линейки, чтобы определить, какие параметры текста необходимо выставить (этот поиск осуществляется во вспомогательной процедуре TextRulers.GetValidRuler).

При вызове одной из процедур чтения (вне зависимости от того, читают ли они вперед или назад), устанавливается значение поля eot объекта чтения. Если элемент может быть прочтен, eot выставляется в FALSE. Если ни один из элементов не может быть прочтен (чтение за концом или началом файла) eot выставляется в TRUE.

Обычно применяется следующий шаблон при работе с текстовыми считывателями. Условие может быть чем-то вроде rd.Pos() # end или ~rd.eot.

```
VAR rd: TextModels.Reader; start: INTEGER;
  BEGIN
    ... define start ...
    rd := text.NewReader(NIL);
    rd.SetPos(start);   (* необходимо только если старт # 0 *)
    rd.Read;
    WHILE condition DO
      ... олределить текстовй элемент обозначенный при помощи rd ...
      rd.Read
    END;
```

Листинг 5-25. Шаблон кода для TextModels.Reader

Следующий пример использует сканер текста. Сканеры читают текстовые отображатели; как и форматеры они определены в модуле TextMappers. Сканеры текстов подобны объектам чтения текстов за исключением того, что они работают с текстом как целым, а не с отдельными литерами. Литерные цепочки и числа — примеры символов Компонентного Паскаля, которые распознаются текстовыми сканерами.

Если сравнить определение TextMappers.Scanner с определением TextModels.Reader, можно заметить, что сканер тоже может вернуть литеру (поле char) и вьюшку вместе с размерами (поля view, w и h), что сканер имеет позицию (Pos, SetPos), но сканер не может читать назад. Процедуре Scan у сканера соответствует процедура Read у объекта чтения. Scan пропускает пробельные элементы (литеры пробела, возврат каретки), пока не будет достигнут конец текста, о чем сообщается значением TextMappers.eot в поле type, или пока не будет прочитан символ. Тип прочитанного символа (вьюшки) указывается значением поля type. Таким значением может быть одна из констант модуля TextMappers:

char, string, int, real, bool, set, view, tab, line, para, eot, invalid.

По умолчанию вьюшки и управляющие символы интерпретируются как пробелы и поэтому игнорируются. Применение дополнительного множества (opts, SetOpts) позволяет распознавать их как активные символы. Элемент множества TextMappers.returnViews указывает, что распознаются вьюшки, а TextMappers.returnCtrlChars указывает, что распознаются три управляющих символа: символы табуляции, возврата каретки и абзаца. Если очередной символ начинается как литерная цепочка Компонентного Паскаля или идентификатор, сканер пытается прочитать цепочку. В случае успеха type выставляется в string; которая так же может быть и короткой строкой, то есть содержать только литеры набора Latin-1. Опция TextMappers.returnQualIdents требует от сканера целиком различать квалифицировоанные идентификаторы как один символ, например, цепочку ThisMod.ThatObj. Длина цепочки возвращается в len.

Если оказывается возможным преобразовать символ в целое или дробное число, type выставляется в соответствующий тип, а так же выставляется поле int или real. Целые числа распознаются в различных системах исчисления, распознанная система исчисления возвращается в base (10 для десятичных чисел)

Специальные литеры !@#$ распознаются как тип TextMappers.char, которые могут быть так же короткими литерами, то есть литерами набора Latin-1.

По умолчанию логические значения и множества (типы BOOLEAN и SET соответственно) не интерпретируются кроме того случая, когда выставлены параметры TextMappers.interpretBool или TextMappers.interpretSets. Для множеств применяется синтаксис Компонентного Паскаля, а логические значения представлены как $TRUE и $FALSE.

Поле сканера start содержит позицию первой литеры, которая будет сейчас прочитана. Поля lines и paras содержат количество символов возврата каретки и абзаца с последнего обращения к процедуре SetPos (см. ниже). Процедура сканера Skip является вспомогательной и обеспечивает продвижение сканера до первой непустой литеры (как указано в множестве opts) и чтения его. Эта литера возвращается в параметре ch в процедуре Skip's.

Так как может быть несколько специализированных текстовых вьюшек, в дополнение к ним поддерживается TextMappers, текстовая модель не может создать объект текстового отображателя, подобно тому как она создает текстовые считыватели и записыватели. Вместо этого сканеры (так же как и форматтеры) в явном виде соединяются с текстом путем вызова процедуры ConnectTo. Она может быть вызвана повторно для соединения того же сканера с другой моделью.

```
Scanner = RECORD
   rider-: TextModels.Reader;
   opts-: SET;
   type: INTEGER;
   start, lines, paras: INTEGER;
   char: CHAR;
   int, base: INTEGER;
   real: REAL;
   bool: BOOLEAN;
   set: SET;
   len: INTEGER;
   string: TextMappers.String;
   view: Views.View;
   w, h: INTEGER;
   (VAR s: Scanner) ConnectTo (text: TextModels.Model), NEW;
   (VAR s: Scanner) Pos (): INTEGER, NEW;
   (VAR s: Scanner) SetPos (pos: INTEGER), NEW;
   (VAR s: Scanner) SetOpts (opts: SET), NEW;
   (VAR s: Scanner) Scan, NEW;
   (VAR s: Scanner) Skip (OUT ch: CHAR), NEW
END;
```

Листинг 5-26. Определение TextModels.Scanner

В следующем примере применяется текстовый сканер для подсчета числа целых, дробный чисел и строк, найденных в тексте.

```
MODULE ObxCount1;

   IMPORT TextModels, TextMappers, TextControllers, StdLog;

   PROCEDURE Do*;
   (** use TextCmds.SelectionGuard as guard for this command **)
      VAR c: TextControllers.Controller; from, to, ints, reals, strings: INTEGER;
         s: TextMappers.Scanner;
   BEGIN
      c := TextControllers.Focus();
      IF (c # NIL) & c.HasSelection() THEN
         c.GetSelection(from, to);   (* получить диапазон выделения; от < до *)
         s.ConnectTo(c.text);   (* соединить сканер с этой текстовой моделью *)
         s.SetPos(from);   (* установить считывателя на начало *)
         s.Scan;           (* прочитать первый символ *)
```

```
      ints := 0; reals := 0; strings := 0;   (* переменные для подсчета *)
      WHILE s.Pos() < to DO   (* прочесть все символы с начала выделенного фрагмента *)
        IF s.type = TextMappers.int THEN   (* символ - целое число *)
          INC(ints)
        ELSIF s.type = TextMappers.real THEN   (* символ - вещественное число *)
          INC(reals)
        ELSIF s.type = TextMappers.string THEN   (* символ - литерная цепочка или идентификатор *)
          INC(strings)
        END;
        s.Scan      (* прочитать следующий символ выделения *)
      END;
      StdLog.String("Integers: "); StdLog.Int(ints); StdLog.Ln;
      StdLog.String("Reals: "); StdLog.Int(reals); StdLog.Ln;
      StdLog.String("Strings: "); StdLog.Int(strings); StdLog.Ln;
      StdLog.Ln
    END
  END Do;

END ObxCount1.
```

Листинг 5-27. Подсчет целочисленных, дробных чисел и строк в тексте

Наиболее типичный шаблон кода для применения сканера выглядит подобно шаблону для считывателя. Условие может содержать что-то вроде: s.pos() < end или s.type # TextMappers.eot.

```
    VAR s: TextMappers.Scanner; start: INTEGER;
  BEGIN
    ... define start ...
    s.ConnectTo(text);
    s.SetPos(start);   (* необходимо, только если start # 0 *)
    s.Scan;
    WHILE condition DO
      IF s.type = TextMappers.int THEN
        ... consume s.int ...
      ELSIF s.type = TextMappers.real THEN
        ... consume s.real ...
      ELSIF ...
        ...
      END;
      s.Scan
    END;
```

Листинг 5-28. Шаблон кода для TextMappers.Scanner

5.4 Изменение текста

В предыдущих разделах было показано, как создать новый текст с помощью объектов записи или форматеров и как существующий текст может быть прочитан с помощью считывателей или сканеров. Многие текстовые команды сперва читают участок текста, проводят над ним какие-то вычисления, затем заменяют его на результат вычисления. Примерами могут служить команды по установке шрифта, цвета, переключения регистра для выделенного участка. Следующий пример команды читает текст выделения и проверяет на наличие строк. Найденные строки интерпретируются как имена и по этим

именам осуществляется поиск в телефонной базе данных из нашего примера. Затем имя заменяется на его телефонный номер.

```
MODULE ObxLookup0;

  IMPORT TextModels, TextMappers, TextControllers, ObxPhoneDB;

  PROCEDURE Do*;
  (** используйте TextCmds.SelectionGuard в качестве охранника команды **)
    VAR c: TextControllers.Controller; buf: TextModels.Model; from, to: INTEGER;
      s: TextMappers.Scanner; f: TextMappers.Formatter; number: ObxPhoneDB.String;
  BEGIN
    c := TextControllers.Focus();
    IF (c # NIL) & c.HasSelection() THEN
      c.GetSelection(from, to);
      s.ConnectTo(c.text);
      s.SetPos(from);
      s.Scan;
      IF s.type = TextMappers.string THEN
        buf := TextModels.CloneOf(c.text);
        f.ConnectTo(buf);
        ObxPhoneDB.LookupByName(s.string$, number);
        f.WriteString(number);
        from := s.start; to := s.Pos() - 1;   (* сканер уже считал символ до строки! *)
        c.text.Delete(from, to);                    (*удалить имя*)
        c.text.Insert(from, buf, 0, buf.Length());  (* переместить телефонный номер из
буфера в текст *)
        c.SetSelection(from, from + LEN(number$))   (* выбрать телефонный номер*)
      END
    END
  END Do;

END ObxLookup0.
```

Листинг 5-29. Замена имени на телефонный номер

Основная идея вышеприведенной команды состоит в том, что вспомогательный текстовый объект используется как буфер для операции. Выделенный текстовый фрагмент сканируется, отсканированная строка используется как ключ для поиска в базе данных телефона и возвращается телефонный номер в виде записи в текстовом буфере, затем первоначальное имя удаляется из текста, затем содержимое буфера помещается на место удаленного имени. В качестве заключительного шага вновь вставленный текст помечается как выделенный. Следует отметить, что процедура текстовой модели Delete укорачивает текст, в то время как процедура Insert удлиняет его. В частности, Insert не перезаписывает имеющийся текст. Insert перемещает текст, то есть удаляет его из источника и вставляет его в назначение.

Один из способов сделать упомянутую выше команду удобней заключается в замене выделения на введение литеры с клавиатуры. Затем можно написать имя и нажать клавиатурное сокращение для команды. Команда прочитает текст назад от позиции курсора ввода до начала имени и заменит его на подходящий для этого имени номер телефона. Подобный способ является простым способом реализации клавиатурных макросов.

Текстовые буферы вроде использованного выше особенно полезны для устранения мерцания экрана: поскольку текст в буфере не показывается, его можно собирать постепенно, без изменения изображения на экране. Когда работа с буфером закончена, его

содержимое можно переместить в видимый на экране текст-приемник. При этому нужно обновить экран только один раз, что делается быстро и исключает мерцание экрана.

Перемещение кусков текста из одной текстовой модели в другую может быть оптимизировано в случае, если обе текстовые модели имеют одну реализацию. В этом нет необходимости, так как различные реализации типа TextModels.Models могут взаимно существовать в одно время. Два вызова TextModels.dir.New() могут вернуть две правильные текстовые модели, имеющие различные реализации.

По этой причине в модуле TextModels экспортируется процедура Clone, которая возвращает пустую текстовую модель с типом, в точности соответствующим типу ее параметра. Фактически стандартная реализация текста даже реализует некоторые дополнительные возможности оптимизации при создании текстовой модели с помощью "клонирования", чем при размещении посредством TextModels.dir.New().

Теперь можно дать (слегка упрощенное) определение TextModels.Model:

```
TYPE
  Model = POINTER TO ABSTRACT RECORD (Containers.Model)
    (m: Model) NewWriter (old: TextModels.Writer): TextModels.Writer, NEW, ABSTRACT;
    (m: Model) NewReader (old: TextModels.Reader): TextModels.Reader, NEW, ABSTRACT;
    (m: Model) Length (): INTEGER, NEW, ABSTRACT;
    (m: Model) Insert (pos: INTEGER; m0: TextModels.Model;
                       beg0, end0: INTEGER), NEW, ABSTRACT;
    (m: Model) InsertCopy (pos: INTEGER; m0: TextModels.Model;
                       beg0, end0: INTEGER), NEW, ABSTRACT;
    (m: Model) Delete (beg, end: INTEGER), NEW, ABSTRACT;
    (m: Model) Append (m0: TextModels.Model), NEW, ABSTRACT;
    (m: Model) Replace (beg, end: INTEGER; m0: TextModels.Model;
                       beg0, end0: INTEGER), NEW, ABSTRACT;
    (m: Model) SetAttr (beg, end: INTEGER; attr: TextModels.Attributes), NEW, ABSTRACT;
  END;
```

Листинг 5-30. Определение TextModels.Model

Практически все процедуры уже были упомянуты. InsertCopy подобна Insert, но вместо перемещения копирует участок текста без изменения источника. Delete удаляет фрагмент текста.

Insert, InsertCopy и Delete являются элементарными текстовыми операциями. Append и Replace поддерживается для удобства и эффективности: они могут быть выражены в терминах элементарных операций.

```
dst.Append(src)  ≈  dst.Insert(dst.Length(), src, 0, src.Length())
```

То есть она перемещает все содержимое текста-источника в конец текста-назначения, таким образом оставляя текст источник пустым.

```
t.Replace(b, e, t1, b1, e1)  ≈  t.Delete(b, e); t.Insert(b, t1, b1, e1)
```

То есть она перезаписывает часть текста-назначения на некоторую часть текста-источника (который таким образом удаляется). Необходимо отметить, что тексты источника и назначения должны быть разными.

SetAttr позволяет выставить текстовые атрибуты для всего участка текста.

5.5 Текстовые сценарии

После успешного применения команды ObxLookup0.Do для участка текста можно попробовать нечто удивительное: можно отменить команду ("undo"), выполнив команду меню Edit->Undo! С помощью Edit->Redo можно отменить отмену операции.

Эта возможность удивительна, так как была реализована только простая процедура Do, но не код для ее отмены или повторного применения. Как случилось, что среда разработки все равно смогла отменить/повторить операцию?

