А.В. Романов

ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ В СРЕДЕ DELPHI

Учебное пособие

Воронеж 2006

ГОУВПО «Воронежский государственный
технический университет»

А.В. Романов

ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ В СРЕДЕ DELPHI

Утверждено Редакционно-издательским советом
университета в качестве учебного пособия

Воронеж 2006

УДК 681.3

Романов А.В. Основы разработки программных средств в среде Delphi: учеб. пособие. / А.В. Романов. – Воронеж: ГОУВПО «Воронежский государственный технический университет, 2006. – 183 с.

В учебном пособии рассматриваются методические и практические вопросы разработки прикладного программного обеспечения в среде Delphi.

Учебное пособие соответствует требованиям Государственного образовательного стандарта высшего профессионального образования по направлению 220200 (651900) "Автоматизация и управление", специальности 220201 (210100) "Управление и информатика в технических системах", дисциплинам "Информационное обеспечение систем управления" "Информатика", "Системное программное обеспечение", "Автоматизированные информационно-управляющие системы".

Издание может быть полезно студентам других специальностей, аспирантам и специалистам, занимающимся вопросами разработки прикладного программного обеспечения.

Учебное пособие подготовлено в электронном виде в текстовом редакторе MS Word XP и содержится в файле РПРиИП.pdf.

Табл. 6. Ил. 20. Библиогр.: 12 назв.

Научный редактор д-р техн. наук, проф. В.Л. Бурковский

Рецензенты: кафедра САПР и информационных систем Воронежского государственного технического университета (зав. каф. д-р техн. наук, проф. Я.Е. Львович);

канд. техн. наук, доц. Ю.С. Слепокуров

ВВЕДЕНИЕ

Еще в первой половине XIX в. английский математик Чарльз Бэббидж попытался построить универсальное вычислительное устройство [10]. Именно ему принадлежат идеи использования памяти и управления с помощью программы, которую предполагалось задавать посредством перфокарт. Однако работа оказалась слишком сложной для техники того времени. В 40-годах XX в. сразу несколько групп исследователей повторили попытку разработки универсальной вычислительной машины. Например, немецкий инженер Конрад Цузе в 1941 г. построил небольшой компьютер на основе электромеханических реле, а в 1943 г. на одном из предприятий фирмы IBM американец Говард Эйкен создал более мощный компьютер под названием Марк-1, который позволял проводить вычисления в сотни раз быстрее, чем с помощью арифмометра, и реально использовался для военных расчетов.

Начиная с 1943 г. в США группа специалистов под руководством Джона Мочли и Преспера Экерта начала конструировать компьютер ENIAC на основе электронных ламп. Созданный компьютер работал в тысячу раз быстрее, чем Марк-1, но обнаружилось, что большую часть времени он простаивал по причине того, что для задания метода расчетов (программы) приходилось в течение нескольких часов или дней подсоединять нужным образом провода, хотя сам расчет занимал всего лишь несколько минут или секунд. Очевидной стала необходимость хранения управляющей программы непосредственно в памяти компьютера.

Мочли и Экерт в 1945 г. привлекли к работе над проектом знаменитого математика Джона фон Неймана, и сформулированные им общие принципы функционирования универсальных вычислительных устройств (компьютеров) актуальны и по сей день. Согласно им, компьютер должен иметь следующие устройства:

- *арифметическо-логическое устройство,* выполняющее арифметические и логические операции;
- *устройство управления,* которое организует процесс выполнения программ;
- з*апоминающее устройство,* или *память* для хранения программ и данных, которая должна состоять из некоторого количества одинаково легко доступных для других устройств пронумерованных ячеек, в каждой из которых могут находиться или обрабатываемые данные, или инструкции программ;
- *внешние устройства* для ввода-вывода информации.

Первый компьютер, реализующий *принципы фон Неймана*, был построен в 1949 г. английским исследователем Морисом Уилксом.

Общеизвестно определение компьютера как *универсального устройства для переработки информации* [4, 10]. Но сам по себе компьютер является просто ящиком с набором электронных схем. Он не обладает знаниями ни в одной предметной области своего применения. Все эти знания сосредоточены в выполняемых на компьютере программах.

Иными словами, для осуществления определенных действий компьютеру необходима программа, то есть *точная и подробная последовательность инструкций обработки данных на понятном компьютеру языке*. Часто употребляемое выражение «компьютер сделал» (подсчитал, нарисовал) означает ровно то, что на компьютере была выполнена программа, которая позволила совершить соответствующее действие. Меняя программы для компьютера, можно превратить его в рабочее место инженера, бухгалтера, агронома, редактировать на нем документы или управлять технологическим процессом. Для эффективного использования компьютера необходимо знать назначение и свойства используемых программ.

Программирование или *умение разработать понятную компьютеру последовательность инструкций обработки данных* (говоря короче – умение написать программу), как следует

из истории появления компьютеров, изначально было связано с механикой, затем с электротехникой, и только после появления электронной машины, основанной на принципах фон Неймана, можно говорить о становлении программирования как самостоятельной научной области. Возникнув сравнительно недавно, программирование на данный момент является наукой, способной решить актуальнейшие и важнейшие задачи современного мира.

Первые программы разрабатывались непосредственно на языке процессора в *машинных кодах*. Стремление ускорить процесс программирования, повысить качество и функциональность разработанного программного обеспечения, адаптировать язык программирования к человеческому мышлению – все это привело к появлению языков программирования высокого уровня. В настоящее время именно они являются наиболее востребованными при общении человека и ЭВМ.

Одному из вопросов теории программирования, а именно – объектно-ориентированному программированию на языке высокого уровня – посвящено данное учебное пособие.

В основу содержательной части были положены требования Государственного образовательного стандарта высшего профессионального образования Российской Федерации по направлению 220200 (651900) "Автоматизация и управление" [11], также была использована примерная программа дисциплины "Программирование на языке высокого уровня", составленная на кафедре "Компьютерные системы и технологии" Московского Государственного Инженерно-физического Университета [12].

Структурно пособие построено следующим образом.

В **первой** главе в общем виде рассмотрены вопросы функционирования электронно-вычислительных устройств и изложены связанные с этим особенности разработки прикладного и информационного программного обеспечения (ПО) на языке высокого уровня.

Во **второй** главе рассмотрен жизненный цикл программных продуктов, спиральная и каскадная модели разработки ПО, общие вопросы методологии и технологии, а также инструментальные CASE-средства этого процесса.

Третья глава посвящена описанию языка Object Pascal. Рассмотрены стандартные типы данных, общая структура программ, операторы и обработка исключительных ситуаций.

В **четвертой** главе приведено описание основных возможностей и настроек интегрированной среды программирования Delphi, в частности, основных команд главного меню, инспектора объектов, редактора кода. Рассмотрены некоторые дополнительные настройки.

Основные элементы построения интерактивного интерфейса прикладных программ рассматриваются в **пятой** главе. В частности, рассмотрены формы и фреймы, компоненты многостраничного интерфейса, визуальные и невизуальные элементы управления, отображение текста и структурированных данных, элементы построения баз данных и управления конфигурацией проектируемого программного обеспечения, компоненты работы с графикой.

1. ПРОЕКТИРОВАНИЕ ПРИКЛАДНЫХ ПРОГРАММ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

1.1. Особенности разработки программного обеспечения на языке высокого уровня

1.1.1. Функциональные принципы работы компьютера

Для понимания специфики программирования, в том числе и на языках высокого уровня, рассмотрим в общих чертах работу компьютера. Вначале с какого-либо внешнего устройства в память компьютера загружается программа. Устройство управления считывает содержимое предопределенной ячейки памяти, в которой находится первая инструкция (команда) программы, и организует ее выполнение. Команды могут задавать выполнение арифметических или логических операций, чтение или запись данных, ввод данных из внешнего устройства в память или вывод данных из памяти на внешнее устройство. Как правило, после выполнения одной команды устройство управления начинает выполнять команду из ячейки памяти, находящейся непосредственно за выполненной командой. Этот порядок может быть изменен с помощью команд *перехода* (передачи управления), т.е. указания устройству управления продолжить выполнение программы с команды, содержащейся в другой ячейке памяти. Такой *скачок* может выполняться при определенных условиях, например, при равенстве *операндов* или если в результате предыдущей арифметической операции получился нуль и т.д. Это позволяет использовать одни и те же последовательности команд в программе много раз (организовывать *циклы*), выполнять различные последовательности команд в зависимости от выполнения некоторых условий и т.д. – другими словами, разрабатывать сложные программы.

Поясним значения некоторых терминов.

Оператором в языке программирования называют предписание, предназначенное для задания некоторого завершенного действия в процессе переработки информации на ЭВМ.

Операнд – это величина, представляющая собой объект операции, реализуемой ЭВМ в ходе выполнения программы вычислений. Например, операндами арифметических операций обычно являются числа: при сложении – слагаемые, при умножении – сомножители.

Современные компьютеры устроены таким образом, что арифметическо-логическое устройство и устройство управления, как правило, объединены в единое устройство – *центральный процессор.* Кроме этого, процесс выполнения программ может прерываться для выполнения *прерываний* – неотложных действий, связанных с поступившими сигналами от внешних устройств. Процессор выполняет инструкции программы автоматически, без вмешательства человека. Он обменивается информацией с оперативной памятью и внешними устройствами, осуществляет ввод-вывод полученных результатов на внешние устройства и может приостанавливать выполнение программы до завершения операции ввода-вывода. По завершению всех необходимых действий компьютер переходит в режим ожидания сигналов от внешних устройств.

Классической архитектуре фон Неймана с точки зрения программирования присущи следующие особенности:

1) единственная линейная последовательно адресуемая память, программа и данные соответственно хранятся в одной памяти, а адреса областей памяти составляют последовательность типа 0, 1, 2,…, т.е. память является одномерной и имеет вид вектора слов;

2) отсутствует явное различие между командами и данными, причем назначение данных не является их неотъемлемой составной частью, например, нет никаких средств, позволяющих явно отличить набор битов для числа с плавающей точкой от набора битов, являющихся строкой символов.

В настоящее время различают два типа построения персональных ЭВМ с архитектурами *CISC* и *RISC*.

Архитектура *CISC* (*Complete Instruction Set Computer*) является практическим стандартом для рынка микрокомпьютеров. Для CISC-процессоров характерно сравнительно небольшое число регистров общего назначения; большое количество машинных команд, некоторые из которых семантически нагружены аналогично операторам высокоуровневых языков программирования и выполняются за много тактов; большое количество методов адресации и форматов команд различной разрядности; преобладание двухадресного формата команд; наличие команд обработки типа регистр-память.

Для современных рабочих станций и серверов основой является архитектура компьютера с сокращенным набором команд (*RISC – Reduced Instruction Set Computer*). Главная идея разработчиков этих машин – отделение медленной памяти от высокоскоростных регистров. Система команд разрабатывалась таким образом, чтобы выполнение любой команды занимало небольшое количество машинных тактов (предпочтительно один машинный такт). Логика выполнения команд с целью повышения производительности ориентируется на аппаратную, а не на микропрограммную реализацию. Для упрощения логики декодирования команд используются команды фиксированной длины и фиксированного формата.

1.1.2. Понятие о низкоуровневом программировании

Из вышеприведенного материала становится ясна технология разработки программ на языках низкого уровня – непосредственно *в машинных кодах* (как правило, в шестнадцатеричных числах) или на более удобном языке программирования – *ассемблере*, в котором числовой код команды заменен некоторым буквенным сокращением, *мнемокодом* (например, команду перехода можно обозначить *goto* или, от англ. jump – прыжок, – *JMP*), а числа (или *операнды*) сохраняют свою шестнадцатеричную форму представления. Очевидны и неудоб-

ства такого программирования – общая логика программы абсолютно непрозрачна даже для разработчика, поэтому процесс создания достаточно больших программ очень трудоемок. Но существуют и преимущества – программы, написанные таким образом, отличаются минимальным количеством программного кода и максимальным быстродействием.

Как правило, при разработке серьезного программного обеспечения минимум одну или более *подпрограммы* (они носят название *процедур* или *функций*) приходится писать на низкоуровневом языке программирования. Знание ассемблера также очень желательно при отладке ПО. Но, вообще говоря, для разработки программ на языках высокого уровня знание ассемблера не является обязательным. Отметим только, что это очень интересный материал, которому посвящено большое количество научно-технической литературы, например [7]. Но в данном учебном пособии вопросы низкоуровневого программирования не рассматриваются.

Продолжим исследование предпосылок возникновения программирования на языках высокого уровня. Уже использование мнемокодов потребовало дополнительных программных средств – программы, которая как бы "переводила" команды ассемблера непосредственно в машинные коды. Этот процесс получил название *компиляции*, а подобная программа соответственно была названа *компилятором*. Для языков высокого уровня компилятор – очень важная часть общего пакета среды программирования. С его помощью создаются так называемые *исполняемые* файлы программного обеспечения, которые обычно имеют расширение "exe". Существуют также *библиотеки кода* или библиотеки подпрограмм, обычно имеющие расширение "*dll*" (или специфичные для Delphi *bpl*-файлы). Именно в исполняемых файлах и библиотеках хранится код программы, т.е. *алгоритмы* тех операций, благодаря которым программные средства реализуют какие-либо действия, требующиеся пользователю.

Алгоритмом называют способ решения какой-либо задачи (не обязательно вычислительной), точно предписывающий, как и в какой последовательности получить результат, однозначно определяемый исходными данными.

Существует и другой вариант работы программного обеспечения. Пусть в памяти машины хранится программа на языке высокого уровня, которая выполняется по отдельным операторам (группам операторов). Такую программу удобно представить в виде листинга, где каждый оператор расположен на отдельной строке, а строки пронумерованы. Тогда можно сказать, что программа выполняется построчно. Вместо компилятора в этом случае необходим *интерпретатор*, выполняющий те же функции преобразования операторов в машинный код. Вообще операции построчного выполнения программы необходимы для отладки ПО и при выявлении ошибок. Принципиальная разница между компиляцией и интерпретированием в том, что при компилировании создается как минимум один дополнительный исполняемый *exe*-файл программы. Интерпретируемый программный код выполняется существенно медленнее.

1.1.3. Основные языки программирования высокого уровня

Соответственно и языки программирования высокого уровня разделяют на *компилируемые* и *интерпретируемые*. Не останавливаясь подробно на истории развития языков программирования высокого уровня, можно отметить три, ставших де-факто основными языковыми системами – Бейсик (Basic), Паскаль (Pascal) и Си (C).