Это оказалось возможным, потому что процедуры текстовой модели, которые изменяют статус модели, такие как Model.Delete и Model.Insert реализованы особым образом. Они не выполняют изменения напрямую. Вместо этого они создают специальный объект операции и регистрируют его в оболочке. Оболочка может вызвать соответствующую процедуру для операции для реализации фактической функциональности выполнения/отмены/повторения. Этот механизм был описан в главе 3.

Следующий пример показывает последовательность процедур, модифицирующих текст, заключенную в скобки BeginScript и EndScript для того, чтобы сделать последовательность целиком откатываемой. BeginScript возвращает объект сценария (script), он задает имя для всей составной операции как ввод.

```
MODULE ObxLookup1;

  IMPORT Stores, Models, TextModels, TextMappers, TextControllers, ObxPhoneDB;

  PROCEDURE Do*;
  (** используйте TextCmds.SelectionGuard в качестве охранника команды **)
    VAR c: TextControllers.Controller; buf: TextModels.Model; from, to: INTEGER;
      s: TextMappers.Scanner; f: TextMappers.Formatter; number: ObxPhoneDB.String;
      script: Stores.Operation;
  BEGIN
    c := TextControllers.Focus();
    IF (c # NIL) & c.HasSelection() THEN
      c.GetSelection(from, to);
      s.ConnectTo(c.text);
      s.SetPos(from);
      s.Scan;
      IF s.type = TextMappers.string THEN
        buf := TextModels.CloneOf(c.text);
        f.ConnectTo(buf);
        ObxPhoneDB.LookupByName(s.string$, number);
        f.WriteString(number);
        from := s.start; to := s.Pos() - 1;   (* сканер уже прочел символ до строки! *)
        Models.BeginScript(c.text, "#Obx:Lookup", script);
        c.text.Delete(from, to);                    (* удалить имя *)
        c.text.Insert(from, buf, 0, buf.Length());  (* переместить телефонный номер из
буфера в текст *)
        Models.EndScript(c.text, script);
        c.SetSelection(from, from + LEN(number$))   (* выбрать телефонный номер *)
      END
    END
  END Do;

END ObxLookup1.
```

Листинг 5-31. Пример сценария (составной операции)

В Models.BeginScript задается имя составной операции. Оно отображается с помощью возможностей отображения строк BlackBox, чтобы сделать это имя видимым пользователю (в команде меню undo/redo).

В модулях ObxLookup0 и ObxLookup1 применяются следующие последовательности опера-

торов для замены имени на номер:

```
c.text.Delete(from, to);
c.text.Insert(from, buf, 0, buf.Length());
```

Объединение этих двух операторов в одну составную операцию (как ожидает пользователь) приводит к необходимости заключения этих двух вызовов в операторные скобки BeginScript/EndScript. В данном специальном случае будет лучше, если применить:

```
c.text.Replace(from, to, buf, 0, buf.Length());
```

TextModels.Model.Replace — мощная процедура, для которой Insert, Delete и Append суть частные случаи (они предоставляются для удобства):

```
t.Delete(beg, end)   =   t.Replace(beg, end, t, 0, 0)
t.Insert(pos, t0, beg, end)   =   t.Replace(pos, pos, t0, beg, end)
t.Append(t0)   =   t.Replace(t.Length(), t.Length(), t0, 0, t0.Length())
```

Отметим, что InsertCopy не может быть выражено в терминах Replace:

```
t.InsertCopy(pos, t0, beg, end)   #   t.Replace(pos, pos, t0, beg, end)
```

Причина в том, что Replace, подобно Insert удаляет текст источника. Но процедура InsertCopy является операцией чистого копирования без модификации текста-источника (кроме случаев, когда текст-источник и текст-приемник идентичны и позиция копирования в тексте-приемнике лежит в копируемом диапазоне текста-источника).

5.6 Резюме

На этом обсуждение текстовой подсистемы BlackBox Component Builder заканчивается. Основной целью этого учебника является обзор связанного комплекса расширений подсистем, использование шаблонов проектирования из предыдущей главы, показ типичных примеров кода текстовой подсистемы.

Познание текстовой подсистемы не может быть завершено, поскольку остается много деталей, относящихся к документации разных моделей. Тем не менее, главная цель и типичные способы применения должны быть ясны. Так как в начале было изложено немного теории и применялись только простые примеры, базовая структура подсистемы составных контейнеров должна быть ясна уже сейчас.

Перед тем как дать список примеров из встроенной помощи, которые следует изучить, приведем список всех стандартных модулей текстовой подсистемы (самый нижний модуль, иерархически, самый первый) с указанием важных типов, экспортируемых из них:

module / type description

TextModels - абстрактная и реализация по умолчанию для текстовых моделей
- Attributes - атрибуты текстовых элементов (шрифт, цвет, вертикальное смещение)
- Context - связь между текстовой моделью и внедренной вьюшкой
- Directory - фабрика текстовых моделей
- Model - текстовый пользователь, фабрика для считывателей и записывателей
- Reader - бегунок для элементоориентированного чтения текста
- Writer - бегунок для элементоориентированной записи текста

TextMapper - абстрактная и реальная реализация текстовых мапперов для символов языка Компонентный Паскаль
- Formatter - преобразователь для посимвольной записи текста
- Scanner - преобразователь для посимвольного чтения текста

TextRulers - абстрактная и реализация по умолчанию для линеек, которые задают атрибуты абзаца
- Attributes - атрибуты текстовых линеек (табуляция, отступы, сетка и т.п.)

Directory	- фабрика для текстовых линеек
Ruler	- вьюшки линеек
Style	- модели линеек
TextSetters	- абстрактная и реализация по умолчанию для текстовых установщиков, обеспечивающих разрыв строки/страницы
Directory	- для текстовых установщиков
Reader	- текстовый отображатель, разделяющий текст по своим полям по текстовым установкам и определение их размера
Setter	- объект, реализующих алгоритм установки текстовых параметров
TextViews	- абстрактная и реализация по умолчанию для текстовых вьюшек
Directory	- фабрика текстовых вьюшек
View	- отображение текстовых моделей
TextControllers	- абстрактная и реализация по умолчанию для текстовых контроллеров
Controller	- контроллеры для текстовых вьюшек
Directory	- фабрика текстовых контроллеров
TextCmds	- набор наиболее важных команд работы с текстами

Таблица 5-32. - таблица модулей и наиболее важных типов для текстовых подсистем

ObxHello0	- пишет "Hello World" в рабочий журнал
ObxHello1	- пишет "Hello World" в новый текст
ObxOpen0	- открывает текстовый документ из файла с применением стандартного диалога открытия файла
ObxOpen1	- Изменение текстового документа в файле с применением стандартных диалогов открытия и сохранения файлов
ObxCaps	- приводит строку к верхнему регистру с применением составных операций
ObxDbd	- управление отсортированным списком записей
ObxMMerge	- слияние для почты шаблона и базы данных
ObxLinks	- создает текст с гиперссылками
ObxAscii	- обычный текстовый файл

Таблица 5-33. Дополнительные встроенные примеры по текстовой подсистеме

Часть III: Построение объектов изображения (вьюшек)

В части I учебника по BlackBox дается введение в схемы проектирования, применяемые в BlackBox. Знание этих схем облегчает понимание и запоминание частных решений в различных модулях BlackBox.

Часть II учебника по BlackBox демонстрирует применение наиболее важных библиотек компонентов: элементы управления, формы, текстовые компоненты.

Часть III учебника по BlackBox демонстрирует разработку новых вьюшек на серии усложняющихся примеров от самых простых к более сложным.

6 Построение объектов изображения (вьюшек)

Объекты избражения (вьюшки) — центральная абстракция системы Блэкбокс. Всё вращается вокруг них: большинство команд выполняется над вьюшками, окна показывают вьюшки, вьюшки могут быть загружены в память и выгружены, с помощью вьюшек проводится взаимодействие с пользователем, и вьюшки могут быть внедрены в другие вьюшки.

6.1 Введение

Учебник программирования вьюшек состоит из четырех разделов, каждый из которых представляет особый аспект программирования вьюшек. В разделе "обработка сообщений" ("message handling") (главы с 6.2 по 6.5) разъясняется поведение вьюшек, определенное для обработки сообщений вьюшек, то есть какие могут быть ответы, сообщения элементов управления и свойства потока взаимодействия вьюшек с окружающей средой и как вьюшки реагируют на пользовательский ввод. Во втором разделе (6.7) объясняются аспекты разделения моделей и вьюшек, благодаря чему возможно многооконное редактирование вьюшек. В следующем разделе (6.7) показывается, как могут быть использованы объекты операций для возможностей отката/повтора BlackBox Component Framework при изменении содержимого вьюшек. В последнем разделе (6.8) объясняется особая структура всех расширяемых модулей BlackBox Component Framework, а именно разделение интерфейса и реализации вьюшек. Фабричный объект (directory object) является ключевой схемой проектирования.

6.2 Обработка сообщений

В этом разделе представлена последовательность реализаций вьюшки, которая показывают прямоугольную закрашенную область, со все более возрастающими возможностями. В частности, вьюшка обрабатывает сообщения Preferences, Controller и Properties.

Каждая вьюшка является подтипом абстрактного типа Views.View. Конкретное расширение должно как минимум реализовывать абстрактную процедуру Views.Restore, которая отрисовывает содержимое отображения. Первый пример нашего отображения просто рисует красные прямоугольник. Эта версия представляет собой простейшую из возможных реализацию отображения. Главными компонентами являются определение расширения от Views.View, реализация процедуры Views.Restore, которая отрисовывает содержимое вьюшек и управляющая процедура для размещения и инициализации вьюшки:

```
MODULE ObxViews0;

   IMPORT Views, Ports;
```

```
  TYPE View = POINTER TO RECORD (Views.View) END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
  END Restore;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); Views.Deposit(v)
  END Deposit;

END ObxViews0.
```

Для выполнения этой программы запустите следующую команду:

❗ "ObxViews0.Deposit; StdCmds.Open"

Результат этой команды представлен на рисунке 6-1.

Рисунок 6-1. Результат "ObxViews0.Deposit; StdCmds.Open"

Эта простая программа уже полностью функциональна. Процедура ObxViews0.Deposit заносит вновь размещенная вьюшка в системную очередь. Команда StdCmds.Open, приведенная выше в командной строке, извлекает вьюшку из очереди и открывает его в новом окне. Следует отметить, что последовательность команд должна быть заключена в кавычки, а для одиночной команды кавычки можно опустить.

Вместо того, чтобы открывать вьюшку в окне, ее можно поместить в контейнер, владеющий фокусом ввода (то есть в текст или форму):

❗ "ObxViews0.Deposit; StdCmds.PasteView" **><***перед выполнением команды поместите*

Как и StdCmds.Open, команда StdCmds.PasteView извлекает вьюшку из очереди, но вставляет ее во вьюшку в фокусе ввода. Вышеприведенная последовательность команд может быть использована, например, в меню.

Каждый документ, содержащий вьюшку ObxViews0 может быть сохранен в файл, который затем может быть открыт вновь с помощью стандартной команды меню Open... Простой красный прямоугольник должен быть отображен правильно во вновь открытом документе, что обеспечивается доступностью файла кода модуля ObxViews0.

Отображение, которое было открыто в собственном окне, так же могло быть сохранено в файле. При повторном открытии документ состоит только из этой единственной вьюшки.

Каждая вьюшка осуществляет операции вывода и обработки мыши/клавиатуры через объект Views.Frame. В приведенном выше примере ObxViews0.View.Restore использует соб-

ственную рамку как параметр f для отрисовки строки. Рамка может считаться объектом отображателя, в данном случае как ввода, так и вывода одновременно (здесь будет интересно уточнить о бегунках и носителях для этого отображателя, но оба они определены в модуле Ports). Рамка реализует в себе преобразование координат и возможности обрезки.

В случае, если вьюшка отрисовывается на экране, рамка является отображателем для порта экрана. Если вьюшка распечатывается, рамка является вьюшкой для порта принтера. С точки зрения отображения эта разница не существенна. Так, не требуется особого программного кода для выполнения печати.

Вьюшки могут быть скопированы и вставлены в контейнеры, такие как текстовые вьюшки или вьюшки в виде форм. Дополнительно может быть изменен размер вьюшки, если ее выбрать и изменить размер. Вся эта функциональность поддерживается каркасом (BlackBox Component Framework) и не требует дополнительного программирования. Если поведение по умолчанию неудобно, его можно изменить. В дальнейшем будет показано, как это сделать.

6.3 Сообщения свойств по умолчанию

Следует отметить, что размер вновь открытой вьюшки может быть различным. Тем не менее, перед открытием вьюшки оболочка посылает вьюшке сообщение с предполагаемыми размерами. Вьюшка может приспособить предлагаемые размеры для собственных нужд. Для этого сообщение типа Properties.SizePrep должно быть обработано в процедуре вьюшки HandlePropMsg. Перед первой отрисовкой вьюшки предполагаемые размер для ширины и высоты вьюшки задан в Views.undefined. В следующей версии нашей вьюшки рисунок отрисовывается с шириной 2 и высотой 1 см. Внесенные в сравнении с предыдущей версией изменения отмечены жирным шрифтом.

```
MODULE ObxViews1;

   IMPORT Views, Ports, Properties;

   TYPE View = POINTER TO RECORD (Views.View) END;

   PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
   BEGIN
      f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
   END Restore;

   PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
   BEGIN
      WITH msg: Properties.SizePref DO
         IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
            msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
         END
      ELSE (* игнорировать другие сообщения *)
      END
   END HandlePropMsg;

   PROCEDURE Deposit*;
      VAR v: View;
   BEGIN
      NEW(v); Views.Deposit(v)
   END Deposit;

END ObxViews1.
```

"ObxViews1.Deposit; StdCmds.Open"

Теперь каждое вновь созданная вьюшка принимает желаемые предопределенные по умолчанию размеры. Это возможно, потому что вьюшка и контейнер взаимодействуют, в данном случае, посредством сообщения Properties.SizePref. Предполагается, что контейнер посылает это сообщение каждый раз, когда он хочет изменить размеры вьюшки, а затем принимает возвращенные данные. Тем не менее для понимания каркаса BlackBox необходимо знать, что контейнер с правильным поведением (well-begaved) должен следовать этому протоколу, но данное следование не обязательно. Именно поэтому подобное сообщение называется "предпочтением". Оно описывает лишь пожелания вьюшки, но охватывающий контейнер может навязать следование специальным правилам для отображения. Контейнер всегда имеет последнее слово в подобных переговорах. Например, контейнер не может позволить размерам вьюшки превысить его собственные. Стандартные контейнеры: документ, текст и формы — являются контейнерами с корректным поведением и поддерживают все предпочтения, определенные в модуле Properties. Специализированные контейнеры, такие как тексты, могут определить дополнительные предпочтения, специфические для типа содержимого. Важно отметить, что вьюшки могут игнорировать все предпочтения, которые они не знают или о которых не беспокоятся.

Properties.SizePref позволяет ограничить возможные размеры по ширине и высоте вьюшки. Например, можно задать принудительно минимальный и максимальный размер или зафиксировать высоту и ширину вьюшки или определить ограничения на их ширину и высоту. Процедуры Properties.ProportionalConstraint и Properties.GridConstraint являются полезным стандартом реализации ограничений. Следующая версия реализации вьюшки определяет, что ширина прямоугольника всегда в два раза больше его высоты. Дополнительно определяются минимальные и максимальные размеры для высоты (тем самым, и для ширины). Если размер вьюшки изменяется с помощью обработчика изменения размеров, ограничения не нарушаются. Это можно проверить.

```
MODULE ObxViews2;

  IMPORT Views, Ports, Properties;

  TYPE View = POINTER TO RECORD (Views.View) END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
  END Restore;

  PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
        msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
      ELSE
        Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                        msg.w, msg.h);
        IF msg.h < min THEN
          msg.h := min; msg.w := 2 * min
        ELSIF msg.h > max THEN
          msg.h := max; msg.w := 2 * max
        END
      END
    ELSE   (* игнорировать другие сообщения *)
```

```
    END
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); Views.Deposit(v)
  END Deposit;

END ObxViews2.
```

❗ "ObxViews2.Deposit; StdCmds.Open"

Мы рассмотрим в этом учебнике еще два других предпочтения. Первое - Properties.ResizePref.

```
  TYPE
    ResizePref = RECORD (Properties.Preference)
      fixed,
      horFitToPage, verFitToPage,
      horFitToWin, verFitToWin: BOOLEAN   (* OUT *)
    END;
```

Получатель этого сообщения может показать, что он не желает изменять размеры, выставив fixed в TRUE. Как следствие вьюшка не показывает метки для изменения размера при выделении, поэтому нельзя изменить размеры интерактивно. Тем не менее, сообщение Properties.SizePref вьюшке все еще посылается и начальный размер все еще необходимо определить.

Если вьюшка является корневой, то есть самая дальня от центра вьюшка в документе или окне (то есть открытая с помощью StdCmds.Open), размер окна, размер вьюшки или оба сразу могут быть изменены.

Однако иногда бывает удобно автоматически приводить размер вьюшки в соответствие с меняющимся размером окна, то есть помочь текстам занимать как можно больше места на экране. Для других вьюшек было бы предпочтительней, если их размер определялся бы не окном, а содержимым или настройками страницы внедренного документа. Вьюшка может использовать эти возможности выставив значения horFitToWin, verFitToWin, horFitToPage и verFitToPage для сообщения Properties.SizePref. Эти значения не имеют силы, если вьюшка не является корневой, то есть является внедренной в документ.

Автоматическое приведение размеров вьюшки по текущему размеру окна может быть достигнуто путем выставления полей horFitToWin и verFitToWin. Размер вьюшки ограничен окном, то есть пользователь может изменить размер напрямую, изменяя размеры окна.

Следует отметить, что если размеры вьюшки ограничены размерами окна (а так же путем выставления hotFitToWin или verFitToWin), то сообщение Properties.SizePref для вьюшки не посылается, то есть ограничения, заданные в Properties.SizePref, больше не применяются. Дополнительно для вьюшки не показываются метки изменения размеров, вне зависимости от состяния fixed.

Установкой полей horFitToPage или verFitToPage ширина или высота отображения может быть ограничена фактическим размером бумаги в принтере. Ширина текстового отображения обычно ограничена шириной листа бумаги и может быть изменена путем изменения установок размеров бумаги, которое поддерживается средствами операционной системы. Следующий пример принудительно выставляет размер корневого отображения по размерам окна:

```
MODULE ObxViews3;

  IMPORT Views, Ports, Properties;

  TYPE View = POINTER TO RECORD (Views.View) END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
  END Restore;

  PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
         msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
      ELSE
        Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                        msg.w, msg.h);
        IF msg.h < min THEN
          msg.h := min; msg.w := 2 * min
        ELSIF msg.h > max THEN
          msg.h := max; msg.w := 2 * max
        END
      END
    | msg: Properties.ResizePref DO
      msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
    ELSE
    END
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); Views.Deposit(v)
  END Deposit;

END ObxViews3.
```

❗ "ObxViews3.Deposit; StdCmds.Open"

Теперь рассмотрим Properties.FocusPref.

```
  TYPE
   FocusPref = RECORD (Properties.Preference)
     atLocation: BOOLEAN;   (* IN *)
     x, y: INTEGER;   (* IN *)
     hotFocus, setFocus, selectOnFocus: BOOLEAN   (* OUT *)
   END;
```

При попытке активации внедренной вьюшки по щелчку на нем, вьюшке посылается сообщение Properties.FocusPref. Если на сообщение не было ответа, вьюшка выделяется целиком (так же называется singleton). Такое поведение вьюшки реализовано в вышеприведенном примере. В подтверждение, разместите курсор на следующей строке и щелкните на командире, затем на вставленном отображении:

❗ "ObxViews3.Deposit; StdCmds.PasteView" >< *перед выполнением команды установите курсор здесь*

Такое поведение применено для пассивоной вьюшки. Тем не менее, если содержимое вьюшки можно редактировать или имеются команды меню, выполняющие действия над вьюшкой, пользователь должен иметь возможность получить фокус ввода на вьюшку. Фокус ввода - это место, куда осуществляется ввод данных с клавиатуры, где показывается текущее выделение и положение курсора (или какая-то другая подобная метка) и где выполняются некоторые команды меню. Фактически меню доступны, только если вьюшка владеет фокусом ввода, и меню становятся недоступными, когда вьюшка теряет фокус ввода (ниже об этом будет сказано подробнее). Корневая вьюшка всегда имеет фокус ввода, если фокус ввода имеет окно документа (то есть верхнее окно).

В случае, если внедренная вьюшка может получать фокус ввода, то оно должно отвечать на сообщение Properties.FocusPref. Отображение может определить, захватывается ли фокус ввода или фокус переводится только на время, в течение которого нажата кнопка мыши. Последняя ситуация называется "горячий фокус" (hot focus). Типичный пример горячего фокуса - кнопка. Для того, чтобы вьюшка приобрела свойство горячего фокуса, необходимо выставить флаг hotFocus. В этом случае фокус ввода переводится дальше сразу после отпускания кнопки мыши. Когда требуется захватить фокус ввода, необходимо выставить флаг setFocus. setFocus следует выставлять для всех редакторов, чтобы контекстнозависимые команды меню могли применяться к вьюшке. В дополнение к флагу setFocus для вьюшки может быть выставлен флаг selectOnFocus для того, чтобы при наведении с помощью клавиатуры содержимое вьюшки выделялось. Текстовые поля ввода являются первым примером подобного поведения: содержимое элемента, только что получившего фокус - выделено, если пользователь попал не него не с помощью мыши, а с применением клавиатуры.

Если пользователь щелкнет на вьюшке без фокуса ввода, то каркас выставляет флаг alLocation перед посылкой сообщения. Отображение-получатель может решить, каким образом сфокусироваться в зависимости от того, где щелкнул пользователь. Позиция мыши передается вьюшке в полях x и y предпочтения фокуса (focus preference). Текстовые линейки - примеры вьюшек, которые фокусируются в зависимости от положения мыши. Если кликаются иконки в области ниже шкалы линейки, то линейка не получает фокус. В противном случае - получает. Это можно попробовать прямо сейчас на линейке. Показать/скрыть линейки можно с помощью команды меню Text->ShowMarks.

Отображение получают фокус не только после щелчка мыши. Например вьюшки могут быть выбраны с помощью клавиш управления курсора. В случае, если вьюшка фокусируется без участия мыши, каркас выставляет поле atLocation в FALSE.

В следующей версии примера будет обрабатываться сообщение FocusPref и выставляться флаг setFocus. Это достигается добавлением в процедуру HandlePropMsg предыдущего примера следующего кода:

```
| msg: Properties.FocusPref DO
  msg.setFocus := TRUE
```

Если внедренная вьюшка получает фокус, отрисовывается рамка вьюшки с соответствующими отметками, активируются специфические для вьюшки пункты меню, а остальные дезактивируются.

6.4 Управляющие сообщения

Другие сообщения, которые могут быть обработаны процедурой HandlerPropMsg будут рассмотрены позже. Дальше будет рассмотрено, как вьюшки реагируют на пользовательский ввод, то есть на щелчки мыши внутри вьюшки или команды, вызываемые из меню. Для простых вьюшек такое поведение реализуется процедурой Views.HandleCtrlMsg. Так же возможно определить отдельный объект, называемый контроллером, которые реализует

поведение вьюшек при взаимодействии с пользователем. Программирование объектов контроллера здесь не рассматривается, так как применение контроллеров - только рекомендация для большинства сложных вьюшек.

Обработчки Views.HandleCtrlMsg отвечает на управляющие сообщения (controller messages), посылаемые вьюшкам. Каждая вьюшка в цепочке обработки фокуса самостоятельно решает завершить ли обработку сообщения, то есть владеет ли она фокусом, или передать сообщение к другой внедренной вьюшке. Важно отметить, что все сообщения контроллеров, которые не подходят к типу вьюшки, игнорируются.

Чтобы можно было выполнять редактирование во вьюшке, владеющей фокусом ввода, каркас должен каким-то образом разрешить эти операции. Щелчки мыши и нажатия клавиш могут быть всегда. Тем не менее, внешний вид меню зависит от решения, принятого непосредственно вьюшкой. Для этой цели вьюшке посылается сообщение Controllers.PollOpsMsg. В зависимости от текущего статуса (например, текущего выделения) и содержимого буфера обмена, вьюшка, владеющая фокусом ввода, может информировать каркас о том, какие операции редактирования сейчас возможны. Доступные операции содержатся в множестве элементов: {Controllers.cut, Controllers.copy, Controllers.paste, Controllers.pasteView}.

```
TYPE
  PollOpsMsg = RECORD (Controllers.Message)
    type: Stores.TypeName;   (* OUT *)
    pasteType: Stores.TypeName;   (* IN *)
    singleton: Views.View;   (* OUT *)
    selectable: BOOLEAN;   (* OUT *)
    valid: SET   (* OUT *)
  END;
```

Множество доступных операций редактирования возвращается в поле valid, где valid среди {Controllers.cut, Controllers.copy, Controllers.paste}. В зависимости от набора доступных операций становятся доступными или недоступными соответствующие пункты меню Edit, то есть Cut, Copy, Paste и PasteObject...(Windows)/Paste as Part (MacOS). Поле pasteType содержит конкретный тип вьюшки, копия которой будет вставлена, если произойдет операция вставки. В зависимости от значения поля вьюшка может решить разрешить или запретить операцию вставки. PollOpsMsg посылается, когда пользователь щелкнул на меню. Единственное его назначение - разрешить или запретить элементы меню, то есть осуществление обратной связи с пользователем.

Если вьюшка поддерживает выделение содержимого, то selectable должно быть установлено в TRUE. Как следствие станет доступной команда меню Select All. Этот флаг следует установить вне зависимоси от того есть или нет текущее выделение.

В поле type передается имя типа. Это имя является контекстом для вьюшки, владеющей фокусом ввода. Этот контекст применяется для определения соответствующего меню для вьюшки, владеющей фокусом ввода. По соглашению вьюшка задает в поле type свой базовый тип интерфейса, то есть "ObxViews4.View". Меню с отметкой активности для "ObxViews4.View" будет показыватся для вьюшки, владеющей фокусом. Имя типа применяется для облегчения поддержки уникальности и избежания конфликта имен. Оболочка никак не интерпретирует имена.

Поле singleton применяется для подчеркивания роли вьюшки-контейнера. Оно обозначает контейнер текущего выделенной вьюшки, если выделение состоит целиком из внедренной вьюшки.

В следующем примере вьюшка может <захватывать?>получать фокус ввода, поддерживает операцию вставки и может информировать оболочку о возможности выделения содержимого. Вследствие этого в меню Edit доступны пункты меню Paste и Select All. Дополнительно определяется контекст "Obx.Tutorial". Поэтому следующее меню появляется

всегда, когда вьюшка выделена; обеспечивается, если меню установлено.

Следует отметить, что имя контекста, в нашем случае "Obx.Tutorial", для удобства начинается с имени подсистемы или модуля с последующей точкой, в нашем случае "Obx.". Такое именование настоятельно рекомендуется в целях обеспечения глобальной уникальности имен. Часто это имя - наименование типа вьюшки, например: "TextViews.View".

Меню так же может быть объявлено в тексте глобального меню System/Rsrc/Menus или, более характерно, в собственном файле меню подсистемы, в данном случае Obx/Rsrc/Menus. Для того, чтобы меню Obx было видно в главном меню, в его текст необходимо добавить выражение INCLUDE "Obx".

```
MENU "New" ("Obx.Tutorial")
  "Beep"   ""   "Dialog.Beep"   ""
END
```

```
MODULE ObxViews4;

  IMPORT Views, Ports, Properties, Controllers;

  TYPE View = POINTER TO RECORD (Views.View) END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawRect(l, t, r, b, Ports.fill, Ports.red)
  END Restore;

  PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
         msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
      ELSE
        Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                      msg.w, msg.h);
        IF msg.h < min THEN
          msg.h := min; msg.w := 2 * min
        ELSIF msg.h > max THEN
          msg.h := max; msg.w := 2 * max
        END
      END
    | msg: Properties.ResizePref DO
      msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE   (* ignore other messages *)
    END
  END HandlePropMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                    VAR msg: Controllers.Message; VAR focus: Views.View);
  BEGIN
    WITH msg: Controllers.PollOpsMsg DO
      msg.valid := {Controllers.paste}; msg.selectable := TRUE;
      msg.type := "Obx.Tutorial"
```

```
    ELSE
    END
  END HandleCtrlMsg;

  stdTables

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); Views.Deposit(v)
  END Deposit;

END ObxViews4.
```

❗ "ObxViews4.Deposit; StdCmds.PasteView" **><** *перед выполнением коменды установите курсор здесь*

Теперь рассмотрим, каким образом вьюшка может отреагировать на сообщение о редактировании. В частности имеются Controllers.EditMsg для операций редактирования, таких как вырезать, скопировать, вставить и Controllers.TrackMsg для щелчка мыши.