Принципиально все языки программирования высокого уровня похожи. Основными элементами каждого являются некоторый список *зарезервированных* (служебных) слов и определенные правила записи операторов и операндов. Зарезервированные слова – это в основном *операторы*, *инструкции* и *элементы структурирования программы*. Исторически сло-

жилось так, что в этом качестве используются слова английского языка, а также некоторые знаки пунктуации. Правила записи операторов и операндов лучше всего воспринимать как *грамматические* правила – в том смысле, что строка программы должна быть построена таким и только таким образом. В противном случае компьютер "не поймет" программу, т.е. она с языка программирования высокого уровня или не будет переведена в машинный код, или это преобразование будет осуществлено некорректно, т.е. разработанное программное обеспечение окажется неработоспособным.

С точки зрения электронно-вычислительной машины безразлично, какой из языков программирования используется. "Понятность" языка для ЭВМ определяется компилятором, а основным фактором, определяющим "понятность" или "приемлемость" языка программирования, является человеческое мышление, т.е. язык программирования высокого уровня в первую очередь должен быть понятен и близок использующему его человеку – программисту.

Первым из языков программирования высокого уровня был разработан Бейсик. Изначально это был интерпретируемый язык, отличающийся простотой и понятностью. Несомненно, многие люди, связанные с программированием (включая и автора данного учебного пособия), будут благодарны разработчикам этого языка за те новые возможности (по сравнению с программированием в машинных кодах), реализацию которых он обеспечивал. Но его достоинства – простота и понятность – с течением времени оказались и его недостатками настолько, что программирование на Бейсике стало ассоциироваться с чем-то примитивным и несерьезным. Авторы языка (компания Microsoft) пытались с этим бороться, разрабатывали различные версии, сделали язык компилируемым, но в конечном итоге были вынуждены признать его "мертвым" языком программирования и разработать на его основе Visual Basic, который мало чем отличается от версии Quick Basic, и, сохра-

няя простоту и понятность, имеет тем не менее существенные ограничения, и едва ли может быть рекомендован для разработки серьезных программных продуктов.

Вторым фундаментальным языком программирования высокого уровня стал Паскаль. Фирма Borland (ныне Inprise) изначально ставила задачу разработать альтернативный Бейсику язык, компилируемый, ориентированный на создание структурно сложных программных средств. И ей это в полной мере удалось. Современный объектно-ориентированный вариант Паскаля – Object Pascal – абсолютно заслуженно снискал наибольшую популярность как у начинающих, так и у опытных программистов. Синтаксис языка прост и ясен, возможности практически ограничены имеющимися ресурсами. К тому же это строго *типизированный* язык программирования, что позволяет повысить скорость разработки ПО и уменьшить число ошибок. Примеры и рекомендации для разработки программных средств, которые в данном учебном пособии будут приведены далее, ориентированы на среду проектирования Delphi, основу которой составляет именно Object Pascal.

Единственное, чего по-настоящему не хватало в Паскале – это свободы, свободы для программистов реализовывать свои собственные (иногда абсолютно фантастические) замыслы, программные структуры и т.п. Именно желание уйти от строгой типизации данных, от раз и навсегда оговоренной грамматики привели к разработке языка Си и его современного варианта C++. Программы, написанные на нем, отличаются большой эффективностью. Язык поддерживается ведущими лидерами в области разработки ПО (Microsoft и Inprise), широко распространен, доступен на всех платформах. Но с другой стороны, его эффективное использование требует от программиста очень высокой профессиональной подготовки и некоторой нестандартности мышления. По оценкам специалистов [8], скорость разработки приложений на языке Си в несколько раз

меньше, чем при использовании более простых языков программирования Object Pascal и Visual Basic.

1.1.4. Процедурное и событийное программирование

Вернемся к функциональным принципам работы компьютера. Было отмечено, что программа, начиная выполняться с определенной ячейки памяти, выполняется последовательно по управляющим инструкциям до своего завершения. Другими словами, программа является совокупностью последовательно выполняемых операций (процедур) и может рассматриваться как *код, воздействующий на данные*. Для программиста это означает следующее: любые инструкции программы будут выполняться по ходу программы немедленно и именно в последовательности, определенной при программировании. Такая модель построения программы называется *процессорно-ориентированной*, а языки программирования, реализующие подобный подход – *процедурными* [8].

Еще одной особенностью процессорно-ориентированной модели является то, что параллельная работа двух разных программ невозможна. До того, как на выполнение будет запущена следующая программа (фактически это означает загрузку в оперативную память), предыдущая должна завершиться. Такой вариант работы поддерживается, например, в операционной среде MS DOS (Microsoft Disk Operating System) и носит название *однозадачного* режима работы.

Стиль разработки ПО в рамках процессорно-ориентированной модели получил много различных определений – *"процедурное программирование*", "*программирование под MS DOS*", "*линейное программирование*", "*работа в режиме командной строки*" (в том смысле, что командная инструкция выполняется сразу после ввода). Программирование в подобном стиле достаточно просто и доступно для понимания, программисту не требуется пространственно-временного мышления. С другой стороны, уровень подавляющего большинства

программ, разработанных таким образом, в настоящее время едва ли можно считать удовлетворительным.

Переход к современному стилю программирования – *событийному* – осуществлялся постепенно. В работе Ч. Петзолда "Программирование для Windows 95" приведено сравнение, что программисты, работающие в рамках MS DOS, сами "вырыли себе яму" использованием *резидентных* программ – программ, которые все время находились в памяти компьютера, но не проявляли себя до определенных действий пользователя. (Самым типичным примером является программа, выполняющая копию экрана при нажатии определенного сочетания клавиш, допустим "Alt + W").

В приведенном примере факт поступления в центральный процессор кода нажатой пользователем клавиши можно назвать *событием*, и это неплохая иллюстрация принципов событийного программирования. Понаблюдайте внимательно: что делает большинство программ при их активации в операционной системе Windows (да, пожалуй, и в любой другой современной операционной системе)? Они рисуют свое окно, меню, панели управления, могут открыть и показать некоторый заданный файл и… И все! Далее они ожидают команд (действий) пользователя – необходимо нажать какую-либо кнопку, выбрать пункт меню, кликнуть где-нибудь мышкой – другими словами, для дальнейшей работы программе необходимо возникновение события. Реакцией на каждое событие будет отработка соответствующего программного кода.

Такая модель построения программы называется *объектно-ориентированной* в том смысле, что программа рассматривается как совокупность фрагментов кода (объектов) [8]. Программист при разработке подобной программы даже с малой долей вероятности не в состоянии предсказать однозначную последовательность выполнения кода программы. Объектно-ориентированные программы можно охарактеризовать как *данные, управляющие доступом к коду*. Если добавить

сюда возможность параллельной работы нескольких приложений (*многозадачный* режим), графический интерфейс, использование одного и того же программного кода несколькими приложениями и многие другие возможности – то преимущества событийной функциональности приложения в современных операционных системах не подлежат сомнению. Однако такой стиль программирования требует значительно большей квалификации программиста.

1.1.5. Технология быстрой разработки приложений

Совместно со словосочетанием "событийное программирование" используется термин *визуальное программирование* или *технология RAD* (Rapid Application Development – быстрая разработка приложений). Все это означает разработку ПО в специальной инструментальной среде и основывается на визуализации процесса создания программного кода. Средства быстрой разработки приложений основываются на *компонентной* архитектуре. При этом *компоненты* являются объектами, объединяющими *данные*, *свойства* и *методы*. Компоненты могут быть как *визуальными*, так и *невизуальными*; *атомарными* и *контейнерными* (содержащими другие компоненты); *низкоуровневыми* (системными) и *высокоуровневыми*.

При визуальном проектировании пользователю предоставляется возможность выбора необходимых компонентов из некоторого набора (*палитры*) с последующим заданием их свойств. Для обозначения инструментов визуального проектирования используется широкий набор терминов [8], включающих: *конструктор компоновки, конструктор форм, визуальный редактор, проектировщик экрана, проектировщик форм, конструктор графического пользовательского интерфейса* и т.д. Процедура разработки интерфейса средствами RAD сводится к набору последовательных операций, включающих:

- размещение компонентов интерфейса в нужном месте;
- задание моментов времени их появления на экране;
- настройку связанных с ними атрибутов и событий.

Отметим, что эффективность визуального программирования определяется не столько наличием самих визуальных компонентов, сколько их взаимосвязью и взаимодействием традиционными средствами. Даже если среда программирования не содержит достаточного количества требуемых компонентов, она все равно будет востребована, если позволяет самостоятельно разрабатывать необходимые компоненты или использовать имеющиеся средства сторонних производителей, альтернативные отсутствующим в ней.

Другими словами, технология быстрой разработки программных средств основывается на интегрированной среде программирования, с помощью которой выполняются процессы проектирования, отладки и тестирования прикладных программных продуктов.

1.1.6. Классификация программных средств

Программы, работающие на компьютере, можно условно разделить на три категории [10]:

- *прикладные программы*, непосредственно обеспечивающие выполнение необходимых пользователю действий: расчетов, редактирование текста и графики, просмотр и обработку массивов информации и т.д.;
- *системные программы*, выполняющие различные вспомогательные функции, например, создание копий используемой информации или проверку работоспособности устройств компьютера. Особую роль среди всех системных программ играет *операционная система* – программа, управляющая компьютером, запускающая все другие программы и выполняющая для них различные сервисные функции;
- *инструментальные системы* (системы программирования), обеспечивающие возможность разработки новых программных средств.

Из этих категорий, согласно тематике данного учебного пособия, наиболее интересна первая. Прикладные программы за полвека своего развития прошли путь от выполнения эле-

ментарных логических и арифметических действий до самых сложных систем автоматизации инженерных и экономических расчетов, управления технологическими процессами и промышленными предприятиями. В прикладном ПО всегда можно было выделить два основных направления:

- выполнение вычислений;
- накопление и обработка информации.

Реализация второго направления привела к выделению технологии разработки *информационных* систем практически в самостоятельную область программирования [8]. Такое ПО предназначено для сбора, хранения и обработки различных информационных структур. В основе, как правило, лежит определенная среда работы с данными (СУБД – система управления *базой данных*). Ориентированы подобные программы на пользователя, не обладающего высокой квалификацией в области применения ЭВМ. Поэтому серьезное внимание приходится уделять надежности работы программы, доступности и понятности интерфейса пользователя, а также дополнительной разработке *информационного обеспечения* или возможности визуализации информации о стандартах, классификаторах, тенденциях, методиках и т.п. – словом, о предметной области, на которую ориентировано разработанное ПО [3]. Последнее роднит информационные системы со стандартным прикладным расчетным ПО, которое, вообще говоря, также не предполагает эксплуатацию пользователем, квалифицированным в области применения вычислительной техники.

Отметим, что разработка информационных систем предполагает изучение работы с СУБД, что является достаточно обширной и сложной темой, достойной самостоятельного изложения в рамках соответствующего учебного курса [6].

1.2. Основные фазы проектирования программных продуктов

1.2.1. Определение проекта и анализ процесса проектирования с позиций теории управления

Любой программный продукт всегда разрабатывается как некоторый *проект*. Многие особенности управления проектами и *фазы разработки проекта* (*фазы жизненного цикла*) являются общими, не зависящими не только от предметной области, но и от характера проекта. Для сложного понятия (в частности, для проекта) трудно дать однозначное и исчерпывающее определение. Согласно [8], будем руководствоваться следующей формулировкой.

Проект – это ограниченное по времени целенаправленное изменение отдельной системы с изначально четко определенными целями, достижение которых определяет завершение проекта, а также с установленными требованиями к срокам, результатам, риску, рамкам расходования средств и ресурсов и к организационной структуре.

Исходя из данного определения, проект можно рассматривать как некий динамический объект (рис. 1.1). Эффективность выполнения проекта достигается путем управления процессом выполнения необходимых работ, координацию их по-

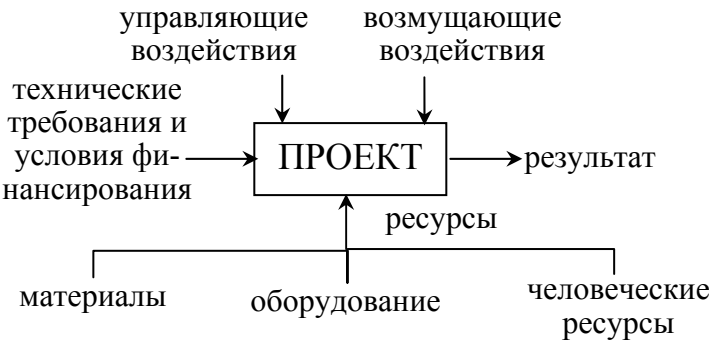


Рис. 1.1.

следовательности, распределения ресурсов и компенсации внутренних и внешних возмущений.

С точки зрения теории управления, объект должен быть *наблюдаем* (необходимо постоянно контролировать ход выполнения проекта) и *управляем* (т.е. необходимы механизмы своевременного воздействия на ход реализации проекта). Последнее особенно актуально в условиях неопределенности и изменчивости предметной области, что характерно при разработке программных продуктов.

Выделим основные отличительные признаки проекта как объекта управления:

- изменчивость – целенаправленный перевод системы из существующего в некоторое желаемое состояние, выраженное в целях проекта;
- ограниченность конечной цели и ограниченность продолжительности проекта во времени;
- ограниченность бюджета и ресурсов;
- комплексность – наличие большого числа факторов, прямо или косвенно влияющих на прогресс и результаты выполнения проекта;
- новизна для структуры, заказавшей проект.

1.2.2. Классификация проектов

Проекты можно классифицировать по самым различным признакам. Отметим основные из них.

По составу и структуре проекта:

- монопроект (отдельный проект любого типа, вида и масштаба);
- мультипроект (комплексный проект, состоящий из ряда монопроектов и требующий использования многопроектного управления процессом разработки).

Тип проекта можно определять, например, по предметной области: технический, организационный, экономический, социальный, смешанный. Отметим, что разработка прикладного ПО (в том числе и информационных систем) относится к

техническим проектам, которые, согласно [8], имеют следующие особенности:

- главная цель проекта четко определена, но отдельные цели должны уточняться по мере достижения частных результатов;
- срок завершения и продолжительность проекта определены заранее, желательно их точное соблюдение, однако они также могут корректироваться в зависимости от полученных промежуточных результатов и общего прогресса.

Масштаб проекта определяется по размерам бюджета и количеству участников: мелкие проекты, малые проекты, средние проекты и крупные проекты. Можно также рассматривать масштабы проектов в более конкретной форме – отраслевые, корпоративные, учебные, ведомственные проекты, проекты одного предприятия и т.д.

1.2.3. Основные фазы проектирования

Каждый проект, независимо от сложности и объема работ, проходит в своем развитии определенные *фазы*, *стадии*, *этапы* – от возникновения идеи до полного завершения проекта [8]. Выделим в общем виде следующие фазы развития прикладного расчетного и информационного ПО:

- формирование концепции;
- разработка технического задания;
- проектирование;
- разработка или изготовление;
- ввод программных средств в эксплуатацию.