При любом нажатии клавиши во вьюшке или при вырезании, копировании или вставке Controllers.EditMsg посылается вьюшке, владеющей фокусом ввода. Операции вырезания, копирования или вставки могут так же генерироваться средой окружения (через меню), если они были объявлены в Contollers.PollOpsMsg.

```
  TYPE
    EditMsg = RECORD (Controllers.RequestMessage)
      op: INTEGER;   (* IN *)
      modifiers: SET;   (* IN *)
      char: CHAR;   (* IN *)
      view: Views.View;   (* IN для вставки, OUT для вырезания и копирования *)
      w, h: INTEGER;   (* IN для вставки, OUT для вырезания и копирования *)
      isSingle: BOOLEAN;   (* IN для вставки, OUT для вырезания и копирования *)
      clipboard: BOOLEAN   (* IN *)
    END;
```

Поле op определяет какого типа операция будет произведена. Если op = Controllers.cut, то для вьюшки, владеющей фокусом ввода, будет сгенерировано копирование. Содержимым новой вьюшки будет копия выделения в вьюшке, владеющем фокусом ввода. Новая вьюшка присваивается полю view. В дополнение, выделение удаляется из вьюшки, владеющей фокусом ввода.

Есть один специальный случай: если выделение состоит только из одиночной вьюшки (синглетон - a singleton), то в поле view копируется сам синглетон, но не копия контейнера синглетона. В этом случае isSingle выставляется в TRUE.

За исключением операции удаления выделенного фрагмента, при значении op = Controllers.copy выполняются те же действия.

Если нажата клавиша на клавиатуре, поле op получает значение Controllers.pasteChar. Символ, который будет вставлен, хранится в поле char. Множество modifiers сообщает о нажатии вспомогательных клавиш. В последнем случае поле char должно интерпретироваться как управляющий символ.

Операция вставки (paste) является обратной к операции копирования (copy): копия

содержимого вьюшки, находящаяся в поле view, перемещается в состав вьюшки, владеющей фокусом ввода. Эта операция обозначается op = Controllers.paste. Вьюшка-приемник должно знать тип вьюшки, чье содержимое будет вставлено и должно решить каким образом содержимое вставляемой вьюшки будет размещено внутри себя. Например, текстовая вьюшка должна поддерживать копирование только из текстовых вьюшек. Если поле isSingle имеет значение TRUE или если содержимое вьюшки не может быть вставлено, потому что оно принадлежит неизвестному типу, то должна быть вставлена копия самой вьюшки (но не содержимого). Конечно, это возможно только для универсального контейнера, который позволяет внедрять вьюшки.

Когда вьюшка вставляется целиком, то предопределенные ширина и высота передаются в полях w и h. Эти значения могут использоваться как ориентир для вьюшки-приемника. Если они не подходят, могут использоваться другие.

При любом нажатии кнопок мыши во вьюшке, обладающей фокусом ввода, вьюшке посылается сообщение Controllers.TrackMsg. Координаты находятся в полях x и y (базового типа Controllers.CursorMessage). Множество modifiers фиксирует нажатие модифицирующих клавиш и факт двойного щелчка. modifiers in {Controllers.doubleClick, Controllers.extend, Controllers.Modify}. Если вьюшке необходимо отразить обратную связь при перемещении мыши, необходимо регистрировать позицию мыши в цикле, вызывая процедуру Input рамки (frame), в котором перемещается мышь. Процедура Input так же возвращает информацию о том, где была отпущена мышь. Любая обратная связь должна быть напрямую отражена в окне, в котором была нажата кнопка мыши.

После цикла обратной связи возможные изменения содержимого вьюшки должны быть распространены на все окна, показывающие эту вьюшку. Необходимо помнить, что один документ, а, следовательно, все его внедренные вьюшки, могут показываться в нескольких окнах. Если внедренная вьюшка видно в трех окнах (windows), значит существует три рамки (frames), показывающие одну вьюшку. Процедура Views.Update приводит к обновлению вьюшки во всех видимых фреймах.

В качестве примера расширим возможности тестовой вьюшки маркером в виде креста с возможностью его перемещения. Следует отметить, что если вьюшка содержит экземпляры переменных (Instance variables), необходимо реализовать процедуры CopyFromSimpleView, Internalize и Externalize для корректной работы. В следующих разделах учебника будет рассказано назначение этих сообщений. В представленном примере вьюшки маркер может передвигаться мышью или клавишами перемещения курсора. При нажатии на такую клавишу вьюшка, имеющая фокус ввода, информируется об этом с помощью сообщения EditMsg с параметром op = pasteChar.

Время от времени вьюшке, владеющей фокусом ввода, посылается сообщение Controllers.PollCursorMsg. Отображение может выставить форму курсора в зависимости от координат мыши. Полю cursor могут быть присвоены значения из набора: {Port.arrowCursor, Ports.textCursor, Ports.graphicsCursor, Ports.tableCursor, Ports.bitmapCursor}. Конкретная форма курсора зависит от особенностей операционной системы. В приведенном примере выставляется графический курсор.

```
MODULE ObxViews5;

  IMPORT Views, Ports, Properties, Controllers;

  TYPE
    View = POINTER TO RECORD (Views.View)
      x, y: INTEGER
    END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
```

```
    f.DrawRect(l, t, r, b, Ports.fill, Ports.red);
    f.DrawLine(v.x, t, v.x, b, 0, Ports.white); f.DrawLine(l, v.y, r, v.y,
                                    0, Ports.white)
  END Restore;

  PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
         msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
      ELSE
        Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                    msg.w, msg.h);
        IF msg.h < min THEN msg.h := min; msg.w := 2 * min
        ELSIF msg.h > max THEN msg.h := max; msg.w := 2 * max
        END
      END
    | msg: Properties.ResizePref DO
      msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
  END HandlePropMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                        VAR msg: Controllers.Message;
                        VAR focus: Views.View);
    VAR x, y, w, h: INTEGER; m: SET; isDown: BOOLEAN;
  BEGIN
    WITH msg: Controllers.PollOpsMsg DO
      msg.valid := {Controllers.paste}; msg.selectable := TRUE;
      msg.type := "Obx.Tutorial"
    | msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN    (* клавиши курсора *)
        IF msg.char = 1DX THEN INC(v.x, Ports.mm)
        ELSIF msg.char = 1CX THEN DEC(v.x, Ports.mm)
        ELSIF msg.char  = 1EX THEN DEC(v.y, Ports.mm)
        ELSIF msg.char  = 1FX THEN INC(v.y, Ports.mm)
        END;
        Views.Update(v, Views.keepFrames)
      END
    | msg: Controllers.TrackMsg DO
      v.x := msg.x; v.y := msg.y;
      v.context.GetSize(w, h); v.Restore(f, 0, 0, w, h);
      REPEAT
        f.Input(x, y, m, isDown);
        IF (x # v.x) OR (y # v.y) THEN
          v.x := x; v.y := y; v.Restore(f, 0, 0, w, h)
        END
      UNTIL ~isDown;
      Views.Update(v, Views.keepFrames)
    | msg: Controllers.PollCursorMsg DO
      msg.cursor := Ports.graphicsCursor
    ELSE
```

```
    END
  END HandleCtrlMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); v.x := 0; v.y := 0; Views.Deposit(v)
  END Deposit;

END ObxViews5.
```

❗ "ObxViews5.Deposit; StdCmds.Open"

Существует много других управляющих сообщений, которые могут быть посланы вьюшке, но в данной книге ограничимся учебными целями. Имеются сообщения, реализующие общий механизм прокрутки BlackBox Component Framework (Controllers.PollSelectionMsg, Contollers.ScrollMsg, Controllers.PageMsg), сообщения, реализующие механизм drag&drop (Controllers.PollDropMsg, Controllers.DropMsg, Controllers.TransferMessage) и сообщения для контроля выделения (Controllers.SelectMsg). За справкой по этим и другим управляющим сообщениям следует обращаться к встроенной документации модуля Controllers.

6.5 Свойства

Этот раздел завершает расширение сквозного примера вьюшки: теперь цвет вьюшки можно изменить с помощью пункта меню Attr[ibutes]. Общий механизм получения или выставления свойств вьюшки через среду окружения реализован в виде свойств (properties). Отображение может обладать информацией об атрибутах, таких как: шрифт, цвет, размер и еще несколько других атрибутов. Свойства (properties) выставляются с помощью процедуры Properties.SetMsg и проверяются с помощью процедуры Properties.PollMsg. Свойства представляют собой сообщения, связанные в список. Когда вьюшка получает Properties.SetMsg, оно проходит список свойств и приводит в соответствие те из них, о которых у него есть информация. Properties.GetMsg посылается вьюшке, если необходимо затребовать значение свойства. Отображение возвращает все значения свойств, о которых есть информация. Свойство может быть вставлено в список свойств сообщения с помощью процедуры Properties.Insert.

Каждое свойство описывает до 32 атрибутов. Множество known определяет, какие атрибуты известны вьюшке. Множество так же может определить, какие атрибуты являются атрибутами "только для чтения" с помощью множества readOnly. Множество valid указывает, какие из атрибутов имеют выставленное значение. Например, если в тексте выделено несколько литер разного размера, атрибут size будет поддерживаться для вьюшки, но его текущее значение не будет определено. Так как выделение не является однородным, то не может быть одного корректного значения.

```
  TYPE
    Property = POINTER TO ABSTRACT RECORD
      next-: Properties.Property;
      known, readOnly, valid: SET;   (* valid and readOnly are subsets of known *)
      (p: Property) IntersectWith (q: Properties.Property; OUT equal: BOOLEAN), NEW,
ABSTRACT
    END;
```

Имеется специальное свойство Properties.StdProp. Это свойство заключает в себе атрибуты шрифта так же как цвет и известно для среды окружения BlackBox. Поддерживаются атрибуты: Properties.color, Properties.typeface, Properties.size, Properties.style, Properties.weight. В этих полях записи содержатся соответствующие значения.

```
  TYPE
    StdProp = POINTER TO RECORD (Properties.Property)
      color: Dialog.Color;
      typeface: Fonts.Typeface;
      size: INTEGER;
      style: RECORD
        val, mask: SET
      END;
      weight: INTEGER
    END;
```

Последняя версия вьюшки поддерживает только атрибут цвета для стандартного свойства. При получении сообщения Properties.PollMsg возвращается объект типа Properties.StdProp, в котором установлено только поле для цвета и только атрибут цвета установлена как поддерживаемый и корректный. При Properties.SetMsg в списке свойств отыскивается стандартное свойство, у которого корректно поле цвета. После того, как цвет вьюшки изменен, его следует обновить во всех рамках.

```
MODULE ObxViews6;

  IMPORT Views, Ports, Properties, Controllers;

  TYPE
    View = POINTER TO RECORD (Views.View)
      x, y: INTEGER;
      c: Ports.Color
    END;

  PROCEDURE (v: View)  Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawRect(l, t, r, b, Ports.fill, v.c);
    f.DrawLine(v.x, t, v.x, b, 0, Ports.white); f.DrawLine(l, v.y, r, v.y, 0, Ports.white)
  END Restore;

  PROCEDURE (v: View) HandlePropMsg (VAR msg: Properties.Message);
    CONST min = 5 * Ports.mm; max = 50 * Ports.mm;
    VAR stdProp: Properties.StdProp; prop: Properties.Property;
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
         msg.w := 20 * Ports.mm; msg.h := 10 * Ports.mm
      ELSE
        Properties.ProportionalConstraint(2, 1, msg.fixedW, msg.fixedH,
                      msg.w, msg.h);
        IF msg.h < min THEN msg.h := min; msg.w := 2 * min
        ELSIF msg.h > max THEN msg.h := max; msg.w := 2 * max
        END
      END
    | msg: Properties.ResizePref DO
      msg.horFitToWin := TRUE; msg.verFitToWin := TRUE
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    | msg: Properties.PollMsg DO
      NEW(stdProp);
      stdProp.color.val := v.c;
      stdProp.valid := {Properties.color};
```

```
        stdProp.known := {Properties.color};
        Properties.Insert(msg.prop, stdProp)
      | msg: Properties.SetMsg DO
        prop := msg.prop;
        WHILE prop # NIL DO
          WITH prop: Properties.StdProp DO
            IF Properties.color IN prop.valid THEN v.c := prop.color.val END
          ELSE
          END;
          prop := prop.next
        END;
        Views.Update(v, Views.keepFrames)
    ELSE  (* пропустить остальные сообщения *)
    END
  END HandlePropMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame; VAR msg: Controllers.Message;
                        VAR focus: Views.View);
    VAR x, y, w, h: INTEGER; m: SET; isDown: BOOLEAN;
  BEGIN
    WITH msg: Controllers.PollOpsMsg DO
      msg.valid := {Controllers.paste}; msg.selectable := TRUE;
      msg.type := "Obx.Tutorial"
    | msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN
        IF msg.char = 1DX THEN INC(v.x, Ports.mm)
        ELSIF msg.char = 1CX THEN DEC(v.x, Ports.mm)
        ELSIF msg.char  = 1EX THEN DEC(v.y, Ports.mm)
        ELSIF msg.char  = 1FX THEN INC(v.y, Ports.mm)
        END;
        Views.Update(v, Views.keepFrames)
      END
    | msg: Controllers.TrackMsg DO
      v.x := msg.x; v.y := msg.y; v.context.GetSize(w, h); v.Restore(f, 0, 0, w, h);
      REPEAT
        f.Input(x, y, m, isDown);
        IF (x # v.x) OR (y # v.y) THEN
          v.x := x; v.y := y; v.Restore(f, 0, 0, w, h)
        END
      UNTIL ~isDown;
      Views.Update(v, Views.keepFrames)
    | msg: Controllers.PollCursorMsg DO
      msg.cursor := Ports.graphicsCursor
    ELSE  (*  пропустить остальные сообщения *)
    END
  END HandleCtrlMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); v.x := 0; v.y := 0; v.c := Ports.black; Views.Deposit(v)
  END Deposit;

END ObxViews6.
```

❗ "ObxViews6.Deposit; StdCmds.PasteView" >< перед выполнением коменды установите курсор здесь

6.6 Разделение модели и отображения

В этом разделе представлены пять все более сложных примеров, которые показывают строку, введенную пользователем. Эта строка представляет собой данные вьюшки, которые показываются на экране. Будет показано, как редактировать строки в нескольких вьюшках, представленных объектами из разных модулей.