Начальные фазы проекта, как правило, имеют существенное влияние на достигаемый результат. В них принимаются основные проектные решения. По экспертным оценкам специалистов, обычно 30 % вклада в конечный результат проекта вносят фазы концепции и технического задания; по 20 % – фазы проектирования и разработки, 30 % – фаза завершения проекта. Вторую и частично третью фазы принято называть *фаза-*

ми системного проектирования, а последние две (иногда совместно с фазой проектирования) – *фазами реализации*.

Главным содержанием работ на первоначальной *концептуальной* фазе проектирования являются:

- формирование идеи, постановку целей;
- формирование ключевой команды проекта;
- изучение мотивации и требований заказчика и других участников;
- сбор исходных данных и анализ существующего состояния;
- определение основных требований и ограничений, требуемых материальных, финансовых и трудовых ресурсов;
- сравнительную оценку альтернативных вариантов решения задачи;
- представление предложений, их экспертизу и утверждение.

Главной задачей фазы *технического задания* является разработка такого *технического предложения*, которое бы в переговорах с заказчиком плавно и непротиворечиво становилось техническим заданием. Здесь имеется в виду, что заказчик, как правило, ожидает абсолютной адекватности предметной области и разрабатываемого ПО, а также полной функциональности при любом наборе исходных данных. Поэтому очень важно на этом этапе определить границы применения разработанной программы и четко оговорить ее функциональные и сервисные возможности. Общее содержание работ этой фазы:

- разработка основного содержания и базовой структуры проекта;
- разработка и утверждение технического задания;
- планирование и декомпозиция базовой структурной модели проекта;
- составление сметы и бюджета проекта, определение потребности в ресурсах;

- разработка календарных планов и укрупненных графиков выполнения работ;
- подписание контракта с заказчиком;
- ввод в действие средств коммуникации участников проекта и контроля за ходом работ.

Среди работ этой фазы стоит выделить определение границ применения разработанного ПО, списка и текста сообщений о внештатных ситуациях.

На фазе *проектирования* формируются основные информационные структуры, определяются подсистемы и их взаимосвязи, выбираются наиболее эффективные способы выполнения проекта и использования ресурсов. Характерные работы на этой фазе:

- выполнение базовых проектных работ;
- разработка частных технических заданий;
- выполнение концептуального проектирования;
- составление технических спецификаций и инструкций;
- представление проектной разработки, экспертиза и утверждение;

Фаза *разработки*, пожалуй, наиболее трудоемкая стадия проектирования. Здесь осуществляется разработка запланированных подсистем, их объединение и тестирование, а также координация и оперативный контроль выполнения проектных работ. Основное содержание:

- выполнение работ по разработке программного обеспечения;
- выполнение подготовки к внедрению системы;
- контроль и регулирование основных качественных показателей проекта.

На завершающей фазе *ввода в эксплуатацию* проводятся испытания, опытная эксплуатация системы в реальных условиях, ведутся переговоры о результатах выполнения проекта и о возможных новых контрактах. Основные виды работ:

- комплексные испытания;
- подготовка кадров для эксплуатации разработанной системы;
- подготовка рабочей документации, сдача системы заказчику и ввод ее в эксплуатацию;
- сопровождение, поддержка, сервисное обслуживание;
- оценка результатов проекта и подготовка итоговых документов;
- разрешение конфликтных ситуаций и закрытие работ по проекту;
- накопление опытных данных для последующих проектов, анализ опыта и определение направлений развития.

Кроме того, необходимо отметить, что на обнаружение ошибок, допущенных на стадии системного проектирования, расходуется примерно в два раза больше времени, чем на последующих фазах, а их исправление обходится в пять раз дороже. Поэтому на начальных стадиях проекта разработку следует выполнять особенно тщательно. Наиболее часто на начальных фазах допускаются следующие ошибки [8]:

- неправильная интерпретация исходной постановки задачи;
- ошибки в определении интересов заказчика;
- концентрация на маловажных, сторонних интересах;
- неправильное или недостаточное понимание деталей;
- неполнота функциональных спецификаций (системных требований);
- ошибки в определении требуемых ресурсов и сроков;
- редкая проверка на согласованность этапов и отсутствие контроля со стороны заказчика (нет привлечения заказчика).

2. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНЫХ ПРОДУКТОВ, МЕТОДОЛОГИЯ И ТЕХНОЛОГИЯ РАЗРАБОТКИ

2.1. Процессы жизненного цикла

2.1.1. Структура жизненного цикла по стандарту ISO/IEC 12207

Понятие *жизненного цикла* является одним из базовых понятий методологии проектирования. Жизненный цикл для программных продуктов является непрерывным процессом. Он начинается с момента принятия решения о создании ПО и заканчивается в момент полного изъятия программы из эксплуатации. Для информационных систем существует международный стандарт, регламентирующий их жизненный цикл – ISO/IEC 12207 (ISO – International Organization of Standardization – международная организация по стандартизации; IEC – International Electrotechnical Commission – международная комиссия по электротехнике).

Согласно данному стандарту структура жизненного цикла основывается на трех группах процессов:

- *основные* процессы жизненного цикла (приобретение, поставка, разработка, эксплуатация, сопровождение);
- *вспомогательные* процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, разрешение проблем);
- *организационные* процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого жизненного цикла, обучение).

2.1.2. Основные процессы

Среди основных процессов жизненного цикла наибольшую важность имеют три: *разработка*, *эксплуатация* и *сопровождение*.

Разработка является одним из важнейших процессов жизненного цикла и, как правило, включает в себя стратегическое планирование, анализ, проектирование и реализацию (собственно программирование). Непосредственно процесс *разработки* включает в себя все работы по созданию ПО и его компонентов в соответствии с заданными требованиями, а также оформление проектной и эксплуатационной документации; подготовку материалов, необходимых для проведения тестирования разработанных программных продуктов; и – по необходимости – разработку материалов, необходимых для организации обучения пользователей.

Эксплуатационные работы можно подразделить на *подготовительные* и *основные*. К *подготовительным* относят конфигурирование базы данных и рабочих мест пользователей; обеспечение пользователей эксплуатационной документацией и обучение персонала. *Основные эксплуатационные* работы включают непосредственно эксплуатацию ПО; локализацию проблем и устранение причин их возникновения; модификацию ПО; подготовку предложений по совершенствованию системы; развитие и модернизацию системы (последние три позиции – по необходимости).

Процесс *сопровождения* программной продукции (службы технической поддержки) играет весьма заметную роль в жизни любого крупного программного продукта. Отметим, что наличие квалифицированного технического обслуживания на этапе эксплуатации является необходимым условием для решения поставленных задач, причем ошибки обслуживающего персонала могут приводить к явным или скрытым финансовым потерям, сопоставимым со стоимостью всей системы. Обеспечение качественного технического обслуживания требует привлечения специалистов высокой квалификации, которые в состоянии решать не только текущие задачи администрирования, но и восстанавливать работоспособность системы при сбоях.

2.1.3. Вспомогательные и организационные процессы

Среди *вспомогательных* процессов одно из главных мест занимает *управление конфигурацией*. Это необходимо прежде всего для разработки и сопровождения ПО. При разработке сложных проектов, состоящих из многих компонентов, каждый из которых может разрабатываться независимо и, следовательно, иметь несколько вариантов реализации и (или) несколько версий одной реализации, возникает проблема учета их связей и функций, создания единой структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовывать, систематически учитывать и контролировать внесение изменений в различные компоненты прикладного ПО на всех стадиях жизненного цикла.

Организационные процессы управления проектом связаны с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством. Среди технического и организационного обеспечения проекта можно выделить следующее:

- выбор методов и инструментальных средств для реализации проекта;
- определение методов описания промежуточных состояний разработки;
- разработку методов и средств испытаний созданных программных средств;
- обучение персонала.

Обеспечение качества ПО связано с проблемами *верификации*, *проверки* и *тестирования* компонентов.

Верификация – это процесс определения соответствия текущего состояния разработки, достигнутого на данном этапе, требованиям этого этапа.

Проверка – это процесс определения соответствия параметров разработки исходным требованиям.

Проверка отчасти совпадает с *тестированием*, которое проводится для определения различий между действительны-

ми и ожидавшимися результатами и оценки соответствия характеристик разработанного ПО исходным требованиям.

2.2. Модели жизненного цикла

Моделью жизненного цикла будем называть некоторую структуру, определяющую последовательность осуществления процессов, действий и задач, выполняемых на протяжении жизненного цикла прикладного ПО, а также взаимосвязи между этими процессами, действиями и задачами.

К настоящему времени наибольшее распространение получили две основные модели [8]:

- *каскадная* модель, иногда также называемая моделью "водопад" (waterfall);
- *спиральная* или *итерационная* модель.

2.2.1. Каскадная модель

Каскадная модель демонстрирует классический подход к разработке различных систем в любых прикладных областях. В ней предусматривается последовательная организация работ. Основной особенностью является деление всей разработки на этапы, причем переход с одного этапа на следующий происходит только после полного завершения всех работ на предыдущем этапе. Кроме того, каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

За десятилетия существования каскадной модели деление работ на стадии и названия этих стадий менялись. Тем не менее можно выделить ряд устойчивых этапов разработки, практически не зависящих от предметной области (рис. 2.1):

- анализ требований заказчика;
- проектирование;
- разработка;
- тестирование и опытная эксплуатация;
- сдача готового продукта.

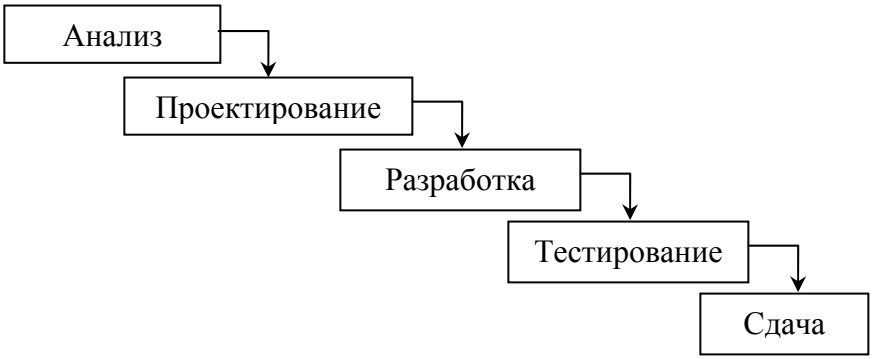


Рис. 2.1.

На этапе *анализа* проводится исследование проблемы, четко формулируются все требования заказчика. Результатом является техническое задание (задание на разработку), согласованное со всеми заинтересованными сторонами.

На этапе *проектирования* разрабатываются проектные решения, удовлетворяющие всем требованиям, сформулированным в техническом задании. Результатом является комплект проектной документации, содержащей все необходимые данные для реализации проекта.

Третий этап – *реализация* проекта. Здесь осуществляется разработка ПО в соответствии с проектными решениями предыдущего этапа. Методы, используемые для реализации, не имеют принципиального значения. Результатом выполнения данного этапа является готовый программный продукт.

На этапе *тестирования* проводится проверка полученного ПО на предмет соответствия требованиям, заявленным в техническом задании. Опытная эксплуатация позволяет выявить различного рода скрытые недостатки, проявляющиеся в реальных условиях работы.

Последний этап – *сдача* готового проекта. Главная задача этого этапа – убедить заказчика, что все его требования реализованы в полной мере.

Этапы работ в рамках каскадной модели часто называют частями *проектного цикла* системы. Такое название возникло

потому, что этапы состоят из многих итерационных процедур уточнения требований к системе и вариантов проектных решений. Жизненный цикл самой системы существенно сложнее и больше. Он может включать в себя произвольное число циклов уточнения, изменения и дополнения уже принятых и реализованных проектных решений. В этих циклах происходит развитие системы и модернизация отдельных ее компонентов.

Каскадная модель имеет ряд положительных сторон, благодаря которым она хорошо зарекомендовала себя при выполнении различного рода инженерных разработок и получила широкое распространение. Рассмотрим основные достоинства:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности. На заключительных этапах также разрабатывается пользовательская документация, охватывающая все предусмотренные стандартами виды обеспечения информационной системы: организационное, методическое, информационное; программное, аппаратное;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения и соответствующие затраты.

Каскадная модель изначально разрабатывалась для решения различного рода инженерных задач и не потеряла своего значения для данной прикладной области до настоящего времени. Кроме того, каскадный подход хорошо зарекомендовал себя и при построении определенных информационных систем – для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем, чтобы предоставить разработчикам свободу выбора реализации, наилучшей с технической точки зрения. К таким информационным системам, в частности, относятся сложные расчетные системы и системы реального времени.

Вместе с этим каскадная модель имеет ряд недостатков, ограничивающих ее применение при разработке программных продуктов. Рассмотрим основные из них:

- существенная задержка получения результатов;
- ошибки и недоработки на любом из этапов выясняются, как правило, на последующих этапах работ, что приводит к необходимости возврата на предыдущие стадии;
- сложность распараллеливания работ по проекту;
- чрезмерная информационная перенасыщенность каждого из этапов;
- сложность управления проектом;
- высокий уровень риска и ненадежность инвестиций.

Задержка получения результатов обычно считается главным недостатком каскадной схемы. Данный недостаток проявляется в основном в том, что вследствие *последовательного подхода* к разработке согласование результатов с заинтересованными сторонами производится только после завершения очередного этапа работ. Поэтому может оказаться, что разрабатываемое ПО не соответствует требованиям заказчика. Причем такие несоответствия могут возникать на любом этапе разработки – искажения могут непреднамеренно вноситься и проектировщиками-аналитиками, и программистами, так как они не обязательно хорошо разбираются в тех предметных областях, для которых производится разработка программы.

Возврат на более ранние стадии – как недостаток является одним из следствий задержки получения результатов. Ошибки, допущенные на ранних этапах, как правило, обнаруживаются только на последующих стадиях работы. Поэтому после выявления ошибок проект возвращается на предыдущий этап, перерабатывается и снова передается на последующую стадию. Самым неприятным является то, что недоработки предыдущего уровня могут обнаруживаться не сразу на последующем уровне, а позднее (например, на стадии опытной эксплуатации могут проявиться ошибки в описании предметной

области). Это означает, что часть проекта должна быть возвращена на начальный уровень работы. Вообще проект может быть возвращен с любого этапа на любой предыдущий этап, поэтому в реальном случае каскадная схема разработки имеет вид, приведенный на рис. 2.2.

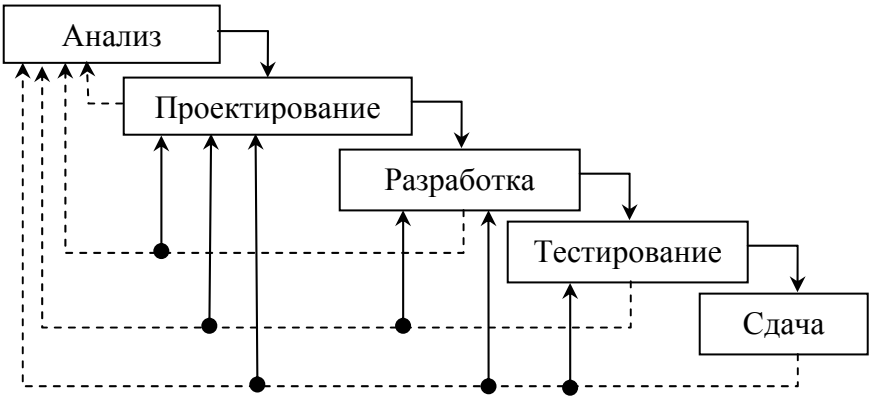


Рис. 2.2.

Одной из причин данной ситуации является то, что в качестве экспертов, участвующих в описании предметной области, часто выступают будущие пользователи системы, которые нередко не могут четко сформулировать то, что они хотели бы получить. Кроме того, заказчики и исполнители часто неправильно понимают друг друга вследствие того, что исполнители обычно не являются специалистами в предметной области решаемой задачи, а заказчики далеки от программирования.

Сложность параллельного ведения работ. Отмеченные выше проблемы возникают как логическое следствие того, что работа над проектом строится в виде цепочки последовательных шагов. Причем даже в случае, когда разработку некоторых частей проекта – *подсистем* – можно вести параллельно, при использовании каскадной схемы распараллеливание работ весьма затруднительно. Сложности связаны с необходимостью постоянного согласования различных частей проекта, и чем

сильнее взаимозависимость отдельных частей, тем чаще и тщательнее должна выполняться синхронизация, т.е. тем сильнее зависимы друг от друга группы разработчиков.

Отсутствие параллелизма затрудняет и организацию работы всего коллектива разработчиков. Работа одних групп сдерживается другими. Например, пока производится анализ предметной области, проектировщики, разработчики и те, кто занимается тестированием и администрированием, почти не имеют работы. Кроме того, при последовательной разработке крайне сложно внести изменения в проект после завершения этапа и передаче проекта на следующую стадию. Так, например, если после передачи проекта на следующий этап группа разработчиков нашла более эффективное решение, оно не может быть использовано по причине того, что более раннее решение уже, возможно, реализовано и связано с другими частями проекта. Поэтому исключается (или, по крайней мере, существенно затрудняется) доработка проекта после его передачи на следующий этап.

Информационная перенасыщенность. Проблема информационной перенасыщенности возникает вследствие сильной зависимости между различными группами разработчиков. Данная проблема заключается в том, что при внесении изменений в одну из частей проекта необходимо оповещать всех разработчиков, которые использовали или могли использовать эту часть в своей работе. Когда система состоит из большого количества взаимосвязанных подсистем, то синхронизация внутренней документации становится важной самостоятельной задачей. Причем синхронизация документации на каждую часть системы – это не более чем процесс оповещения групп разработчиков. Самим же разработчикам необходимо ознакомиться с изменениями и оценить, не сказались ли эти изменения на уже полученных результатах. Все это может потребовать проведения повторного тестирования и даже внесения изменений в уже готовые части проекта. Причем эти измене-

ния, в свою очередь, должны быть отражены во внутренней документации и быть разосланы другим группам разработчиков. Как следствие, объем документации по мере разработки проекта растет очень быстро, так что требуется все больше времени для составления документации и ознакомления с ней.

Следует также отметить, что, кроме изучения нового материала, не отпадает и необходимость в изучении старой информации. Это связано с тем, что вполне вероятна ситуация, когда в процессе выполнения разработки изменяется состав группы разработчиков. Новым разработчикам необходима информация о том, что было сделано до них. Причем чем сложнее проект, тем больше времени требуется, чтобы ввести "новичков" в курс дела.

Сложность управления проектом при использовании каскадной схемы в основном обусловлена строгой последовательностью стадий разработки и наличием сложных взаимосвязей между различными частями проекта. Последовательность разработки проекта приводит к тому, что одни группы разработчиков должны ожидать результатов работы других команд. Поэтому требуется административное вмешательство для того, чтобы согласовать сроки работы и состав передаваемой документации.

В случае же обнаружения ошибок в выполненной работе необходим возврат к предыдущим этапам выполнения проекта, что приводит к дополнительным сложностям. Разработчики, допустившие просчет или ошибку, вынуждены прервать текущую работу (над новым проектом) и заняться исправлением ошибок. Следствием этого обычно является срыв сроков выполнения как исправляемого, так и нового проектов. Требовать от команды разработчиков ожидания окончания следующей стадии разработки нерационально, так как это приведет к существенным потерям рабочего времени.

Упростить взаимодействие между группами разработчиков и уменьшить информационную перенасыщенность доку-

ментации можно, уменьшая количество связей между отдельными частями проекта. Однако далеко не каждую систему можно разделить на несколько слабо связанных подсистем.

Высокий уровень риска. Чем сложнее проект, тем больше продолжительность каждого из этапов разработки и тем сложнее взаимосвязи между отдельными частями проекта, количество которых также увеличивается. Причем результаты разработки можно реально увидеть и оценить лишь на этапе тестирования, то есть после завершения анализа, проектирования и разработки – этапов, выполнение которых требует значительного времени и средств. Как уже было отмечено выше, запоздалая оценка создает значительные проблемы при выявлении ошибок анализа и проектирования.

С другой стороны, возврат на предыдущие стадии может быть связан не только с ошибками, но и с изменениями, произошедшими за время выполнения разработки в предметной области или в требованиях заказчика. Причем возврат проекта вследствие этих причин на доработку не гарантирует, что предметная область снова не изменится к тому моменту, когда будет готова следующая версия проекта. Фактически это означает, что существует вероятность того, что процесс разработки «зациклится» и никогда не дойдет до сдачи в эксплуатацию. Расходы на проект будут постоянно расти, а сроки сдачи готового продукта – постоянно откладываться.

Поэтому можно утверждать, что сложные проекты, разрабатываемые по каскадной схеме, имеют повышенный уровень риска. Этот вывод подтверждается практикой [8]: по сведениям консалтинговой компании The Standish Group, в США более 31 % проектов корпоративных информационных систем (IT-проектов) заканчивается неуспехом; почти 53 % IT-проектов завершается с перерасходом бюджета (в среднем на 189 %, то есть почти в два раза); и только 16,2 % проектов укладывается и в срок, и в бюджет.

2.2.2. Спиральная модель

Спиральная модель, в отличие от каскадной, предполагает *итерационный* процесс разработки системы. При этом возрастает значение начальных этапов жизненного цикла, таких, как анализ и проектирование, на которых проверяется и обосновывается реализуемость технических решений.

Каждая *итерация* представляет собой законченный цикл разработки, приводящий к выпуску внутренней или внешней версии изделия (или подмножества конечного продукта), которое совершенствуется от итерации к итерации, чтобы в конечном итоге стать законченной системой (рис. 2.3).

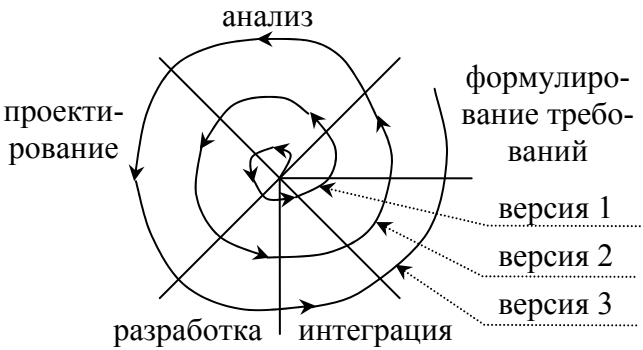


Рис. 2.3.

Таким образом, каждый виток спирали соответствует созданию фрагмента или версии программного изделия, на нем уточняются цели и характеристики проекта, определяется его качество, планируются работы следующего витка спирали. На каждой итерации углубляются и последовательно конкретизируются детали проекта, в результате чего выбирается обоснованный вариант, который доводится до окончательной реализации.

Использование спиральной модели позволяет осуществлять переход на следующий этап выполнения проекта, не дожидаясь полного завершения работы на текущем – недоделан-

ную работу можно будет выполнить на следующей итерации. Главная задача каждой итерации – как можно быстрее создать работоспособный продукт, который можно показать пользователям системы. Таким образом, существенно упрощается процесс внесения уточнений и дополнений в проект.

Спиральный подход к разработке программного обеспечения позволяет преодолеть большинство недостатков каскадной модели и, кроме того, обеспечивает ряд дополнительных возможностей, делая процесс разработки более гибким.

Рассмотрим преимущества итерационного подхода:

- существенно упрощается *внесение изменений* в проект при изменении требований заказчика;
- отдельные элементы разработанного ПО *интегрируются* в единое целое *постепенно* и фактически *непрерывно*. Поскольку интеграция начинается с меньшего количества элементов, то с ней возникает гораздо меньше проблем (по некоторым оценкам [8], при использовании каскадной модели интеграция занимает до 40 % всех затрат в конце проекта);
- *уменьшение уровня рисков* – данное преимущество является следствием предыдущего, так как риски обнаруживаются именно во время интеграции. Вообще уровень рисков максимален в начале разработки проекта, а по мере продвижения разработки ожидаемый риск уменьшается. Данное утверждение справедливо при любой модели, однако для спиральной модели уменьшение уровня рисков происходит с наибольшей скоростью. Это объясняется тем, что при итерационном подходе интеграция выполняется уже на первой итерации, и при выполнении начальных итераций выявляются многие аспекты проекта (пригодность используемых инструментальных средств и программного обеспечения, квалификация разработчиков и т. п.). На рис. 2.4 приведены графики зависимости уровня рисков от времени разработки при использовании каскадного и итерационного подходов;

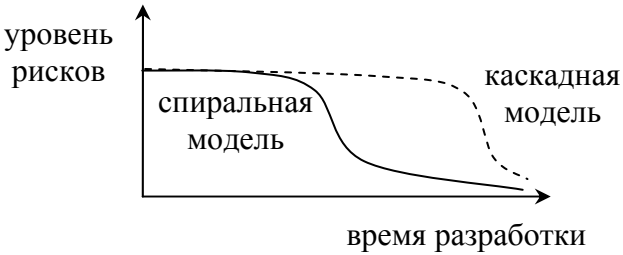


Рис. 2.4.

➢ итерационная разработка обеспечивает *большую гибкость в управлении проектом*, давая возможность внесения тактических изменений в разрабатываемое изделие. Например, можно сократить сроки разработки за счет уменьшения функциональности системы или использовать в качестве составных частей системы продукцию сторонних фирм вместо собственных разработок;

➢ итерационный подход позволяет использовать *компонентный подход* к программированию (упрощает повторное использование ранее разработанных компонентов). Анализ проекта после проведения начальных итераций позволяет выявить общие, многократно используемые компоненты, которые на последующих итерациях будут совершенствоваться;

➢ спиральная модель позволяет получить более *надежную и устойчивую систему*. Это связано с тем, что по мере развития системы ошибки и слабые места обнаруживаются и исправляются на каждой итерации. Одновременно могут корректироваться критические параметры эффективности, что при использовании каскадной модели выполняется только перед внедрением системы;

➢ анализ, проводимый в конце каждой итерации, позволяет оценить необходимые изменения в организации разработки, и улучшить ее на следующей итерации, т.е. итерационный подход позволяет *совершенствовать процесс разработки*.

Основная проблема спирального цикла – *определение момента перехода* на следующий этап. Для ее решения необ-

ходимо ввести временные или функциональные ограничения на каждый из этапов, иначе процесс разработки может превратиться в бесконечное совершенствование уже сделанного. Полезно также следовать принципу "лучшее – враг хорошего". Завершение итерации должно производиться строго в соответствии с планом, даже если не вся запланированная работа закончена. Планирование работ обычно проводится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

2.3. Методология, технология и инструментальные средства разработки прикладного программного обеспечения

Методология, *технология* и *инструментальные средства* (*CASE-средства*) составляют основу проектирования любой – программной, технической, информационной – систем. Применительно к ПО, методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов жизненного цикла программных продуктов.

Методология создания прикладных программ заключается в организации процесса построения ПО и обеспечении управления этим процессом для того, чтобы гарантировать выполнение требований как к самой системе, так и к характеристикам процесса разработки.

Среди основных задач, решение которых должна обеспечивать методология (с помощью соответствующего набора инструментальных средств), необходимо выделить следующие:

- разрабатываемое ПО должно отвечать запланированным целям и задачам;
- желательно обеспечить простоту сопровождения, модификации и расширения системы, которая также должна отвечать требованиям открытости, переносимости и масштабируемости;

- возможность использования в создаваемой системе средств информационных технологий (программного обеспечения, баз данных, средств вычислительной техники, телекоммуникаций).

Основное содержание *технологии* проектирования составляют технологические инструкции, состоящие из описания последовательности технологических операций, условий, в зависимости от которых выполняется та или иная операция, и описании самих операций. Технология проектирования может быть представлена как совокупность трех составляющих:

- заданной последовательности выполнения технологических операций проектирования;
- критериев и правил, используемых для оценки результатов выполнения технологических операций;
- графических и текстовых средств (*нотаций*), используемых для описания проектируемой системы.

Каждая технологическая операция должна быть обеспечена данными, полученными на предыдущей операции (или исходными данными); методическими материалами, инструкциями, нормативами, стандартами; программными и техническими средствами. Результаты выполнения операции должны представляться в некотором определенном *стандартном* виде, обеспечивающем их адекватное восприятие при выполнении последующих технологических операций (где они будут использоваться в качестве исходных данных).

Среди общих требований, которым должна удовлетворять технология проектирования, необходимо отметить возможность разделения крупных проектов на ряд подсистем – *декомпозицию* проекта, которая повышает эффективность работ. Подсистемы, на которые разбивается проект, должны быть, по возможности, слабо связаны по данным и функциям и разрабатываться отдельно. При этом необходимо обеспечить координацию работ и исключить дублирование результатов.

Желательно предусмотреть возможность управления конфигурацией, ведения версий и составляющих проекта, возможность автоматического выпуска проектной документации. Также в ряде случаев требуется обеспечивать независимость выполняемых проектных решений от средств реализации системы – системы управления базами данных, операционной системы, языка и системы программирования.