В первой версии вьюшки строка сохраняется в самой вьюшке. Каждая вьюшка показывает свою собственную строку. Длина строки ограничена 255 литерами, обработка ошибок не производится.

```
MODULE ObxViews10;

  IMPORT Fonts, Ports, Views, Controllers, Properties;

  CONST d = 20 * Ports.point;

  TYPE
    View = POINTER TO RECORD (Views.View)
      i: INTEGER;          (* позиция следующего свободного места в строке *)
      s: ARRAY 256 OF CHAR  (* строка *)
    END;

  PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawString(d, d, Ports.black, v.s, Fonts.dir.Default())
  END Restore;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                        VAR msg: Controllers.Message;
                        VAR focus: Views.View);
  BEGIN
    WITH msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN   (* принять ввод *)
        v.s[v.i] := msg.char; INC(v.i); v.s[v.i] := 0X;   (* добавить литеру к цепочке *)
        Views.Update(v, Views.keepFrames)   (* восстановить v в любой рамке, которая его
показывает *)
      END
    ELSE                (* пропустить остальные сообщения *)
    END
  END HandleCtrlMsg;

  PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
        msg.w := 10 * d; msg.h := 2 * d
      END
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
```

```
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); v.s := ""; v.i := 0;
    Views.Deposit(v)
  END Deposit;

END ObxViews10.
```

❗ "ObxViews10.Deposit; StdCmds.Open"

Как было сказано в предыдущем разделе, напечатанная литера передается вьюшке в виде записи управляющего сообщения для последующей обработки процедурой отображения HandleCtrlMsg. В качестве параметра в эту процедуру передается сообщение типа Controllers.EditMsg с введенной литерой, и процедура вставляет введенную литеру в поле s. После этого необходимо обновить вьюшку, то есть везде, где эта вьюшка видна, ее необходимо перерисовать в новом статусе. При этом введенная литерная цепочка становится на одну литеру длиннее.

В приведенном выше примере вьюшка содержит смысловую компоненту (поле s типа цепочка литер). Эта компонента должна сохраняться, когда содержимое окна сохраняется на диске. С этой целью вьюшка поддерживает две процедуры с именами Internalize и Externalize, применение которых показано в очередном обновлении сквозного примера.

```
MODULE ObxViews11;
(* То же, что и ObxViews10, но цепочка литер вьюшки может быть сохранена и скопирована *)

  IMPORT Fonts, Ports, Stores, Views, Controllers, Properties;

  CONST d = 20 * Ports.point;

  TYPE
    View = POINTER TO RECORD (Views.View)
      i: INTEGER;              (* position of next free slot in string *)
      s: ARRAY 256 OF CHAR     (* string *)
    END;

  PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
    VAR version: INTEGER;
  BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
      rd.ReadInt(v.i); rd.ReadString(v.s)
    END
  END Internalize;

  PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
  BEGIN
    wr.WriteVersion(0);
    wr.WriteInt(v.i); wr.WriteString(v.s)
  END Externalize;

  PROCEDURE (v: View) CopyFromSimpleView (source: Views.View);
```

```
  BEGIN
    WITH source: View DO
      v.i := source.i; v.s := source.s
    END
  END CopyFromSimpleView;

  PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawString(d, d, Ports.black, v.s, Fonts.dir.Default())
  END Restore;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                          VAR msg: Controllers.Message;
                          VAR focus: Views.View);
  BEGIN
    WITH msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN   (* принять ввод *)
        v.s[v.i] := msg.char; INC(v.i); v.s[v.i] := 0X;   (* добавить литеру к цепочке *)
        Views.SetDirty(v);   (* пометить документ вьюшки как испорченный *)
        Views.Update(v, Views.keepFrames) (* восстановить v в каждом фрейме, в котором
v виден *)
      END
    ELSE                    (* пропустить остальные сообщения *)
    END
  END HandleCtrlMsg;

  PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
        msg.w := 10 * d; msg.h := 2 * d
      END
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    NEW(v); v.s := ""; v.i := 0;
    Views.Deposit(v)
  END Deposit;

END ObxViews11.
```

Несколько комментариев по поводу View.Internalize и View.Externalize:

Во-первых: обе процедуры получают в качестве параметра считыватель и записыватель соответственно. Отображатели файла (file mappers) создаются самостоятельно системой BlackBox, вьюшка просто использует их.

Во-вторых: View.Internalize должна считать в точности те данные (общее количество), которые записаны View.Externalize.

В-третьих: интерфейс пользователя имеет некоторые отличия для документов, содержащих

модифицированные вьюшки, то есть для "грязных" документов. Им может потребоваться дополнительный вопрос о сохранении содержимого при закрытии документа. Для обеспечения этой возможности вьюшка должна сообщить, когда ее сообщение содержит целиком обновленные данные. Это достигается вызовами процедур Views.BeginModification/Views.EndModification.

В дополнение к View.Internalize и View.Externalize в приведенной выше реализации вьюшки появляется еще процедура CopyFromSimpleView. Она должна копировать в себя содержимое вьюшки того же типа, переданной в качестве параметра. CopyFromSimpleView обычно имеет тот же эффект, что и операция выброса (externalized) во временный файл с последующим затягиванием (internalized) во вьюшку-назначение. Кончено же могут быть и исключения. Например, копии текста, содержащие временные метки ошибок не выбрасывают (externalized) их в хранилище.

Для вьюшек, содержащих (изменившуюся) составляющую, следует реализовывать CopyFromSimpleView для возможности их печати. Это нужно, потому что Блэкбокс делает новую копию вьюшки, которая собственно и печатается, во избежание изменений в исходной вьюшке, которые могут произойти в процессе разметки страниц, прокруток и других возможных действий во время печати.

В сущности, в ObxViews11 затронуто большинство возможностей простых вьюшек. Простой вьюшки такого сорта часто бывает достаточно, если легкость реализации имеет большую важность. Тем не менее, в некоторых случаях требуется более продвинутый дизайн вьюшки. В частности, если окно обычно показывает малую часть содержимого вьюшки, то необходима поддержка многооконного редактирования (multi-view editing).

Многооконное редактирование (multi-view editing) означает, что может быть несколько вьюшек, содержащих одни и те же данные. Типичным применением этой возможности является возможность показа различных частей одного документа, причем каждая часть показывается в своем окне. Так, если данные вьюшки изменяются, все задействованные вьюшки должны быть оповещены об этом, чтобы можно было обновить соответствующие изображения.

Следующий пример программы, который отличается от предыдущего только поддержкой многооконного редактирования (multi-view editing), примерно в 2 раза длиннее. Это показывает, что разработка таких вьюшек требует бОльших усилий, чем простое отображение. Такое главное проектировочное решение, несмотря на дополнительные сложности, оправдывается возрастающим удобством.

```
MODULE ObxViews12;
(* То же, что и ObxViews11, но использует отдельную модель для цепочек литер *)

  IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

  CONST d = 20 * Ports.point;

  TYPE
    Model = POINTER TO RECORD (Models.Model)
      i: INTEGER;        (* позиция следующего свободного места в строке *)
      s: ARRAY 256 OF CHAR   (* строка *)
    END;

    View = POINTER TO RECORD (Views.View)
      model: Model
    END;


  (* Модель *)
```

```
PROCEDURE (m: Model) Internalize (VAR rd: Stores.Reader);
   VAR version: INTEGER;
BEGIN
   rd.ReadVersion(0, 0, version);
   IF ~rd.cancelled THEN
      rd.ReadInt(m.i); rd.ReadString(m.s)
   END
END Internalize;

PROCEDURE (m: Model) Externalize (VAR wr: Stores.Writer);
BEGIN
   wr.WriteVersion(0);
   wr.WriteInt(m.i); wr.WriteString(m.s)
END Externalize;

PROCEDURE (m: Model) CopyFrom (source: Stores.Store);
BEGIN
   WITH source: Model DO
      m.i := source.i; m.s := source.s
   END
END CopyFrom;


(*  Отображение *)

PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
   VAR version: INTEGER; st: Stores.Store;
BEGIN
   rd.ReadVersion(0, 0, version);
   IF ~rd.cancelled THEN
      rd.ReadStore(st);
      v.model := st(Model)
   END
END Internalize;

PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
BEGIN
   wr.WriteVersion(0);
   wr.WriteStore(v.model)
END Externalize;

PROCEDURE (v: View) CopyFromModelView (source: Views.View;
                        model: Models.Model);
BEGIN
   v.model := model(Model)
END CopyFromModelView;

PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
BEGIN
   f.DrawString(d, d, Ports.black, v.model.s, Fonts.dir.Default())
END Restore;

PROCEDURE (v: View) ThisModel (): Models.Model;
BEGIN
   RETURN v.model
END ThisModel;
```

```
  PROCEDURE (v: View) HandleModelMsg (VAR msg: Models.Message);
  BEGIN
    Views.Update(v, Views.keepFrames)   (* восстановить v в любой рамке, которая его
отображает *)
  END HandleModelMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame;
                        VAR msg: Controllers.Message;
                        VAR focus: Views.View);
    VAR m: Model; umsg: Models.UpdateMsg;
  BEGIN
    WITH msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN
        m := v.model;
        m.s[m.i] := msg.char; INC(m.i);
         m.s[m.i] := 0X;   (* добавить литеру к цепочке *)
        Views.SetDirty(v);   (* пометить документ вьюшки как "испорченный" *)
        Models.Broadcast(m, umsg)   (* обновить все вьюшки этой модели <на>*)
      END
    ELSE                (* пропустить остальные сообщения *)
    END
  END HandleCtrlMsg;

  PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
        msg.w := 10 * d; msg.h := 2 * d
      END
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View; m: Model;
  BEGIN
    NEW(m); m.i := 0; m.s := "";
     NEW(v); v.model := m; Stores.Join(v, m);
    Views.Deposit(v)
  END Deposit;

END ObxViews12.
```

❗ "ObxViews12.Deposit; StdCmds.Open"

Многооконное редактирование (multi-view editing) реализуется путем выделения данных вьюшки в отдельную структуру, называемую моделью (model). Эта модель используется совместно всеми вьюшками, то есть каждая такая вьюшка содержит ссылку на модель.

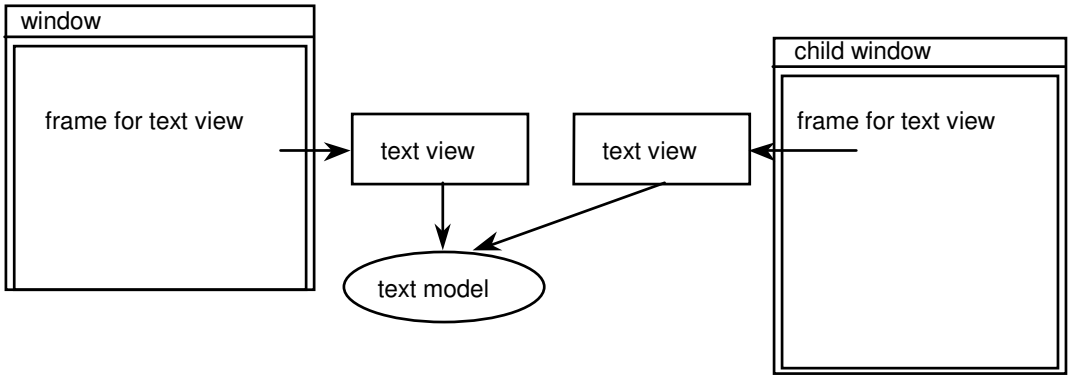

Рисунок 6-2. Две вьюшки для одной модели.

Модель, как и вьюшка, является расширением Stores.Store, который, в свою очередь, является базовым типом для всех устойчивых (persistent) объектов. Для хранилищ (Stores) определены процедуры Internalize и Externalize, которые уже были рассмотрены для вьюшек. В ObxViews12 модель хранит строку и ее длину, в то время как вьюшка хранит модель целиком. С этой целью Stores.Writer поддерживает WriteStore и StoresReader поддерживает ReadStore процедуры.

Хранилища (Stores) могут состоять из нескольких графических образов. Например, одна или несколько вьюшек могут ссылаться на одну модель, а модели и вьюшки содержат расширения. Графический образ хранилища может быть сохранен в файле, называемом документ (document). Для возможности распознавания границ документа при выгрузке, хранилища ограничены доменом (domain). Все хранилища из документа относятся к одному и тому же домену (домен представлен объектом типа Stores.Domain).

Домены так же используются в качестве контейнера для операций, то есть команд, выполняемых над хранилищем (store) с возможностью их отмены и повторения (см. следующую главу). История операций привязана к домену. Более того, домены определяют набор объектов, которые получают широковещательные сообщения. (Views.Broadcast, Views.Domaincast, Models.Broadcast).

За дальнейшей информацией о хранилищах (stores) и доменах обращайтесь к встроенной документации модуля Stores.

Теперь, когда модель отделена от отображения, копирование выполняется только для модели, но не для вьюшки.

Процедура View.ThisModel возвращает модель для вьюшки. Реализация по умолчанию возвращает NIL, то есть ее нужно перекрыть для вьюшек, содержащих модели. Эта процедура используется оболочкой для поиска всех вьюшек, показывающих данную модель с целью реализации широковещания для модели, как показано в следующем абзаце.

Вьюшки с моделями должны реализовать процедуру View.CopyFromModelView взамен View.CopyFromSimpleView. Подобно Internalize и Externalize, процедура CopyFromSimpleView и CopyFromModelView экспортируются как процедуры для реализации, то есть они только реализуются, но не вызываются. Они вызываются внутри модуля Views в функциях CopyOf и CopyWithNewModel.

При изменении модели все вьюшки, показывающие модель, должны быть обновлены. Для этой цели генерируется широковещание (broadcast) <рассылка сообщений> для модели: модель рассылает сообщения всем отображениям, содержащим конкретную модель, с информацией о произошедших изменениях в модели.

Это дает возможность вьюшке обновить изображение на экране. Широковещание модели происходит в процедуре View.HandleCtrlMsg путем вызова Models.Broadcast. Сообщения получает каждая вьюшка, содержащая корректную модель, посредством процедуры View.HandleModelMsg.

Такая косвеная связь важна, если одну модель представляют разные типы вьюшек, например, табличное и графическое представление модели табличных данных, и, вместо обновления всей вьюшки (как сделано в предыдущих примерах), обновляется только необходимая ее часть. Необходимая для обновления часть может быть совершенно разной для разных типов вьюшек.

Очень сложные модели могут содержать (или в них могут быть внедрены) несколько вьюшек в качестве составных частей. Например, текстовая вьюшка может изображать не только текстовую модель, содержащую только литеры (её «естественное» содержимое), но и графические или другие отображения, обособленно следующие в текстовом потоке.