Инструментальные средства автоматизированной разработки прикладных программ принято называть *CASE-средствами* (*Computer Aided Software/System Engineering*). Отметим, что в настоящее время значение этого термина расширилось и приобрело новый смысл, охватывающий процесс разработки сложных информационных систем в целом. Теперь под термином CASE-средства понимаются программные средства, поддерживающие процессы создания и сопровождения информационных систем, включая анализ и формулировку требований, проектирование прикладного программного обеспечения и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

О методологии RAD как о технологии быстрой разработки приложений говорилось в пп. 1.1.5. Дело в том, что на начальном этапе существования программных средств их разработка велась на традиционных языках программирования. По мере возрастания сложности разрабатываемых систем и требований к функциональности ПО (чему в значительной степени способствовал прогресс в области вычислительной техники, а также появление удобного графического интерфейса) потребовались новые средства, сокращающие сроки разработки. Это послужило предпосылкой к созданию целого направления в области разработки ПО – инструментальных средств для быстрой разработки прикладных программ. Развитие этого направления привело к появлению на рынке программного обес-

печения средств автоматизации практически всех этапов жизненного цикла программных и информационных систем.

RAD – это комплекс специальных инструментальных средств быстрой разработки прикладных программных систем, оперирующих с определенным набором графических объектов, функционально отображающих отдельные информационные компоненты приложений. При использовании этой методологии большое значение имеют опыт и профессионализм разработчиков.

Основные принципы методологии RAD можно свести к следующему [8]:

- используется итерационная модель разработки, причем полное завершение работ на каждом из этапов жизненного цикла не обязательно;
- в процессе разработки необходимо тесное взаимодействие с заказчиком и будущими пользователями;
- необходимо применение CASE-средств и средств быстрой разработки приложений;
- необходимо применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- тестирование и развитие проекта осуществляются одновременно с разработкой;
- разработка ведется немногочисленной и хорошо управляемой командой профессионалов, при этом необходимо грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

Средства RAD дали возможность реализовывать совершенно иную по сравнению с традиционной технологию создания приложений. Информационные объекты формируются как некие действующие модели (*прототипы*), чье функционирование согласовывается с пользователем, а затем разработчик может переходить непосредственно к формированию законченных приложений, не теряя из виду общей картины проек-

тируемой системы. Возможность использования подобного подхода в значительной степени является результатом применения принципов *объектно-ориентированного проектирования*, которые позволяют преодолеть одну из главных трудностей, возникающих при разработке сложных систем – колоссальный разрыв между реальным миром (*предметной областью* описываемой проблемы) и имитирующей средой.

Использование объектно-ориентированных методов позволяет создать описание (модель) предметной области в виде совокупности *объектов* – сущностей, объединяющих данные и методы обработки этих данных (процедуры). Каждый объект обладает своим собственным поведением и моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует определенное поведение. В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования. Объекты обладают целостностью, которая не может быть нарушена. Таким образом, свойства, характеризующие объект и его поведение, остаются неизменными. Объект может только менять состояние, управляться или становиться в определенное отношение к другим объектам.

Широкую известность объектно-ориентированное программирование получило с появлением *визуальных средств проектирования*, когда было обеспечено слияние (*инкапсуляция*) данных с процедурами, описывающими поведение реальных объектов, в объекты программ, которые могут быть отображены определенным образом в графической пользовательской среде. Это позволило приступить к созданию программных систем, максимально похожих на реальные, и добиваться наивысшего уровня абстракции. В свою очередь, объектно-ориентированное программирование позволяет создавать более надежные коды, т.к. у объектов программ существует точно определенный и жестко контролируемый интерфейс. При

разработке приложений с помощью инструментов RAD используется множество готовых объектов, сохраняемых в общедоступном хранилище (*депозитарии*), но обеспечивается и возможность разработки новых объектов, которые могут разрабатываться как на основе существующих, так и "с нуля".

Инструментальные средства RAD обладают удобным графическим интерфейсом и позволяют на основе стандартных объектов формировать простые приложения практически без написания программного кода. Это в значительной степени сокращает рутинную работу по разработке интерфейсной части приложений, т.к. при использовании обычных средств разработка интерфейсов представляет собой достаточно трудоемкую задачу, отнимающую много времени. Таким образом, инструменты RAD позволяют разработчикам сконцентрировать усилия на сущности реальных процессов предметной области объекта программирования, что в конечном итоге приводит к повышению качества разрабатываемой системы.

Визуальные средства разработки ПО оперируют в первую очередь со стандартными интерфейсными объектами и элементами управления – *окнами*, *списками*, *текстами*, *кнопками*, *переключателями*, *флажками*, *меню* и т. п., которые позволяют легко преобразовывать информацию, отображать ее на экране монитора, осуществляется управление отображаемыми данными. Все эти объекты могут быть стандартным образом описаны средствами языка, а сами описания сохранены для дальнейшего повторного использования.

В настоящее время существует довольно много различных визуальных средств разработки приложений, которые могут быть разделены на две группы – *универсальные* и *специализированные*. Последние ориентированы только на выполнение специализированных работ определенного класса, например, на создание приложений баз данных, а с помощью универсальных CASE-средств могут быть разработаны приложения практически любого типа.

Логика приложения, построенного с помощью RAD, является событийно-ориентированной, т.е. управление объектами осуществляется с помощью *событий*. Это означает следующее: каждый объект, входящий в состав приложения, может генерировать события и реагировать на события, генерируемые другими объектами. Примерами событий могут быть: *открытие* и *закрытие окон*, *нажатие кнопки* или *клавиши клавиатуры*, *движение мыши*, *изменение данных* в базе данных и т. п. Разработчик реализует логику приложения путем определения *обработчика* каждого события – процедуры, выполняемой объектом при наступлении соответствующего события. Например, обработчик события "нажатие кнопки" может открыть диалоговое окно.

При использовании методологии быстрой разработки приложений разработка программных продуктов состоит из четырех фаз:

- фаза анализа и планирования требований;
- фаза проектирования;
- фаза построения;
- фаза внедрения.

Несмотря на все свои достоинства, методология RAD (как, впрочем, и любая другая методология) не может претендовать на универсальность. Ее применение наиболее эффективно при разработке сравнительно небольших ПО. При разработке же типовых систем, не являющихся законченным продуктом, а представляющих собой совокупность типовых элементов (например, средств автоматизации проектирования), большое значение имеют такие показатели проекта, как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Это связано с тем, что типовые системы обычно централизованно сопровождаются и могут быть адаптированы к различным программно-аппаратным платформам, системам управления базами данных, коммуникационным средствам, а также интегрироваться

с существующими разработками. Поэтому для такого рода проектов необходим высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Ограничено применение методологии RAD для построения сложных расчетных программ, операционных систем или программ управления сложными инженерно-техническими объектами – программ, требующих написания большого объема уникального кода. Методология RAD мало эффективна для разработки приложений, в которых интерфейс пользователя является вторичным, то есть отсутствует наглядное определение логики работы ПО. Примерами могут служить приложения реального времени, драйверы или утилиты.

Совершенно неприемлема методология RAD для разработки систем, от которых зависит безопасность людей, например, систем управления транспортом или атомных электростанций. Это обусловлено тем, что итеративный подход, являющийся одной из основ RAD, предполагает, что первые версии системы не будут полностью работоспособны, что в данном случае может привести к серьезнейшим катастрофам.

3. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В РАМКАХ ЯЗЫКА OBJECT PASCAL

Прежде чем приступить к изложению основ языка программирования высокого уровня Object Pascal, необходимо отметить следующее.

Во-первых, объем всего материала по данному вопросу явно превышает размеры настоящего учебного пособия. По этой причине авторы настоятельно рекомендуют активно работать с научно-технической литературой. Основы Object Pascal и технология программирования подробно изложены, например, в [2, 4, 8]. Особенно стоит выделить справочные материалы [1], поскольку ими очень часто приходится пользоваться. Не стоит забывать о клавише `F1`, которая в среде Delphi активизирует справочную информацию.

Во-вторых, отдельные разделы по содержанию очень тесно взаимосвязаны друг с другом, что создает определенные трудности при изучении, ибо для понимания текущего материала необходимы знания, изложение которых предполагается далее. Идеальным вариантом изучения, по мнению авторов, является неоднократное прочтение предлагаемого материала.

Завершая вводную часть данного раздела, приведем краткую аннотацию некоторых книг, в которых можно найти специализированную информацию по программированию на языке высокого уровня, в том числе и в среде Delphi. Вопросы методологии и технологии проектирования баз данных (как *реляционных*, так и *объектно-ориентированных*) рассмотрены в [6, 8]. Разработка мультимедийных приложений и работа с графическими компонентами подробно разбираются, например, в [9]. Очень интересные, нестандартные и профессиональные приемы программирования изложены в [5]. Методология и вопросы разработки справочной системы для проектируемых программных средств приведены в [2, 3, 8].

3.1. Элементарная грамматика языка Object Pascal

Программный код в языке Object Pascal записывается с помощью символов латинского алфавита, цифр и некоторых других символов. Строчные и ПРОПИСНЫЕ буквы различаются только в строковых выражениях, но не в символических именах *классов*, *переменных*, *функций* и т.д. Для *символических имен* в дальнейшем будет использоваться понятие *допустимый идентификатор* – произвольное символьное имя, состоящее из букв латинского алфавита, цифр, символа подчеркивания «_», причем первым символом не может быть цифра. Примеры:

```
A, D3, _23, Andr_Romanov;    //допустимые имена
2W, 34, @EE, КОЛЯ;           //недопустимые имена
```

Комментарии отделяются от текста программы тремя способами:

1) символами "`//`", как в приведенном примере (таким образом обычно определяется *короткий* комментарий, находящийся на одной строке с программным кодом);

2) заключаются в фигурные скобки – `{ комментарий }`; заметим, что при изложении теории программирования в фигурные скобки могут заключаться необязательные фрагменты программного кода;

3) большой текстовой фрагмент комментария принято помещать между сочетаниями символов "`(*`" и "`*)`".

Также при изложении теоретических положений программирования принято использовать символы "<" и ">". В них заключаются фрагменты программного кода, как правило, произвольно определяемые пользователем.

Текст, являющийся комментарием, игнорируется компилятором. Но существует единственное исключение: если сразу после фигурной скобки стоит символ "`$`", то в этом случае мы имеем дело с *директивой компилятора*. Например, `{$I+}` – директива компилятора, а `{$ I+}` – просто комментарий. *Ди-*

рективы компилятора – это указания компилятору на выполнение тех или иных операций. Записываются в формате `{$<директива>}`. По области действия различаются *глобальные*, действующие на весь компилируемый файл, и *локальные*, действующие от места их объявления до места появления другой директивы, отменяющей предыдущую.

В главе 1 (пп. 1.1.3) было сказано, что основными элементами любого языка программирования высокого уровня являются список *служебных* (*зарезервированных*) слов и определенные правила записи *операторов* и *операндов*.

Зарезервированные слова являются *недопустимыми* идентификаторами, т.е. не могут быть использованы в качестве имен переменных, процедур, функций. Полный список их можно найти в специальной литературе, например, в [4]. При случайном использовании их в качестве идентификатора компилятор программы укажет на ошибку. Со многими основными служебными словами Object Pascal будем знакомиться по мере изучения материала. Отметим, что визуальный редактор Delphi выделяет многие зарезервированные слова. Например, если не менять установки редактора, определяемые по умолчанию, то служебные слова будут выделяться **черным жирным** шрифтом (как и в излагаемом далее учебном материале).

Каждая логически завершенная строка (т.е. она не обязательно должна располагаться физически на одной строке, возможен и перенос строк) программного кода завершается символом "`;`".

Если требуется логически выделить несколько операторов в единую структуру, то используются специальные *операторные скобки* – **`begin`** `<группа операторов>` **`end`**.

3.2. Основные структурные единицы

3.2.1. Структуры главного файла программы и модулей

В Delphi *головной* (главный) файл программы обычно не содержит ничего, кроме описания используемых модулей и

операторов инициализации приложения, создания форм и запуска приложения. Структура головного файла имеет следующий вид:

```
Program <имя>; // заголовок программы

Uses ... // объявления подключаемых модулей

{объявления локальных для головного файла типов, классов, констант, переменных, описания локальных функций и переменных}

begin
... {операторы тела программы}
end.
```

Программный код обычно помещают в файлы модулей, каждый из которых в общем случае имеет структуру:

```
unit <имя модуля>;

interface // Открытый интерфейс модуля
(* Сюда могут помещаться списки подключаемых модулей, объявления типов, констант, переменных, функций и процедур, к которым будет доступ из других модулей. *)

implementation // Реализация модуля
(* Сюда могут помещаться списки подключаемых модулей, объявления типов, констант, переменных, недоступных для других модулей. Здесь же должна быть реализация всех объявленных в разделе interface функций и процедур, а также может быть код любых дополнительных, не объявленных ранее процедур и функций. *)

initialization {реализация может отсутствовать}
(* Операторы, выполняемые один раз при первом обращении к модулю *)

finalization { реализация может отсутствовать }
(* Операторы, выполняемые один раз при завершении работы модуля *)

end.
```

3.2.2. Общая характеристика объявляемых элементов

Раздел **interface** представляет собой внешний интерфейс модуля. Подключаемые *модули*, объявленные *типы*, *классы*, *константы*, *переменные*, *функции* и *процедуры* доступны внешним модулям, обращающимся к данному модулю.

Раздел **implementation** представляет собой программную реализацию модуля. Все объявленные в нем элементы доступны только в пределах данного модуля. Основное тело модуля составляет программный код, реализующий объявленные в разделе **interface** функции и процедуры.

Раздел **initialization** включает в себя операторы, которые выполняются только один раз при первом обращении программы к данному модулю. Этот раздел не является обязательным. В нем могут помещаться какие-либо операторы, производящие начальную настройку модуля.

Раздел **finalization** включает в себя операторы, выполняемые только один раз при любом завершении работы программы: *нормальном* или *аварийном*. Это необязательный раздел. В нем могут помещаться операторы, производящие зачистку "мусора" – удаление временных файлов, освобождение ресурсов памяти и т.п. Введение раздела **finalization** не разрешается, если в модуле нет раздела **initialization**.

И в главном файле программы, и в разделах **interface** и **implementation** возможно объявление различных элементов программирования – *модулей*, *классов*, *меток*, *констант*, *пользовательских типов*, *переменных*, *процедур* и *функций*.

Используемые модули идентифицируются ключевым словом **uses**, после которого следует список имен модулей, разделяемый запятыми. Например:

```
uses Windows, SysUtils, Classes, Unit2, MyUnit;
```

Объявление подключаемого модуля в головном файле программы может также содержать после имени модуля ключевое слово **in**, после которого указывается имя файла модуля.

```
uses Unit1 in 'Unit1.pas',
     Unit2 in 'c:\exampes\Unit2.pas';
```

Служебное слово **in** используется в случаях, если Delphi неясно, где надо искать соответствующие файлы. В объявлениях **uses**, включаемых в модули, использование **in** не допускается.

При взаимных ссылках модулей друг на друга не разрешаются *циклические* ссылки с помощью **uses**, размещенных в разделах **interface**. Такие циклы надо разрывать, перенося в одном из модулей, требующем взаимной ссылки, **uses** из разделов **interface** в разделы **implementation**.

Главный файл ПО может содержать только один раздел **uses**, файлы модулей могут одновременно содержать **uses** в разделах **interface** и **implementation**. Разделов объявления *меток*, *типов*, *констант* и *переменных* может быть несколько, и они могут следовать в любом порядке.

Начало *раздела объявления меток* указывается с помощью директивы **label**, после которого через запятую следует список меток. Каждая метка обозначается допустимым идентификатором, но может состоять и из цифр. Например:

```
label Lbegin, 1, second, 9999;
```

Отметим, что использование меток при программировании на современных объектно-ориентированных языках высокого уровня не считается хорошим стилем программирования.

На начало *раздела объявления типов* указывает служебное слово **type**. Вообще в Object Pascal существует большое число стандартных (встроенных) типов. Раздел **type** необходим для создания собственных, нестандартных, *определяемых пользователем типов данных*. После ключевого слова **type** должен следовать ряд объявлений типов в форме:

```
<идентификатор типа> = <описание типа>;
```

Например, объявим перечисляемый тип TMode и тип, определяющий расширение файла как трехсимвольную строку.

```
type TMode : (mRead, mEdit, mWrite);
     TFileExp : String[3];
```

Раздел констант начинается со служебного слова **const**. При компиляции программы вместо символьного имени константы фактически подставляется ее численное значение. Кроме обычных констант, в Object Pascal можно использовать и *константы-переменные*, которые, по сути, являются обычными переменными с начальной инициализацией. Примеры:

```
Const
  Const_1 = 120;         //константа целого типа
  Const_2 = 100.0;       //константа вещественного типа
  Const_3 = '100.0';     //строковая константа
  _A3 = Const_3/sqr(Const_1)+12.3; {константа,
                                 определяемая выражением}
  v_1 : real = 3.14158; {вещественная
                                    константа-переменная}
  v1 : integer = 156;    //целая константа-переменная
```

Начало *раздела переменных* определяется служебным словом **var**. В этих разделах в модулях должны быть описаны все используемые переменные. Компилятор Object Pascal не допускает использования переменных, не объявленных в разделе **var**. Допускается объявление неиспользуемых переменных. Формат объявления переменных в общем виде:

```
<идентификатор переменной> : <тип> {= <значение>};
```

Однотипные переменные можно перечислять через запятую и обозначать любым допустимым идентификатором. Для переменных нестандартного типа имя типа должно быть описано в разделе **type**. Например:

```
type TFileExp : String[3];

var  var33, Myvar, Alfa : real;
```

```
var40 : integer = 55;
txtExp : TFileExp; // переменная, тип которой
                   // объявлен пользователем
```

Как было сказано выше, все объявленные переменные в Delphi действительны только в пределах своей *области действия*. Переменные, объявленные в интерфейсной части модуля, являются *глобальными*, т.е. видимыми во всех других модулях, использующих данный модуль. Глобальные переменные занимают пространство в памяти и не освобождают его, пока программа работает. Переменные, объявленные в разделе **`implementation`** или внутри процедуры или функции, являются *локальными*. С точки зрения программных ресурсов, их использование предпочтительнее, поскольку в этом случае экономится память и несколько облегчается процесс отладки.

3.3. Типы данных и операции над ними

Типы данных в Object Pascal можно разделить на *предопределенные* в языке (*встроенные*) типы и типы, *определяемые пользователем*. На рис. 3.1 приведена классификация предопределенных типов Object Pascal, учитывающая некоторые общие свойства различных типов.

Четыре математических оператора (`+`, `-`, `/`, `*`) и операторы сравнения (`=`, `>`, `>=` и т.д.) работают с любыми числовыми типами. Если хотя бы один из операндов в математических вычислениях является *действительным* типом (т.е. числом с плавающей точкой), то результат также будет действительным числом. Для целочисленного деления используются операторы `DIV` (возвращает целочисленное частное) и `MOD` (возвращает остаток от деления).

3.3.1. Порядковые типы

Порядковыми (ordinal) типами называются те, в которых значения *упорядочены* и для каждого из них можно указать

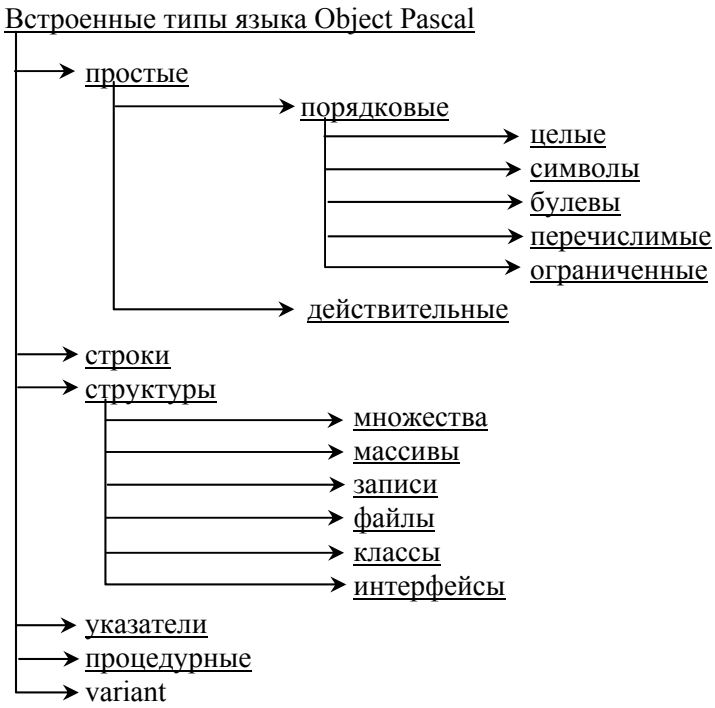


Рис. 3.1. Классификация типов языка Object Pascal

предшествующее и *последующее* значения. Для порядковых типов предопределен ряд функций:

- `Ord(<Выражение порядкового типа>)` – возвращает порядковый номер значения аргумента;
- `Pred(<Выражение порядкового типа>)` – возвращает предшествующее значение аргумента;
- `Succ(<Выражение порядкового типа>)` – возвращает следующую величину значения аргумента;
- `High(<Идентификатор или переменная порядкового типа>)` – максимально возможное значение аргумента;
- `Low(<Идентификатор или переменная порядкового типа>)` – минимально возможное значение аргумента.

Для порядковых типов определены также процедуры инкремента `Inc(i)` и декремента `Dec(i)`, которые соответственно увеличивают или уменьшают на единицу порядковый номер своего аргумента `i`.

Целые типы данных относятся к *целым порядковым* типам и используются для представления целых чисел. В табл. 3.1 приведены диапазоны их изменений. *Родовыми* типами (т.е. обеспечивающими максимальную производительность вычислений) среди перечисленных являются `Integer` и `Cardinal`. Приведенные затраты памяти могут изменяться от версии к версии, поэтому для достоверной оценки рекомендуется пользоваться функцией `SizeOf(X)`, которая возвращает число байт, занимаемых любой переменной или типом `X`.

Таблица 3.1.

Тип	Диапазон значений	память в байтах	Знаковый тип
`Byte`	0 .. 255	1	нет
`Word`	0 .. 65535	2	нет
`ShortInt`	-128 .. 127	1	да
`SmallInt`	-32768 .. 32767	2	да
`Cardinal` или `Longword`	0 .. 4294967295	4	нет
`Integer` или `LongInt`	-2147483648 .. 2147483647	4	да

Символьные типы данных предназначены для хранения одного символа. Родовым является однобайтный тип `Char`, эквивалентный в настоящее время типу `ANSIChar`, поддерживающий множество символов *ANSI* (*American Standard Code for Information*). Именно тип `Char` имеет смысл использовать во всех случаях, кроме обращений к функциям, требующим другой тип символьных данных – `WideChar`, поддерживающий универсальную двухбайтовую кодировку Unicode. Первые 256 символов в этих множествах одинаковы и соответствуют сим-

волам *ASCII* (*American Standard Code for Information Interchange*) от 0 до 255. Для символьного типа предопределена функция `Chr`, возвращающая символ любого целого значения в пределах `AnsiChar` или `WideChar`. Например, `Chr(65)` возвращает символ "`A`".

Переменные *логических* (*булевых*) типов представляют два значения – `true` (истина) и `false` (ложь). Для совместимости Delphi с другими системами определены несколько логических типов (табл. 3.2). Предпочтительнее всегда использовать тип `Boolean`, кроме обращений к процедурам, явным образом требующим другой тип.

Таблица 3.2.

Тип	память в байтах
`Boolean`	1
`ByteBool`	1
`WordBool`	2
`LongBool`	4

Для булевых типов определены булевы операции: `AND` (И), `OR` (ИЛИ), `NOT` (НЕ, отрицание) и `XOR` (исключающее ИЛИ). На рис. 3.2 представлены таблицы истинности для операторов `AND`, `OR` и `XOR`.

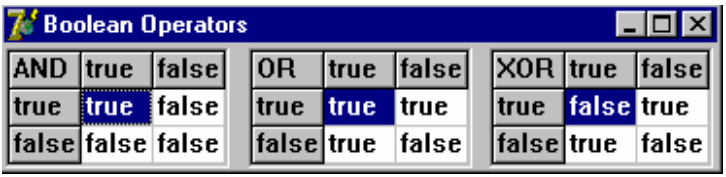


Рис. 3.2. Таблицы истинности для операторов `AND`, `OR` и `XOR`

Ячейки в каждой таблице содержат результат объединения логических значений заголовков по строке и столбцу при использовании заданного логического оператора. Использование этих операций расширяет возможности по формированию сложных условий в ряде операторов.

Предопределенные константы `true` и `false` обладают в разных булевых типах несколько разными свойствами, которые приведены в табл. 3.3.

Таблица 3.3.

тип Boolean	типы ByteBool, WordBool, LongBool
false < true	false <> true
Ord(false) = 0	Ord(false) = 0
Ord(true) = 1	Ord(true) <> 0
Succ(false) = true	Succ(false) = true
Pred(true) = false	Pred(false) = true

Перечислимые типы определяют упорядоченное множество идентификаторов, представляющих собой возможные значения переменных этого типа. Вводятся эти типы для улучшения логичности восприятия программного кода. Многие типы Delphi являются перечислимыми. Это упрощает работу с ними, поскольку дает возможность работать не с абстрактными числами, а с осмысленными значениями.

Пусть, например, в программе должна быть переменная Mode, в которой зафиксирован один из возможных режимов работы приложения: чтение данных, их редактирование, запись данных. Можно, конечно, этой переменной присваивать в нужные моменты времени одно из трех условных чисел: 0 – режим чтения, 1 – режим редактирования, 2 – режим записи. Тогда программа будет содержать операторы вида

```
if (Mode = 1) then ...
```

Можно поступить иначе: определить перечисляемый тип TMode, как это сделано в одном из примеров предыдущего раздела, а переменную Mode объявить как переменную этого типа:

```
type TMode : (mRead, mEdit, mWrite);
var   Mode : TMode;
```

Приведенный оператор изменится следующим образом.

```
if (Mode = mEdit) then ...
```

Конечно, такой оператор понятнее, чем предыдущий.

Переменная перечислимого типа определяется предложениями вида:

```
type <имя> = (<значение 1>, .., < значение n>);
var  <имя> : (<значение 1>, .., < значение n>);
```

Второй вариант делает невозможным объявление и использование другой переменной с теми же значениями.

Переменной перечисляемого типа можно присваивать указанные значения, проверять ее величину, сравнивая с возможными значениями. Кроме того, применимы любые операции сравнения (>, < и т.п.), а также процедуры и функции, определенные для порядковых типов.

Ограниченный тип или *тип-диапазон* для порядковых типов задает *поддиапазон* возможных значений для вводимого типа или переменной. Задается выражением вида `<минимальное значение>..<максимальное значение>`.

Приведем пример: пусть переменная `Letter` может принимать только символы латинских букв в нижнем регистре, переменная `Num` – только целые числа в диапазоне 1..12 (например, номера месяцев) и объявим тип `Caps` как совокупность символов латинских букв в верхнем регистре.

```
var          Letter :  'a'..'z';
             Num : 1..12;
type  Caps = 'A'..'Z';
```

Введение ограниченных типов является неплохим средством отладки. Они часто используются при объявлениях размеров массивов, но могут использоваться и самостоятельно. В компиляторе Object Pascal имеется опция (директива компилятора `{$R+}`), позволяющая включить проверку диапазона при присваивании значения переменной ограниченного типа. При попытке присвоить переменной ограниченного типа значение, выходящее за пределы заданного поддиапазона, генерирует сообщение "`Range check error`".

3.3.2. Действительные типы

Действительные типы данных предназначены для хранения чисел, имеющих дробную часть (табл. 3.4).

Таблица 3.4

Тип	Диапазон значений	Число значащих разрядов	память в байтах
`Real48`	$2.9 \cdot 10^{-39} .. 1.7 \cdot 10^{38}$	11–12	6
`Real` или `Double`	$5.0 \cdot 10^{-324} .. 1.7 \cdot 10^{308}$	15–16	8
`Single`	$1.5 \cdot 10^{-45} .. 3.4 \cdot 10^{38}$	7–8	4
`Extended`	$3.6 \cdot 10^{-4951} .. 1.1 \cdot 10^{4932}$	19–20	10
`Currency`	–922337203685477.5808 .. 922337203685477.5807	19–20	8

Родовым является тип `Real`, который в настоящий момент эквивалентен типу `Double`. Наименьшую производительность обеспечивает тип `Real48`, сохраняемый только для обратной совместимости с более ранними версиями языка. Тип `Extended` обладает максимальной точностью, но могут возникать проблемы с его переносимостью на другие платформы. Для тип `Currency` минимизирована ошибка округления и поэтому он используется для представления денежных величин.

3.3.3. Строки

Строки представляют собой последовательность символов и определяются совокупностью четырех типов – `ShortString`, `AnsiString`, **`String`**, `WideString`. Различные типы строк можно классифицировать по двум признакам:

- *длинные* или *короткие* строки;
- строки, *использующие* или *не использующие* нулевой символ `#0` в конце строки.

Родовым является тип **`String`**, который имеет разный смысл в зависимости от директивы компилятора `$H`. Если

включена директива {$H+} (включена по умолчанию), то **String** интерпретируется компилятором как тип AnsiString – длинная строка (до 2 Гбайт) с нулевым символом ("#0") в конце. Подобное представление строкового типа данных имеет дополнительное имя – ASCIIZ, используются также термины *ограниченная нулем строка* и *строка с завершающим нулем*. При директиве {$H-} **String** интерпретируется как тип ShortString – короткая строка (максимум 255 символов) без завершающего нулевого символа. Тип WideString является длинной строкой с завершающим нулем, используется в серверах COM и интерфейсах, может содержать до 1 Гбайта символов Unicode.