Комбинация многооконного (multi-view) редактирования и иерархического внедрения вьюшек может привести к ситуации, когда две текстовых вьюшки показывают одну текстовую модель, в которую в свою очередь внедрена графическая вьюшка. Каждая текстовая вьюшка существует в отдельном окне, поэтому имеет собственный фрейм. Но графическая вьюшка уникальна тем, что внедрена в текстовую модель, которая, в свою очередь, разделяется обеими текстовыми вьюшками. Тем не менее, графика может показываться одновременно в обеих текстовых вьюшках, а, значит, для внедренной вьюшки может быть два фрейма.

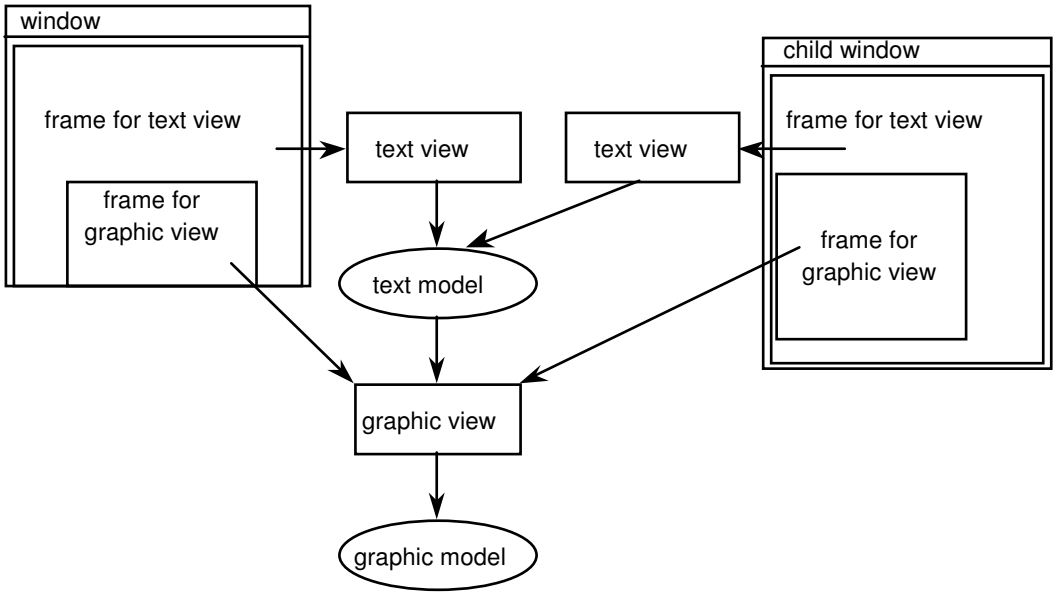

Рисунок 6-3. Два фрейма для одной вьюшке.

Итак, одна и та же вьюшка может показываться в нескольких местах экрана одновременно. Это означает, что при изменении вьюшки должны обновиться несколько участков: вьюшка должна восстановить необходимые участки <изображения> в каждом фрейме. Как следствие, механизм сообщений должен обеспечить обновление вьюшки для каждого из ее фреймов. Это достигается подобно механизму сообщений при изменении моделей: с помощью рассылки сообщений вьюшки (view broadcast). К счастью, стандартный механизм обновлений внутри оболочки автоматически обрабатывает второй уровень широковещания (см. ниже).

Теперь можно свести вместе типичные события, происходящие при взаимодействии пользователя и вьюшки.

Обработка сообщений контроллера:

1. Какое-либо управляющее сообщение посылается вьюшке, обладающей фокусом ввода.
2. Отображение, обладающее фокусом ввода, интерпретирует сообщение и соответствующим образом изменяет свою модель.

3. Модель осуществляет широковещание сообщений, описывающих случившиеся изменения.

Обработка сообщений модели:

4. Каждая вьюшка для данной модели получает сообщение модели.
5. Они определяют, как должно измениться изображение в связи с изменениями модели.
6. Они (вьюшки) инициируют рассылку сообщений вьюшки, описывающих как должно измениться изображение.

Обработка сообщений вьюшки:

7. Каждая вьюшка получает уведомительное сообщение по одному разу на каждый фрейм.
8. Каждый раз вьюшка обновляет изображение в своих фреймах в соответствии с сообщениями вьюшки.

Этот механизм лежит в основе компонентного каркаса Блэкбокса. Его понимание — наиболее важный и сложный пункт при реализации вьюшек. Интерпретация различных управляющих сообщений, определенных в модуле Controllers, является более чем необходимой для понимания трудностей новой концепции. Более того, необходимо интерпретировать только интересующие нас управляющие сообщения, потому что неподходящие сообщения следует просто игнорировать.

Обычно шаги 6, 7 и 8 обрабатываются каркасом автоматически: вьюшки просто вызывают Views.Update для перерисовки всей вьюшки целиком или Views.UpdateIn для перерисовки какой-либо прямоугольной области. Вызов этих процедур обновления приводит к так называемому ленивому обновлению (lazy update), то есть оболочка помечает область для обновления, но перерисовка НЕ производится сразу. Вместо этого вначале завершаются команды, чтобы все структуры данных изменяемой вьюшки пришли в согласованное состояние. Только изображения вьюшек остаются теми же, что и до выполнения команд. Только после этого Блэкбокс перерисовывает все области изображения, помеченные к обновлению. Это означает, что для каждого фрейма, отрисованного в отображении, вызывается метод вьюшки Restore и в качестве аргумента передается фрейм. Если вьюшка показывается в двух (или более) фреймах разных окон, то обновляется два (или более) число раз, по одному разу на каждый фрейм. Когда оболочка обновляет фреймы внутри окна, происходит временное перенаправление операции в порт фонового пиксельного буфера. После обновления всех рамок окна, подлежащих перерисовке, Блэкбокс копирует пиксельный буфер в пиксели экрана и отключает перенаправление операции перерисовки.

В результате применения механизмов буферизации и ленивого обновления минимизируется мерцание. Механизм ленивого обновления (как для Apple MacOS, так и для Microsoft Windows) означает, что для сложных операций редактирования данная область экрана будет обновляться однократно. Это позволяет уменьшить мерцание, возникающее при быстрой многократной перерисовке какой-либо области. Механизм буферизации уменьшает мерцание при отрисовке слоев "снизу-вверх" (так как это так же означает быструю многократную перерисовку). Например, вначале может отрисовываться фон одним цветом, а затем поверх рисуется прямоугольник с заливкой другого цвета. Благодаря механизму буферизации пользователь не увидит такое состояние экрана, когда фон отрисован, а прямоугольник — еще нет.

Механизм ленивого обновления обладает еще одним преимуществом: целиком разделяется изменение модели и ее отрисовка. Без этого механизма было бы действительно сложно правильно реализовать обновление экрана при сложном редактировании. Например, рассмотрим перемещение выделения графического объекта в графическом редакторе. Вначале выделенный объект должен быть уделен со старой позиции, что означает, что объекты, которые ранее были скрыты, должны быть перерисованы (но не сам выделенный объект). Затем позиция объекта изменяется и он должен быть перерисован в новой

позиции (если исходная и конечная позиции перекрываются, перерисовка произойдет дважды). Перерисовка может потребоваться несколько раз, так как одни и те же данные могут показываться в нескольких окнах. Но, конечно же, изменение позиции объекта происходит только единожды. Гораздо менее мудрено просто вызвать Views.UpdateIn для прямоугольников, которые требуют (могут требовать) обновления и позволить оболочке вызвать требуемые Restore позже.

Обычно механизма ленивого обновления вполне достаточно. Таким образом программирование вьюшек с использованием чистых сообщений вьюшек (explicit view messages) применяется, только если не требуется полного обновления всех фреймов вьюшки. То есть для специальных символов типа: отметки выделения, фокуса, каретки, "резиновых" рамок и тому подобных обратных связей в процессе перемещения мыши. Подобные метки иногда могут быть обработаны более эффективно, так как они являются непроизвольными: двойное их применение приводит к их подавлению. Поэтому часто метки переключаются пользовательскими сообщениями вьюшек в процессе выполнения команды, но не посредством механизма ленивого обновления. За исключением подобных легковесных отметок, всегда следует применять описанный выше механизм ленивого обновления. Отметим наличие метода Ports.Frame.MarkRect, специально предназначенного для отрисовки подобных меток.

6.7 Операции

Теперь, после того, как рассмотрены критические составляющие реализации вьюшек, можно представить второстепенные особенности. В следующей программе, являющейся вариантом приведенных выше, управляющие сообщения не интерпретируются напрямую. Вместо этого создается и выполняется объект операции (operation object). Операция обеспечивает возможности отката, как показано в нижеследующей программе.

В этом примере определяется операция PasteCharOp. Процедура Do операции, выполняющая желаемые модификации, должна быть непроизвольной (involutory), то есть при повторном вызове она нейтрализует предыдущий вызов. Флаг PasteCharOp.do применяется для обозначения того, была ли литера PasteCharOp.char вставлен или удален из модели PasteCharOp.m. В любом случае процедура Do обязана обновить все вьюшки для модели PasteCharOp.m с помощью широковещания модели.

Процедура NewPasteCharOp создает новую операцию, процедура Models.Do выполняет эту операцию, то есть вызывается процедура Do операции, а собственно операция записывается для возможности дальнейшего отката. Наименование операции, определенной в команде Models.Do появляется в меню Edit после пунктов Undo или Redo соответственно.

```
MODULE ObxViews13;
(* Такой же, как ObxViews12, но генерирует операции с возможность отмены для вставки литеры *)

  IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

  CONST d = 20 * Ports.point;

  TYPE
    Model = POINTER TO RECORD (Models.Model)
      i: INTEGER;        (* позиция следующего свободного места в цепочке *)
      s: ARRAY 256 OF CHAR   (* литерная цепочка *)
    END;

    View = POINTER TO RECORD (Views.View)
      model: Model
    END;
```

```
   PasteCharOp = POINTER TO RECORD (Stores.Operation)
      model: Model;
      char: CHAR;
      do: BOOLEAN
   END;


(* PasteCharOp *)

PROCEDURE (op: PasteCharOp) Do;
   VAR m: Model; msg: Models.UpdateMsg;
BEGIN
   m := op.model;
   IF op.do THEN         (* do изменение операции *)
      m.s[m.i] := op.char; INC(m.i)   (* вставка литеры в цепочку *)
   ELSE                  (* отмена операции *)
      DEC(m.i)           (* удаление литеры из цепочки *)
   END;
   m.s[m.i] := 0X;
   op.do := ~op.do;      (* переключатель между "do" и "undo" *)
   Models.Broadcast(m, msg)   (* обновить все вьюшки модели*)
END Do;

PROCEDURE NewPasteCharOp (m: Model; char: CHAR): PasteCharOp;
   VAR op: PasteCharOp;
BEGIN
   NEW(op); op.model := m; op.char := char; op.do := TRUE;
   RETURN op
END NewPasteCharOp;


(* Модель *)

PROCEDURE (m: Model) Internalize (VAR rd: Stores.Reader);
   VAR version: INTEGER;
BEGIN
   rd.ReadVersion(0, 0, version);
   IF ~rd.cancelled THEN
      rd.ReadInt(m.i); rd.ReadString(m.s)
   END
END Internalize;

PROCEDURE (m: Model) Externalize (VAR wr: Stores.Writer);
BEGIN
   wr.WriteVersion(0);
   wr.WriteInt(m.i); wr.WriteString(m.s)
END Externalize;

PROCEDURE (m: Model) CopyFrom (source: Stores.Store);
BEGIN
   WITH source: Model DO
      m.i := source.i; m.s := source.s
   END
END CopyFrom;
```

```
  (* Отображение *)

  PROCEDURE (v: View) Internalize (VAR rd: Stores.Reader);
    VAR version: INTEGER; st: Stores.Store;
  BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
      rd.ReadStore(st);
      v.model := st(Model)
    END
  END Internalize;

  PROCEDURE (v: View) Externalize (VAR wr: Stores.Writer);
  BEGIN
    wr.WriteVersion(0);
    wr.WriteStore(v.model)
  END Externalize;

  PROCEDURE (v: View) CopyFromModelView (source: Views.View; model: Models.Model);
  BEGIN
    v.model := model(Model)
  END CopyFromModelView;

  PROCEDURE (v: View) Restore (f: Views.Frame; l, t, r, b: INTEGER);
  BEGIN
    f.DrawString(d, d, Ports.black, v.model.s, Fonts.dir.Default())
  END Restore;

  PROCEDURE (v: View) ThisModel (): Models.Model;
  BEGIN
    RETURN v.model
  END ThisModel;

  PROCEDURE (v: View) HandleModelMsg (VAR msg: Models.Message);
  BEGIN
    Views.Update(v, Views.keepFrames) (* восстановить v в любом фрейме, ее
избражающем *)
  END HandleModelMsg;

  PROCEDURE (v: View) HandleCtrlMsg (f: Views.Frame; VAR msg: Controllers.Message;
                        VAR focus: Views.View);
    VAR op: Stores.Operation;
  BEGIN
    WITH msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN
        op := NewPasteCharOp(v.model, msg.char);   (* сгенерировать операцию *)
        Models.Do(v.model, "Typing", op)   (* выполнить операцию *)
      END
    ELSE                (* пропустить остальные сообщения *)
    END
  END HandleCtrlMsg;

  PROCEDURE (v:View) HandlePropMsg (VAR msg: Properties.Message);
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
```

```
        msg.w := 10 * d; msg.h := 2 * d
      END
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
  END HandlePropMsg;

  PROCEDURE Deposit*;
    VAR v: View; m: Model;
  BEGIN
    NEW(m); m.i := 0; m.s := "";
    NEW(v); v.model := m; Stores.Join(v, m);
    Views.Deposit(v)
  END Deposit;

END ObxViews13.
```

❗ "ObxViews13.Deposit; StdCmds.Open"

Другие примеры с применением механизма отката/повтора посредством операций можно найти, например, в ObxOmosi или ObxLines.

6.8 Разделение интерфейса и реализации

В качестве последней версии простой вьюшки, изменим предыдущий вариант путем экспорта типов Model и View и разделения интерфейса и реализации этих двух типов. Для выполнения последнего вводится так называемые фабричные объекты (directory objects), которые создают (скрытые) предопределенные реализации абстрактных типов данных. Теперь пользователь обязан реализовать собственные версии абстрактных типов и предоставить их через собственный фабричный объект, который не может быть унаследован от реализации по умолчанию во избежание проблемы хрупких базовых классов.