Если в объявлении типа после ключевого слова **String** следует число символов в квадратных скобках (например, **String**[4]), то, независимо от директив компилятора, тип трактуется как строка без нулевого символа в конце с указанным числом символов.

Стандартная функция Length возвращает число символов в строке, переданной ей в качестве параметра. Процедура SetLength устанавливает длину строки. Вообще, для строк определено достаточно большое количество стандартных операторов, процедур и функций.

К строкам применимы операции сравнения. Если сравниваются строки разной длины, то большей считается более длинная строка. При сравнении строк одинаковой длины принимаются в расчет символы в одинаковых позициях – сравниваются коды этих символов.

Сцепление (*конкатенация*) строк осуществляется операцией "+". Например, выражение S1+S2 даст в результате строку, в которой после символов строки S1 будут расположены символы строки S2. Строки разных типов могут присваиваться друг другу и смешиваться в одном выражении. Компилятор при этом осуществляет автоматическое приведение типов. Но строки, передаваемые как параметры в функции и процедуры,

должны иметь указанный в объявлениях тип. В противном случае необходимо использовать операции явного приведения типов или другие приемы программирования.

Со строками можно оперировать как с индексированными массивами символов. Например, `S[i]` – это символ, расположенный в строке в позиции `i` (индексы отсчитываются от `1`, т.е. индекс `1` соответствует первому символу). Для строк типа `ShortString` или `AnsiString` выражение `S[i]` имеет тип `AnsiChar`, для строк типа `WideString` – `WideChar`.

При работе со строками с завершающим нулем часто используют специальные типы указателей: `PChar` и `PWideChar`, которые являются указателями соответственно на массивы с элементами типов `Char` и `WideChar` с нулевым символом в конце. Переменные этих типов часто требуется передавать в различные функции и процедуры в качестве параметров. К тому же типы `PChar` и `PWideChar` существенно упрощают коды по причине совместимости со строковыми константами. Указатели этих типов могут индексироваться точно так же, как строки и массивы символов. Например, если `P` определен как `PChar`, то `P[0]` – это первый символ строки.

В выражениях, операторах присваивания и при передаче параметров в функции и процедуры можно смешивать длинные строки (`AnsiString`) и строки с нулевым символом в конце типа `PChar`. Но иногда требуется явное приведение типа `PChar` к типу длинной строки. Например, операция конкатенации строк требует, чтобы хотя бы один ее операнд имел строковый тип. Если же требуется склеить два объекта типа `PChar`, то это надо сделать с помощью приведения типа:

```
S := string(P1) + string(P2);
```

3.3.4. Массивы

Массив представляет собой структуру данных, позволяющую хранить под одним именем совокупность данных одного любого конкретного типа. Массив характеризуется *име-*

нем, *типом* хранимых элементов, *размером* (числом элементов), *нумерацией* элементов и *размерностью*. Различают массивы *статические*, размеры которых устанавливаются в момент их объявления, и *динамические*, размеры которых могут изменяться во время выполнения. В функции или процедуры можно передавать в качестве параметров как статические, так и *открытые* массивы, размер которых неизвестен. Кроме того, можно передавать *открытые массивы констант*, содержащие значения различных типов. Для массивов определен ряд операций и функций.

Объявление статического массива имеет вид:

```
var <имя>: array [<ограниченный тип>,...] of <тип элементов>;
```

Например:

```
var A: array [1..10] of integer; //одномерный массив
  A2: array[1..10,1..3] of integer; //двумерный массив

{массив с индексом перечислимого типа:}
Type color=(red,yellow,green);
var ACol: array [color] of integer;

    Ch: array ['a'..'z'] of char; //массив символов
```

Доступ к элементам массива осуществляется по индексам. Например, `A[i]` или `A2[4,3]`.

Объявление переменной или типизированной константы типа массива можно совмещать с заданием начальных значений, которые перечисляются после знака равенства, разделяются запятыми и заключаются в круглые скобки. Например:

```
Var A1 : array[0..10] of integer =
                              (1,2,3,4,5,6,7,8,9,10,11);
Const A1 : array[0..10] of integer =
                              (1,2,3,4,5,6,7,8,9,10,11);
```

Массивы символов эквивалентны строкам, и с ними можно во многом обращаться как со строкой. Для массивов

символов можно использовать задание начальных значений и по элементам, и сразу всей строке. Например, эквивалентны друг другу следующие объявления

```
Const S:array[0..3] of char = ('A','B','C','D');
Const S:array[0..3] of char = 'ABCD';
```

При задании начальных условий для *многомерных массивов* список значений по каждой размерности заключается в скобки, например:

```
type Ar3 = array[1..4,1..3,1..2] of integer;
const A3: Ar3=(((0,1),(2,3),(4,5)),
               ((6,7),(8,9),(10,11)),
               ((12,13),(14,15),(16,17)),
               ((18,19),(20,21),(22,23)));
```

Отметим, что в конструкциях, подобным объявлению константы `A3`, легко запутаться в открывающих (закрывающих) скобках. В Delphi существует специальное сочетание клавиш `Alt+[` (`Alt+]`), позволяющее осуществить поиск соответствующей открывающей (закрывающей) скобки.

Многомерный динамический массив определяется как динамический массив динамических массивов и т.д. Например, определим двумерный динамический массив:

```
var A2: array of array of integer;
```

Для выделения памяти под динамический массив необходимо использовать процедуру `SetLength`, передавая ей в качестве параметров несколько размеров. Например, для задания динамического массива размерностью 3 на 4 для объявленной переменной `A2` необходим оператор

```
SetLength(A2,3,4);
```

Отметим, что можно создавать и непрямоугольные массивы, в которых, например, второй размер не постоянен и варьируется в зависимости от номера строки [1].

При создании функции или процедуры работы с массивами в ее объявление нельзя включать описание индексов. Например, объявление

```
procedure MyProc(A: array[1..10] of Integer);
```

будет расценено как синтаксическая ошибка и вызовет соответствующее сообщение компилятора. Правильное объявление:

```
type ta = array[1..10] of Integer;
procedure MyProc(A: ta);
```

Функции и процедуры в Object Pascal могут воспринимать в качестве параметров не только массивы фиксированного размера, но и так называемые *открытые* массивы, размер которых неизвестен. В этом случае в объявлении функции или процедуры они описываются как массивы базовых типов без указания их размерности. Например:

```
procedure SumArray(A:array of integer;
                   var B: array of integer);
```

При таком определении передаваемый в функцию первый массив будет копироваться и с этой копией – массивом `A` будет работать процедура. Второй открытый массив определен как **`var`**. Этот массив передается по ссылке, т.е. он не копируется, и процедура будет работать непосредственно с исходным массивом. Открытый массив воспринимается в теле процедуры или функции как массив с целыми индексами, начинающимися с нуля. Размер массива может быть определен функциями `Length` – число элементов и `High` – наибольшее значение индекса. Очевидно, что всегда `High = Length - 1`.

При вызове функции или процедуры с параметром в виде открытого массива можно использовать в качестве аргумента *конструктор открытого массива*, который формирует массив непосредственно в операторе вызова. Список элементов такого конструктора массива заключается в квадратные скобки, а

значения элементов разделяются запятыми. Например, функцию `Sum`, объявленную в специальном математическом модуле `Math` как

```
function Sum(const Data: array of Double): Extended;
```

которая суммирует элементы числового массива, можно вызвать следующим образом:

```
Sum([1.2,4.45,0.1]);
```

3.3.5. Множества

Множество – это группа элементов, которая ассоциируется с идентификатором (именем) и с которой можно сравнивать другие величины, чтобы определить: принадлежат ли они этому множеству. Один и тот же элемент не может входить в множество более одного раза. Как частный случай, множество может быть пустым. Множество определяется перечислением элементов, заключенных в прямоугольные скобки. Такая форма определения называется *конструктором множества*.

Например, если множество возможных единичных символов, которые могут быть получены в ответ на вопрос программы "Yes/No", содержит символы `y`, `Y`, `n` и `N`, то это множество можно описать конструктором `['y','Y','n','N']`. Для определения, принадлежит ли переменная множеству, служит операция `in`. Например, проверить, дал ли пользователь один из допустимых ответов, можно оператором:

```
if (key in ['y','Y','n','N'])
  then <оператор, выполняемый при допустимом ответе>
```

Множества могут содержать не только отдельные значения, но и ограниченные типы. Например, если необходимо контролировать символы, вводимые пользователем при вводе целого положительного или отрицательного числа, можно определить множество `['0'..'9', '+', '-']` и использовать

его, например, при обработке события OnKeyPress любого окна редактирования с помощью следующего кода:

```
if not(Key in ['0'..'9', '+', '-']) then Key:=#0;
```

Подобный оператор не позволяет пользователю ввести символы, отличные от имеющихся в множестве.

В приведенных операторах множество использовалось непосредственно, заранее не объявляясь в виде типа. Но если, например, в приложении в ряде мест надо проводить проверки, аналогичные приведенным выше, то целесообразнее объявить переменную или типизированную константу типа множества или тип множества и несколько переменных этого типа. Объявление типа множества делается в форме **set of** <базовый тип>. Приведем примеры.

```
var K: set of Char=['0'..'9', '+', '-']; // Объявление
                  глобальной переменной с инициализацией

// Объявление типизированной константы
const K1: set of Char=['0'..'9', '+', '-'];

// Объявление типа множества
type TDigit = set of '0'..'9';
var D1, D2: TDigit;          // и переменных этого типа
```

Далее в программе допустимы следующие операторы.

```
...
if ((key in K)OR(key in K1)) then ...
...
D1:=['0','1'];
D2:=['2'..'9'];
```

Помимо операции **in** для множеств определен еще ряд операций: *объединение*, *пересечение*, *операции отношения* и ряд других (табл. 3.5).

Таблица 3.5.

Обозначение	Операция	Типы операндов	Тип результата	Пример
+	объединение	Set	Set	Set1+Set2
-	разность	Set	Set	S-T
*	пересечение	Set	Set	S*T
<=	подмножество	Set	Boolean	Q<=MySet
>=	включающее множество	Set	Boolean	S1>=S2
=	эквивалентность	Set	Boolean	S2=MySet
<>	неэквивалентность	Set	Boolean	MySet<>S1
in	является элементом	порядковый, Set	Boolean	A in Set1

В указанных операциях действуют следующие правила.

1. Z является элементом X+Y, если он является элементом X или Y, или и X, и Y.

2. Z является элементом X-Y, если он является элементом X, но не является элементом Y.

3. Z является элементом X*Y, если он является элементом и X, и Y.

4. Выражение X<=Y возвращает true, если каждый элемент X является элементом Y.

5. Выражение X>=Y эквивалентно выражению Y<=X.

6. Выражение X=Y возвращает true, если X и Y содержат точно одни и те же элементы. В противном случае true будет возвращено выражением X<>Y.

3.3.6. Записи

Запись (record), также часто называемая *структурой*, представляет собой *набор данных различных типов, объеди-*

ненный общим именем. Отдельные данные записи называются *полями*. Тип записи объявляется следующим образом:

```
type <имя типа> = record
  <список имен полей> : <тип>;
  ...
  <список имен полей> : <тип>;
 end;
```

Например, определим запись и переменные этого типа:

```
type
 TPers= record
  Fam,Name : String[15]; // фамилия, имя
  Year     : Integer;    // год рождения
 end;

var Pers, Pers1 : TPers;
```

Можно объявлять запись и непосредственно в объявлении переменной, не прибегая специально к объявлению типа.

Доступ к отдельным полям записи осуществляется указанием на соответствующую переменную типа записи и после символа точки "." – имени поля. Например, `Pers.Fam:= 'Иванов'`. При групповых операциях с полями удобно использовать оператор `With..do`, позволяющий не указывать перед каждым полем имя переменной типа записи.

Могут быть определены более сложные записи *с вариантной частью*. Основная цель указания вариантной части – экономия памяти. Синтаксис такого объявления следующий:

```
type <имя типа записи> = record
       <список имен полей> : <тип>;
       ...
       <список имен полей> : <тип>;
       case <тег>: <порядковый тип> of
         <список констант>: (<вариант 1>);
          ...
         <список констант>: (<вариант 2>);
     end;
```

Первая часть этого объявления до ключевого слова **case** не отличается от того, что было описано ранее. За словом **case** следует не обязательный элемент *тег* – произвольный допустимый идентификатор. Он вводится только для удобства и может быть пропущен вместе с последующим двоеточием. Затем указывается любой порядковый тип. После слова **of** следуют строки со *списком констант* указанного порядкового типа, завершающиеся двоеточиями. После двоеточия в круглых скобках с необязательной завершающей точкой с запятой записываются поля соответствующего варианта. Эти поля представляются в форме, аналогичной основным полям структуры. В полях вариантов нельзя использовать типы длинных строк, динамических массивов, `variant`, а также структуры, содержащие поля этих типов. Тег и списки констант вводятся просто для удобства программирования. Никакой смысловой нагрузки они не несут, а просто помогают разобраться, к чему относится каждый вариант. Константы в списках не должны повторяться. Поля различных вариантов размещаются в одной и той же области памяти, перекрывая друг друга. Размер этой перекрываемой области компилятор выбирает по размеру, необходимому для наиболее объемного варианта.

В записи могут быть поля, содержащие указатели на другие аналогичные записи. Такие записи называются *самоадресуемыми* и используются для организации в динамически распределяемой области памяти списков структур. Приведем пример подобной самоадресуемой структуры.

```
Type trec=^rec;
     rec=record
      in1,in2:word;
      s:string[10];
      pr:trec  // указатель на структуру типа trec
     end;
```

Подобное описание является исключением из общего правила языка Pascal, по которому нельзя использовать в

предложениях еще не объявленные типы и переменные. В данном случае первое предложение объявляет тип `trec` как указатель на структуру типа, объявленного после этого. Сразу за этим предложением следует объявление самой структуры. В этом объявлении имеется поле `pr`, представляющее собой указатель на структуру аналогичного вида.

3.3.7. Файлы

Различают *файлы* трех видов: *текстовые* файлы, *типизированные* файлы и *нетипизированные* файлы.

Текстовые файлы состоят из последовательностей символов, разбитых на строки. В Object Pascal предопределен тип `TextFile`, соответствующий текстовому файлу. Таким образом, объявление файловой переменной может иметь вид:

```
var <имя файловой переменной>: TextFile;
```

Работа с текстовыми файлами осуществляется процедурами и функциями *файлового ввода/вывода*. Основные процедуры чтения – `Read`, `ReadLn`, `Write` и `WriteLn`.