Реальные подсистемы Блэкбокса будут дополнительно разделять модуль на два: один для модели, другой для вьюшки. Может быть введен и третий модуль с командами, если число и сложность команд тревожно высоко (например: TextModels, TextViews и TextCmds). Еще более сложный тип вьюшек в свою очередь следует разделять на вьюшки и контроллеры, определенные в отдельных модулях.

```
MODULE ObxViews14;
(* Такой же как ObxViews13, но интерфейсы и реализация разделены, и операция
напрямую в процедуре вставки *)

  IMPORT Fonts, Ports, Stores, Models, Views, Controllers, Properties;

  CONST d = 20 * Ports.point;

  TYPE
    Model* = POINTER TO ABSTRACT RECORD (Models.Model) END;

    ModelDirectory* = POINTER TO ABSTRACT RECORD END;

    View* = POINTER TO ABSTRACT RECORD (Views.View) END;

    Directory* = POINTER TO ABSTRACT RECORD END;
```

```
   StdModel = POINTER TO RECORD (Model)
      i: INTEGER;          (* следующая свободная позиции в цепочке *)
      s: ARRAY 256 OF CHAR    (* строка *)
   END;

   StdModelDirectory = POINTER TO RECORD (ModelDirectory) END;

   StdView = POINTER TO RECORD (View)
      model: Model
   END;

   StdDirectory = POINTER TO RECORD (Directory) END;

   PasteCharOp = POINTER TO RECORD (Stores.Operation)
      model: StdModel;
      char: CHAR;
      do: BOOLEAN
   END;


VAR
   mdir-: ModelDirectory;
   dir-: Directory;


(* Модель*)

PROCEDURE (m: Model) Insert* (char: CHAR), NEW, ABSTRACT;
PROCEDURE (m: Model) Remove*, NEW, ABSTRACT;
PROCEDURE (m: Model) GetString* (OUT s: ARRAY OF CHAR), NEW, ABSTRACT;

(* ModelDirectory *)

PROCEDURE (d: ModelDirectory) New* (): Model, NEW, ABSTRACT;

(* PasteCharOp *)

PROCEDURE (op: PasteCharOp) Do;
   VAR m: StdModel; msg: Models.UpdateMsg;
BEGIN
   m := op.model;
   IF op.do THEN           (* do изменение операции *)
      m.s[m.i] := op.char; INC(m.i);
   ELSE                 (* отменить изменение операции *)
      DEC(m.i)           (* удалить литеру из цепочки *)
   END;
   m.s[m.i] := 0X;
   op.do := ~op.do;         (* переключатель между "do" и "undo" *)
   Models.Broadcast(m, msg)   (* обновить все отображения модели*)
END Do;


(* StdModel *)

PROCEDURE (m: StdModel) Internalize (VAR rd: Stores.Reader);
   VAR version: INTEGER;
```

```
  BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
      rd.ReadInt(m.i); rd.ReadString(m.s)
    END
  END Internalize;

  PROCEDURE (m: StdModel) Externalize (VAR wr: Stores.Writer);
  BEGIN
    wr.WriteVersion(0);
    wr.WriteInt(m.i); wr.WriteString(m.s)
  END Externalize;

  PROCEDURE (m: StdModel) CopyFrom (source: Stores.Store);
  BEGIN
    WITH source: StdModel DO
      m.i := source.i; m.s := source.s
    END
  END CopyFrom;


  PROCEDURE (m: StdModel) Insert (char: CHAR);
    VAR op: PasteCharOp;
  BEGIN
    NEW(op); op.model := m; op.char := char; op.do := TRUE;
    Models.Do(m, "insertion", op)
  END Insert;

  PROCEDURE (m: StdModel) Remove;
    VAR msg: Models.UpdateMsg;
  BEGIN
    DEC(m.i); m.s[m.i] := 0X;
    Models.Broadcast(m, msg)   (* обновить все отображения этой модели*)
  END Remove;

  PROCEDURE (m: StdModel) GetString (OUT s: ARRAY OF CHAR);
  BEGIN
    s := m.s$
  END GetString;


  (* StdModelDirectory *)

  PROCEDURE (d: StdModelDirectory) New (): Model;
    VAR m: StdModel;
  BEGIN
    NEW(m); m.s := "";  m.i := 0; RETURN m
  END New;

  (* Directory *)

  PROCEDURE (d: Directory) New* (m: Model): View, NEW, ABSTRACT;

  (* StdView *)

  PROCEDURE (v: StdView) Internalize (VAR rd: Stores.Reader);
```

```
    VAR version: INTEGER; st: Stores.Store;
  BEGIN
    rd.ReadVersion(0, 0, version);
    IF ~rd.cancelled THEN
      rd.ReadStore(st);
      IF st IS Model THEN
        v.model := st(Model)
      ELSE
        (* реализация конкретной модели не может быть загружена->
           было создано хранилище-нелегал *)
        rd.TurnIntoAlien(Stores.alienComponent)
        (* загрузка v отменена *)
      END
    END
  END Internalize;

  PROCEDURE (v: StdView) Externalize (VAR wr: Stores.Writer);
  BEGIN
    wr.WriteVersion(0);
    wr.WriteStore(v.model)
  END Externalize;

  PROCEDURE (v: StdView) CopyFromModelView (source: Views.View; model: Models.Model);
  BEGIN
    WITH source: StdView DO
      v.model := model(Model)
    END
  END CopyFromModelView;

  PROCEDURE (v: StdView) Restore (f: Views.Frame; l, t, r, b: INTEGER);
    VAR s: ARRAY 256 OF CHAR;
  BEGIN
    v.model.GetString(s);
    f.DrawString(d, d, Ports.black, s, Fonts.dir.Default())
  END Restore;

  PROCEDURE (v: StdView) ThisModel (): Models.Model;
  BEGIN
    RETURN v.model
  END ThisModel;

  PROCEDURE (v: StdView) HandleModelMsg (VAR msg: Models.Message);
  BEGIN
    Views.Update(v, Views.keepFrames) (* восстановить v в любом фрейме,
отображающей его *)
  END HandleModelMsg;

  PROCEDURE (v: StdView) HandleCtrlMsg (f: Views.Frame;
                        VAR msg: Controllers.Message;
                        VAR focus: Views.View);
  BEGIN
    WITH msg: Controllers.EditMsg DO
      IF msg.op = Controllers.pasteChar THEN
        v.model.Insert(msg.char)  (* отменяемая вставка *)
      END
    ELSE                (* пропустить другие сообщения *)
```

```
    END
  END HandleCtrlMsg;

  PROCEDURE (v:StdView) HandlePropMsg (VAR msg: Properties.Message);
  BEGIN
    WITH msg: Properties.SizePref DO
      IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
        msg.w := 10 * d; msg.h := 2 *d
      END
    | msg: Properties.FocusPref DO
      msg.setFocus := TRUE
    ELSE
    END
  END HandlePropMsg;


  (* StdDirectory *)

  PROCEDURE (d: StdDirectory) New* (m: Model): View;
    VAR v: StdView;
  BEGIN
    ASSERT(m # NIL, 20);
    NEW(v); v.model := m; Stores.Join(v, m);
    RETURN v
  END New;


  PROCEDURE Deposit*;
    VAR v: View;
  BEGIN
    v := dir.New(mdir.New());
    Views.Deposit(v)
  END Deposit;

  PROCEDURE SetModelDir* (d: ModelDirectory);
  BEGIN
    ASSERT(d # NIL, 20);
    mdir := d
  END SetModelDir;

  PROCEDURE SetDir* (d: Directory);
  BEGIN
    ASSERT(d # NIL, 20);
    dir := d
  END SetDir;

  PROCEDURE Init;
    VAR md: StdModelDirectory; d: StdDirectory;
  BEGIN
    NEW(md); mdir := md;
    NEW(d); dir := d
  END Init;

BEGIN
  Init
END ObxViews14.
```

❗ "ObxViews14.Deposit; StdCmds.Open"

Модуль DevMarkers представляет собой простой пример модуля BlackBox, отвечающий такому дизайну. Вьюшка-маркер является простой вьюшкой без модели, поэтому предоставляется только фабричный объект для создания новых вьюшек. Второй фабричный объект DevMarkers.StdDir сохраняет стандартную реализацию, обеспечиваемую данным модулем. Он может применяться для восстановления реализации по умолчанию или для повторного использования реализации по умолчанию (как экземпляр) в другом компоненте.

Разделение определения типов и их реализации рекомендуется при проектировании новых подсистем Блэкбокса. Простые вьюшки, не предназначенные для широкого применения или дальнейшего расширения, могут распространяться без таких дополнительных усилий.

Следует отметить, что мастер подсистем (subsystem wizard) (пункт меню Tools -> Create Subsystem...) помогает создавать шаблоны различных видов вьюшек. В частности возможно создавать раздельные модули для модели и вьюшки. Инструмент использует текстовые шаблоны, хранящиеся в Dev/Rsrc/New. Может быть интересным изучение файлов Models5, Views5 и Cmds5 в этом каталоге.

Приложение А: Краткая история языка Паскаль

Замечания переводчика даны в угловых скобках <>.
Некоторые термины оригинала приведены в квадратных скобках [].

Алгол

Язык Компонентный Паскаль является кульминацией нескольких десятилетий исследовательской работы. Это самый младший член семейства алголоподобных языков. Алгол, определенный в 1960, был первым языком высокого уровня с синтаксисом, который был легко читаем, *структурирован* и описан формальным образом. Несмотря на его успешное использование в качестве нотации для математичесих алгоритмов, в нем недоставало важных типов данный, таких как указатели и литеры.

Паскаль

В конце 60-х гг. было выдвинуто несколько предложений об эволюционном развитии Алгола. Самым успешным оказался Паскаль, определенный в 1970 профессором Никлаусом Виртом из ETH, швейцарского Федерального политехнического университета в Цюрихе. Наряду с очищением языка от некоторых непрозрачных средств Алгола, в Паскале добавлена возможность объявления новых структур данных, построенных из уже существующих более простых. Паскаль также поддерживал *динамические структуры данных*, т.е. такие структуры данных, которые могут расти или уменьшаться во время выполнения программы. Паскаль получил мощный импульс к распространению, когда в ETH был выпущен компилятор, порождавший простой промежуточный код для виртуальной машины (P-код) вместо «родного» кода для конкретного процессора. Это существенно упростило перенос Паскаля на другие процессорные архитектуры, т.к. для этого нужно было только написать новый интерпретатор для P-кода вместо всего нового компилятора. Один из таких проектов был предпринят в Университете Калифорнии в Сан-Диего. Замечательно, что эта реализация (UCSD Pascal) не требовала мощного компьютера [mainframe] и могла работать на новых тогда персональных компьютерах семейства Apple II. Это дало распространению Паскаля второй серьезный импульс. Третьим был выпуск компанией Борланд продукта Турбо Паскаль, содержавшего быстрый и недорогой компилятор вместе с интегрированной средой разработки программ для компьютеров IBM PC. Позднее Борланд возродил свою версию Паскаля, выпустив среду быстрой разработки приложений Дельфи.

Паскаль сильно повлиял на дизайн и эволюцию многих других языков, от Ada до Visual Basic.

Модула-2

В середине 70-х гг., вдохновленный годичным академическим отпуском, проведенным в исследовательском центре PARC компании Xerox в г. Пало Альто, Вирт начал проект по созданию нового компьютера класса рабочая станция <проект Lilith>. Компьютер должен был быть полностью программируемым на языке высокого уровня, так что язык должен был обеспечить прямой доступ к аппаратному уровню. Далее, он должен был поддерживать *коллективное программирование* и современные принципы разработки программного обеспечения, такие как *абстрактные типы данных*. Эти требования были реализованы в языке программирования Модула-2 (1979). Модула-2 сохранила успешно зарекомендовавшие себя средства Паскаля и добавила систему модулей, а также контролируемые возможности обойти систему типов языка в случаях *программирования низкого уровня*, например, при написании драйверов устройств. Модули могли добавляться к операционной системе непосредственно во время работы. На самом деле вся операционная система представляла собой набор модулей без выделенного ядра или иного подобного объекта. Модули могли

компилироваться и загружаться раздельно, причем обеспечивалась *полная проверка типов и версий их интерфейсов*.

Успех Модулы-2 был наиболее значителен в задачах с высокими требованиями на надежность, таких как системы управления движением. <Например, бортовое программное обеспечение на запускаемых в настоящее время российских спутниках связи пишется на Модуле-2 с помощью кросс-среды разработки приложений, описанной А.А.Колташевым в докладе на конференции JMLC'2003; см. Modular Programming Languages. Lecture Notes in Computer Science (LNCS 2789), Springer-Verlag, 2003, сс. 98-101>

Simula, Smalltalk и Cedar

Однако Вирт продолжал интересоваться прежде всего настольными компьютерами, и опять важный импульс пришел из центра PARC компании Xerox. В этом центре были изобретены рабочая станция, лазерный принтер, локальная сеть, графический дисплей и многие другие технологии, расширяющие возможности использования компьютеров человеком. Кроме того, в центре PARC были популяризированы некоторые более старые и малоизвестные технологии, такие как мышь, интерактивная графика и, наконец, *объектно ориентированное программирование*. Эта последняя концепция (хотя и не сам термин) была впервые использована в языке высокого уровня Симула (1966) — еще одном из семейства алголоподобных языков. Как и предполагает имя, язык Simula использовал объектные технологии прежде всего для целей моделирования [simulation]. Однако язык Smalltalk (1983), разработанный в центре PARC компании Xerox, использовал их практически для *всего, что угодно*. Проект Smalltalk был также пионерским в плане дизайна пользовательского интерфейса: графический интерфейс пользователя (GUI), каким мы его теперь знаем, был разработан для системы Smalltalk.

В центре PARC эти идеи повлияли на другие проекты, например, паскалеподобный язык Cedar. Как и Smalltalk и позднее Оберон, Cedar представлял собой не только язык программирования, но и операционную систему. Операционная система Cedar была весьма впечатляющей и мощной, однако сложной и нестабильной.

Оберон [Oberon]

Проект Оберон был начат в 1985 в ETH Виртом и его коллегой Юргом Гуткнехтом [Jurg Gutknecht]. Это была попытка выделить все существенное из системы Cedar в виде универсальной, но все же обозримой операционной системы для рабочих станций. Получившаяся система оказалась очень маленькой и эффективной, прекрасно работала в оперативной памяти размером всего 2 MB и требовала при этом лишь 10 MB памяти на диске. Важной причиной малого размера системы Оберон был ее компонентный дизайн: вместо интеграции всех желаемых средств в один монолитный программный колосс, менее часто используемые программные компоненты (модули) могли быть реализованы как расширение ядра системы. Такие компоненты загружались, только когда они были действительно нужны, и они могли совместно использоваться всеми приложениями.

Вирт понял, что компонентно-ориентированное программирование требовало некоторых средств объектно-ориентированного программирования, таких как *упрятывание информации* [*information hiding*], *позднее связывание* [*late binding*] и *полиморфизм*.

Упрятывание информации было сильной чертой Модулы-2. Позднее связывание поддерживалось в Модуле-2 посредством процедурных переменных. Однако там не было полиморфизма. Поэтому Вирт добавил *расширенное переопределение типов* [*type extension*]: записевый тип мог быть объявлен как расширение <потомок> другого записевого типа <предка>. Тип-потомок можно было использовать всюду вместо одного из его предков.

Но компонентно-ориентированное программирование выходит за рамки объектно-ориентированного. В системе, построенной из компонентов, компонент может разделять свои структуры данных с произвольным числом других компонентов, о которых она ничего не знает. Эти компоненты обычно также не знают о существовании друг друга. Такое взаимное незнание делает управление динамическими структурами данных, и в частности

правильное освобождение уже ненужной памяти, *принципиально* более трудной проблемой, чем в закрытых программных системах. Следовательно, необходимо оставить на долю реализации языка всю работу по определению момента, когда какая-то область памяти более не нужна, чтобы повторно использовать ее без ущерба для безопасности системы. Системный сервис, выполняющий такую автоматическую утилизацию памяти, называется *сборщик мусора* [*garbage collector*]. Сбор мусора предотвращает две из числа наиболее труднонаходимых и попросту опасных ошибок в программах: *утечки памяти* [*memory leaks*] (когда более не используемая память не освобождается) и *висячие ссылки* [*dangling pointers*] (преждевременное освобождение памяти). Висячие ссылки позволяют одному компоненту разрушить структуры данных, принадлежащие другим. Такое нарушение *защиты типов* [*type safety*] должно быть предотвращено, т.к. компонентные системы могут содержать многие независимо написанные компоненты неизвестного качества (например, полученные из Интернета).

Хотя алголоподобные языки всегда имели высокую репутацию в плане безопасности, введение полной защиты типов (и, следовательно, сбора мусора) явилось качественным скачком. Именно по этой причине полная совместимость с Модулой-2 оказалась невозможной. Получившаяся модификация Модулы-2 была названа как и вся система — Оберон.

Система модулей в Обероне, как и в Модуле-2, обеспечивала упрятывание информации для целых семейств типов, а не только для отдельных объектов. Это позволило определять и гарантировать инварианты для нескольких взаимодействующих объектов. Другими словами, разработчики получили возможность разрабатывать механизмы защиты более высокого уровня, отталкиваясь от базовых средств защиты на уровне модулей и защиты типов, обеспечиваемых хорошей реализацией Оберона.

Ортодоксальные объектно-ориентированные языка типа Smalltalk пренебрегали как типизацией переменных (вообще не поддерживая типы), так и упрятыванием информации (ограничивая ее объектами и классами), что явилось большим шагом назад в технологии разработки программного обеспечения. Оберон примирил миры объектно-ориентированного и модульного программирования.

Последнее требование компонентно-ориентированного программирования — возможность *динамически загружать* новые компоненты. В Обероне единица загрузки та же, что и единица компиляции — модуль.

Компонентный Паскаль

В 1992 г. сотрудничество с профессором Х.П.Мёссенбёком (H.P. Mossenbock) привело к нескольким добавлениям к первоначальному языку Оберон («Оберон-2»). Так возник фактический стандарт языка. <На данный момент (2009) это утверждение потеряло свой первоначальный смысл, т.к. Оберон-2 потеснен Компонентным Паскалем. —*прим. перев.*>

В 1997 г. компания Oberon microsystems, Inc., отпочковавшаяся от ETH (с Виртом в составе совета директоров), сделала некоторые небольшие добавления к Оберону-2 и назвала его Компонентный Паскаль, чтобы четче выразить как его нацеленность (компонентно-ориентированное программирование), так и его происхождение (Паскаль). Это промышленная версия Оберона, являющаяся наследницей первоначального Паскаля и Модулы-2.

Главная идея уточнений по сравнению с Обероном-2 была в том, чтобы дать проектировщику компонентного каркаса [framework] (т.е. интерфейсов модулей, определяющих абстрактные классы для конкретной проблемной области) более полный контроль над ее проектируемыми свойствами в плане безопасности. Положительным результатом стало то, что теперь легче обеспечить целостность больших компонентных систем, что особенно важно во время итеративных циклов проектирования, когда библитека разрабатывается, и позднее, когда архитектура системы должна быть переработана [refactored], чтобы обеспечить дальнейшую эволюцию и поддержку.

BlackBox

Компания Oberon microsystems разрабатывала компонентную библиотеку <каркас> *BlackBox Component Framework* начиная с 1992 г. (сначала библиотека называлась Oberon/F). Эта библиотека написана на Компонентном Паскале и упрощает разработку компонентов графического пользовательского интерфейса. Она поставляется с несколькими компонентами, включая текстовый редактор, систему визуального проектирования, средство доступа к базам данных SQL, интегрированную среду разработки, а также систему поддержки выполнения программ на Компонентном Паскале. Весь пакет представляет собой развитый, но весьма нетребовательный к системным ресурсам инструмент быстрой разработки компонентных приложений, названный *BlackBox Component Builder* (Блэкбокс). Он нетребователен к системным ресурсам, т.к. полностью построен из модулей Компонентного Паскаля — включая ядро со сборщиком мусора, а также самого компилятора языка Компонентный Паскаль. Это — иллюстрация как мощи концепции компонентного программного обеспечения вообще, так и адекватности языка Компонентный Паскаль в частности.

Недавно диапазон приложений системы BlackBox Component Builder был значительно расширен за счет среды кросс-программирования *Denia*, которая является компонентом, расширяющим систему Блэкбокс. Denia позволяет выполнять кросс-программирование на Компонентном Паскале для новой операционной системы реального времени *Portos*, которая тоже полностью реализована на Компонентном Паскале. Portos предназначен для встроенных систем и приложений с жесткими требованиям реального времени [hard real-time requirements], например, в робототехнике и промышленной автоматизации.

Приложение B: Отличия Компонентного Паскаля от Паскаля

Исключенные средства

• Типы-диапазоны
Используйте один из стандартных целых типов.

• Перечислимые типы
Используйте вместо них целые константы.

• Произвольные диапазоны для массивов
Массивы теперь всегда определены над целым диапазоном 0..max-1.

Пример
```
A = ARRAY 16 OF INTEGER    (* разрешены индексы из диапазона 0..15 *)
```

• Нет общих множеств
Тип SET теперь описывает набор целых чисел, который может включать элементы 0..31.

• Нет явного оператора DISPOSE
Неиспользуемая более память автоматически собирается сборщиком мусора. Вместо DISPOSE, просто присвойте переменной значение NIL.

• Нет вариантных записей
Используйте расширение (расширенное переопределение) записей.

• Нет упакованных структур
Используйте типы SHORTCHAR или BYTE для значений, умещающихся в байт.

• Нет GOTO

• Нет стандартных функций PRED и SUCC
Используйте DEC и INC для целых значений.

• Нет встроенных средств ввода/вывода
Нет файловых типов. Ввод/вывод обеспечивается библиотечными процедурами.

Измененные средства

• Стандартная процедура ENTIER вместо ROUND

• Синтаксис для констант типа REAL
3.0E+4, но не 3.0e+4

• Синтаксис для объявлений указательных типов
P = POINTER TO R
вместо
P = ^R

• Синтаксис для оператора CASE
"|" вместо ";" в качестве разделителя случаев.
Предложение ELSE.

Пример
```
CASE i * 3 - 1 OF
  0: StdLog.String("нуль")
| 1..9: StdLog.String("от единицы до девяти")
| 10, 20: StdLog.String("десять или двадцать")
ELSE StdLog.String("что-то еще")
END
```

• Имя процедуры должно быть повторено

Пример
```
PROCEDURE DrawDot (x, y: INTEGER);
BEGIN
END DrawDot;
```

• Большие и малые буквы различаются

Пример "proc" не то же самое, что "Proc".

• Синтаксис литерных цепочек
Литерные цепочки-константы заключаются между " или между '. В одной цепочке не может быть одновременно одиночных и двойных кавычек. Литерные цепочки-константы единичной длины могут присваиваться литерным переменным.

Пример
```
"That's great"    'Write "hello world" to the screen'
ch := "x"
ch := 'x'
```

• Комментарии
Комментарии заключаются между (* и *) и могут быть вложены.

• Скобки для множеств
Константы-множества задаются между { и } вместо [и].

Пример {0..2, 4, j..2 * k}

• Синтаксис функций
Используйте ключевое слово PROCEDURE также и для функций вместо FUNCTION.
Процедуры, возвращающие значение, всегда должны иметь (возможно пустой) список параметров в своих объявлениях и в вызовах.
Результат функции возвращается явно оператором RETURN, вместо присваивания имени функции.

Пример

```
PROCEDURE Fun (): INTEGER;
BEGIN
   RETURN 5
END Fun;
```

вместо

```
FUNCTION Fun: INTEGER;
BEGIN
   Fun := 5
END;

n := Fun()   вместо   n := Fun
```

• Объявления
Последовательность объявлений теперь имеет вид
{ ConstDecl | TypeDecl | VarDecl} {ProcDecl | ForwardDecl}
вместо
[ConstDecl] [TypeDecl] [VarDecl] {ProcDecl}.

Упреждающее (forward) объявление необходимо, если процедура используется до ее определения.

Пример
```
PROCEDURE ^ Proc;
```
вместо
```
PROCEDURE Proc; FORWARD;
```

• Процедурные типы
Процедуры могут быть не только переданы в качестве параметров, но и присваиваться переменным процедурных типов.

Пример
```
TYPE P = PROCEDURE (x, y: INTEGER);
VAR v: P;
v := DrawDot;    (* присваивание *)
v(3, 5);    (* вызов DrawDot(3, 5) *)
```

• Явные END вместо составных операторов
BEGIN может появляться только перед последовательностью операторов, но не внутри ее.
IF, WHILE и LOOP всегда заканчиваются ключевым словом END.

• Оператор WITH
Оператор WITH является охраной типа, действующей в соответствующем программном фрагменте, он не подразумевает наличие скрытой переменной и не открывает новый диапазон видимости для переменных.
См. детали в описании языка.

• ELSIF
Операторы IF могут иметь несколько ветвей.

Пример
```
IF name = "top" THEN
   StdLog.Int(0)
ELSIF name = "bottom" THEN
   StdLog.Int(1)
ELSIF name = " charm" THEN
   StdLog.Int(2)
ELSIF name = "beauty" THEN
   StdLog.Int(3)
ELSE
   StdLog.String("strange")
END
```

• BY вместо только DOWNTO в FOR
Циклы FOR могут использовать любое константное значение в качестве приращения (положительного или отрицательного).

Пример
```
FOR i := 15 TO 0 BY -1 DO StdLog.Int(i, 0) END
```

• Логические выражения используют «сокращенное» вычисление
Вычисление логического выражения прекращается, как только его результат определен.

Пример
Следующее выражение не вызывает ошибки при выполнении, когда p = NIL:
```
IF (p # NIL) & (p.name = "quark") THEN
```

• Константные выражения
В объявлениях констант допустимы не только буквальные константы, но и константные выражения.

Пример
```
CONST
   zero = ORD("0");
   one = zero + 1;
```

• Разные операции
используется вместо <> для проверки на неравенство.
& используется вместо AND для логической конъюнкции.
~ используется вместо NOT для логического отрицания.

• Явное преобразование к меньшему типу с помощью SHORT
Включение типов для числовых типов позволяет присваивать значения меньшего типа переменной большего типа. Присваивание в обратном направлении должно использовать стандартную процедуру SHORT.

Пример
```
int := shortint;
shortint := SHORT(int)
```

Новые средства

• Шестнадцатеричные числа и литеры

Пример
```
100H   (* десятичное 256 *)
0DX    (* возврат каретки *)
```

• Дополнительные числовые типы
Добавлены типы LONGINT, SHORTINT, BYTE, SHORTREAL.

• Симметрическая разность множеств
Множества могут вычитаться.

• Новые стандартные процедуры
Добавлены новые стандартные процедуры INC, DEC, INCL, EXCL, SIZE, ASH, HALT, ASSERT, LEN, LSH, MAX, MIN, BITS, CAP, ENTIER, LONG и SHORT.

• LOOP с EXIT

Имеется новый оператор цикла с явным операторов выхода. См. детали в сообщении о языке.

• ARRAY OF CHAR могут сравниваться
Литерные массивы могут сравниваться с помощью операций =, #, <, >, <= и >=.

• Открытые массивы, многомерные массивы
Можно определять массивы, не указывая их размера, возможно, с несколькими измерениями.

Пример
```
VAR a: POINTER TO ARRAY OF CHAR;
NEW(a, 16)

PROCEDURE ScalarProduct (a, b: ARRAY OF REAL; VAR c: ARRAY OF REAL);

TYPE Matrix = ARRAY OF ARRAY OF REAL;
PROCEDURE VectorProduct (a, b: ARRAY OF REAL; VAR c: Matrix);
```

• Разыменование указателей не обязательно
Операция разыменования ^ может быть опущена.

Пример
```
root.next.value := 5
вместо
root^.next^.value := 5
```

• Модули
Модули суть единицы компиляции, упрятывания информации, а также загрузки. Упрятывание информации -- одна из главных черт объектно-ориентированного программирования. Возможны разные уровни упрятывания информации: полное упрятывание, экспорт только для чтения/реализации, полный экспорт.
См. детали в сообщении о языке.

• Расширенное переопределение (расширение) типов
Типы записей (указательные типы) могут переопределяться, обеспечивая таким образом полиморфизм. Полиморфизм -- одно из главных средств объектно-ориентированного программирования.

• Методы
Процедуры могут быть связаны с типами записей (указательными типами), таким образом обеспечивая позднее связывание [late binding]. Позднее связывание является одним из главных средств объектно-ориентированного программирования. Такие процедуры еще называются *методами*.

• Операция с цепочками литер
Литерная цепочка, содержащаяся в литерном массиве, может быть выбрана посредством селектора $.

• Атрибуты записей
По умолчанию записи не могут быть расширены (переопределены), но могут быть помечены как EXTENSIBLE, ABSTRACT или LIMITED.

• Атрибуты методов
По умолчанию методы не могут быть расширены (переопределены), но могут быть помечены как EXTENSIBLE, ABSTRACT или EMTPY. Вновь вводимые методы должны быть помечены как NEW.