Типизированные файлы являются *двоичными* файлами, содержащими последовательность однотипных данных. Объявление файловых переменных таких файлов имеет вид:

```
var <имя файловой переменной>: file of <тип данных>;
var F: file of real; // пример
```

Процедуры чтения и записи `Read` и `Write` для типизированных файлов не отличаются от соответствующих процедур для текстовых файлов. Не определены процедуры `Readln` и `Writeln`, но есть процедура `Seek`, позволяющая перемещаться по файлу не только последовательно, как в текстовых файлах, а сразу переходить к требуемому элементу. Имеется также функция `FilePos`, возвращающая текущую позицию в файле.

Нетипизированные файлы – это двоичные файлы, которые могут содержать самые различные данные в виде последо-

вательности байтов. Программист при чтении этих данных сам должен разбираться, какие байты к чему относятся. Тип файловой переменной нетипизированного файла объявляется следующим образом:

```
var <имя файловой переменной>: file;
```

Открытие нетипизированных файлов осуществляется процедурами `Reset` и `Rewrite`, синтаксис которых несколько отличен от аналогичных процедур для других видов файлов тем, что в этих процедурах указывается размер одной записи в байтах. Для записи и чтения данных в нетипизированных файлах имеются процедуры `BlockRead` и `BlockWrite`, которые читают или записывают указанное в них число записей.

3.3.8. Указательные типы

Указатель является величиной, указывающей на некоторый адрес в памяти, где хранятся какие-то данные. Указатели бывают *типизированные*, указывающие на данные определенного типа, и *нетипизированные* (типа `pointer`), которые могут указывать на данные произвольного типа. Используются указатели обычно для работы с объектами в динамически распределяемой области памяти, особенно при работе с записями.

Объявление указателя имеет вид:

```
type <имя типа указателя> = ^<тип данных>;
    F1 = ^real;     // типизированный указатель
    F2 = pointer;  // нетипизированный указатель
```

Предопределена операция `@`, возвращающая адрес своего операнда: переменной, функции, процедуры, метода. Иначе говоря, операция `@` создает указатель на свой операнд. При этом если `X` – переменная, то `@X` возвращает адрес `X`, причем при директиве компилятора `{$T-}` (включена по умолчанию) тип результата `pointer`. В случае функции или процедуры с именем `F` – `@F` возвращает точку входа, причем тип результата всегда `pointer`. Также существует предопределенная констан-

та **nil**, которая обычно присваивается указателям, которые в данный момент ни на что не указывают.

Чтобы получить доступ к данным, на которые указывает типизированный указатель, надо применить операцию его разыменования. Она записывается с помощью символа `^`, помещаемого после указателя. Например, для того, чтобы по адресу объявленного типизированного указателя на тип `real` `F1` находилось какое-либо число, необходим оператор вида `F1^:=0.18;`. Операция разыменовывания не применима к типу `pointer`. Для разыменовывания подобного указателя сначала необходимо привести его к другому типу.

Процедурные типы данных также являются указателями на любые определенные в программе процедуры и функции. При их использовании необходимо следить за соответствием параметров, передаваемых переменной процедурного типа, формальному набору параметров функции или процедуры, на тип которой указывает процедурная переменная.

3.3.9. Вариантные типы

В переменных типа `variant` могут храниться данные любых типов, кроме структур и указателей. Тип `variant` имеет смысл использовать в тех случаях, когда тип того или иного объекта заранее не известен или когда какие-то функции и процедуры требуют именно такой тип аргументов. При этом надо иметь в виду, что затраты памяти и времени на работу с переменными типа `variant` больше, чем при работе с обычными типами. К тому же недопустимые операции с переменными этого типа приводят к ошибкам времени выполнения, тогда как аналогичные недопустимые операции с переменными других типов выявляются на этапе компиляции.

Переменные типа `variant` занимают 16 битов и содержат код типа и значение переменной или указатель на это значение. Узнать действительный тип значения переменной `variant` можно с помощью функции `VarType`, возвращающей значение

типа `TVarData`, содержащее код типа. Переменные типа `variant` могут использоваться в выражениях как операнды любых операций, кроме `^`, **`is`** и **`in`**, совместно с другими величинами типов `variant`, `integer`, `real`, **`string`**, `boolean`. Компилятор в этих случаях автоматически осуществляет приведение типов величин в выражении.

3.3.10. Объекты, классы и интерфейсы

Класс – это тип данных, определяемый пользователем. То, что в Delphi имеется множество предопределенных классов, не противоречит этому определению – ведь разработчики Delphi тоже пользователи Object Pascal. Понятие *объекта* (object) использовалось при зарождении объектно-ориентированного программирования при переходе от Pascal к Object Pascal. В настоящее время тип **`object`** практически эквивалентен классу. В научно-технической литературе можно встретить иную интерпретацию термина "*объект*" – как конкретный *экземпляр* или инициализированную переменную класса.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Синтаксис объявления следующий.

```
Type
 <имя класса> = Class (<имя класса - родителя>)
  public {доступно всем}
   <поля, методы, свойства, события>
  published {видны в Инспекторе Объекта и изменяемы}
   <поля, свойства>
  protected {доступ только потомкам}
   <поля, методы, свойства, события>
  private {доступ только в модуле}
   <поля, методы, свойства, события>
 end;
```

Имя класса может быть любым допустимым идентификатором, однако идентификаторы большинства классов принято начинать с символа "`T`". Имя класса-родителя может не ука-

зываться. Тогда предполагается, что данный класс является непосредственным наследником `TObject` – наиболее общего из предопределенных классов. Таким образом, эквивалентны следующие объявления:

```
type TMyClass = class
 ...
end;

type TMyClass = class(TObject)
  ...
end;
```

Класс наследует поля, методы, свойства, события от своих предков и может отменять какие-то из этих элементов класса или вводить новые. Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены.

Раздел **`public`** (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел **`published`** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел **`private`** (закрытый) содержит объявления полей, процедур и функций, используемых только внутри данного класса. Раздел **`protected`** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако, в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные объекты, причем не требуется, чтобы производные объекты объявлялись в этом же модуле.

Объявления полей выглядят так же, как объявления переменных или объявления полей в записях. Объявления мето-

дов в простейшем случае также не отличаются от обычных объявлений процедур и функций.

Интерфейсы во многом напоминают интерфейсную часть классов. Но они содержат только объявления методов и не содержат их реализацию, которая должна осуществляться в классах, поддерживающих данный интерфейс. Однако в приложении могут создаваться переменные типа интерфейса. Через них можно обращаться к объявленным в интерфейсе методам объектов классов, использующих данный интерфейс, точно так же, как это делается при непосредственном обращении к объектам. Особенность заключается в том, что при обращении через интерфейс можно работать с объектами разных классов (в том числе с объектами классов, не имеющими общего предка), в которых определены используемые методы. Таким образом, возможна реализация *полиморфного поведения* объектов совершенно различных классов. Необходимо отметить, что интерфейсы обеспечивают возможность множественного наследования, отсутствующую в классах Object Pascal.

Объявление интерфейса имеет вид:

```
type
<имя интерфейса>=interface(<имя интерфейса-родителя>)
  ['GUID']
  <список элементов>
end;
```

Имена интерфейсов принято начинать с символа "`I`". Имя интерфейса-родителя может отсутствовать. В этом случае интерфейс наследует непосредственно классу `IUnknown` – базовому классу всех интерфейсов. Интерфейс, подобно классу, наследует от своих родителей все методы, но не наследует их реализацию. Может также отсутствовать указание `GUID` – глобального уникального идентификатора, который задается строкой шестнадцатеричных символов (от 0 до F) вида **`['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}'].`**

Объявление интерфейса в основном аналогично объявлению класса и имеет следующие ограничения:

- список элементов может включать только методы и свойства (с соответствующими методами для чтения и записи), но не может включать поля;
- все методы интерфейса открытые (**public**), т.е. при объявлении метода необходима инструкция **stdcall**;
- поскольку реализация методов интерфейса определяется в классе, поддерживающем данный интерфейс, то интерфейсные методы не могут быть виртуальными, динамическими, абстрактными, не могут перегружаться, также интерфейс не имеет конструктора и деструктора.

3.4. Операторы языка Object Pascal

3.4.1. Оператор присваивания

Оператор присваивания записывается в виде: `<переменная> := <выражение>;` где `<переменная>` – это переменная или типизированная константа любого типа, включая разыменованный указатель, объект, структуру и т.д., а `<выражение>` – любое допустимое выражение, совместимое по типу с переменной в левой части оператора. Оператор вычисляет значение выражения, записанного как правый операнд операции присваивания `:=`, и присваивает полученное значение переменной в левой части оператора. Например, оператор `I:=3` присваивает переменной `I` значение `3`, а оператор `I:=I+1` увеличивает значение переменной `I` на `1`.

Применительно к объектам надо четко представлять различие между оператором присваивания и методом копирования параметров `Assign`, свойственным многим классам.

3.4.2. Оператор безусловного перехода

Оператор **goto** позволяет прервать обычный поток управления и *передать управление в произвольную точку кода*, помеченную *меткой*. В тексте программы метка отмечает точ-

ку, в которую передается управление, и может располагаться в любом месте блока, как после оператора **goto**, так и до него. Точка, в которую может передаваться управление, помечается именем метки, после которого следует двоеточие. Например:

```
L1: <оператор, на который передается управление>
```

Сам оператор `goto` имеет форму:

```
goto <метка>;
```

Все три элемента конструкции: объявление меток, сами метки и операторы **goto**, передающие на них управление, должны размещаться в пределах одного блока.

3.4.3. Оператор if

Оператор условного перехода **if** предназначен для выполнения тех или иных действий в зависимости от истинности или ложности некоторого условия. Условие задается выражением, имеющим результат типа `boolean`.

В общем виде оператор условного перехода имеет вид:

```
if <условие> then <оператор1> else <оператор2>;
```

Если условие возвращает `true`, то выполняется первый из указанных операторов, в противном случае выполняется второй оператор. Заметим, что в конце первого оператора перед ключевым словом `else` точка с запятой не ставится. Ветвь с **else** может отсутствовать. Тогда при невыполнении условия управление передается следующему за конструкцией **if** оператору. Например, в результате выполнения операторов

```
C := A;
if B > A then C := B;
```

переменная `C` станет равна максимальному из чисел `A` и `B`, поскольку оператор `C:=B` будет выполнен только при `B>A`.

При вложенных конструкциях **if** могут возникнуть неоднозначности в понимании того, к какой из вложенных конструкций **if** относится элемент **else**. Компилятор всегда считает, что **else** относится к последней из конструкций **if**, в которой не было раздела **else**. Например, в конструкции

```
if <условие1> then if <условие2> then <оператор1>
              else <оператор2>;
```

else будет отнесено компилятором ко второй конструкции **if**, т.е. <оператор2> будет выполняться в случае, если первое условие истинно, а второе ложно. Иначе говоря, вся конструкция будет прочитана как

```
if <условие1> then begin
  if <условие2> then <оператор1> else <оператор2>
                end;
```

Если же необходимо отнести **else** к первому **if**, то это надо реализовывать как *составной оператор*, т.е. записать в явном виде с помощью *операторных скобок* **begin...end.**

```
if <условие1> then begin
                 if <условие2> then <оператор1>
                 end
               else <оператор2>
```

3.4.4. Оператор case

Оператор множественного выбора **case** позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнять те или иные действия. В общем случае формат записи оператора **case** следующий:

```
case <выражение> of
 <список значений 1>: <оператор 1>;
  ...
 <список значений n>: <оператор n>;
else
 <оператор>
end;
```

В этой конструкции выражение должно иметь порядковый тип. Поэтому, например, нельзя использовать выражения, возвращающие действительные числа или строки.

Списки значений могут содержать одно или несколько разделенных запятыми возможных значений константных выражений. После списка ставится двоеточие, а затем пишется оператор (составной оператор), который должен выполняться, если выражение приняло одно из значений, перечисленных в списке. После выполнения этого оператора работа структуры **case** завершается, и управление передается оператору, следующему за этой конструкцией. Операторы всех последующих разделов не выполняются.

Если значение выражения не соответствует ни одному из перечисленных во всех списках, то выполняется оператор, следующий после ключевого слова **else**. Впрочем, раздел **else** не является обязательным. В этом случае, если в списках не нашлось соответствующего значения выражения, то ни один оператор не будет выполнен.

Списки могут содержать константы и константные выражения, которые совместимы по типу с объявленным выражением и которые компилятор может вычислить заранее, до выполнения программы. Допустимы ограниченных типы, но нельзя использовать переменные и многие функции. Повторения в списках одних и тех же значений не допускается, поскольку в этом случае выбор был бы неоднозначным.

3.4.5. Организация цикла с помощью оператора for

Оператор **for** обеспечивает *циклическое повторение* некоторого оператора или группы операторов – *тела цикла* – заданное число раз. Повторение цикла определяется некоторой локальной *управляющей* переменной порядкового типа (*счетчиком*), которая увеличивается или уменьшается на единицу при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения. Оператор записывается в одной из следующих форм.

```
for <счетчик>:=<начальное значение>
     to <конечное значение> do <оператор>;

for <счетчик>:=<начальное значение>
     downto <конечное значение> do <оператор>;
```

В начале выполнения оператора **for** счетчику присваивается <начальное значение>. После каждого очередного выполнения тела цикла <оператор> его значение увеличивается (в первой форме с **to**) или уменьшается (во второй форме с **downto**) на единицу. Когда значение управляющей переменной достигает значения <конечное значение>, тело цикла выполняется последний раз, после чего управление передается оператору, следующему за структурой **for**. Начальные и конечные значения являются выражениями, совместимыми по типу с управляющей переменной. Если они равны друг другу, тело цикла выполняется только один раз. Если в форме с **to** начальное значение больше конечного или в форме с **downto** начальное значение меньше конечного, то тело цикла не выполняется ни разу. Внутри цикла значение счетчика может использоваться в выражениях, но после выполнения цикла значение управляющей переменной использовать не рекомендуется.

3.4.6. Цикл repeat … until

Структура **repeat...until** используется для организации *циклического выполнения совокупности операторов*, составляющих тело цикла, до тех пор пока не выполнится некоторое условие. Синтаксис структуры:

```
repeat
  <операторы тела цикла>
until <выражение условия>;
```

Структура работает следующим образом. Выполняются операторы тела цикла. Затем вычисляется <выражение условия>, которое должно возвращать результат логического типа. Если выражение возвращает `false`, то повторяется выполне-

ние операторов тела цикла. Циклическое повторение продолжается до тех пор, пока проверяемое выражение не вернет `true`. После этого цикл завершается и управление передается оператору, следующему за структурой **`repeat...until`**. Поскольку проверка выражения осуществляется после выполнения операторов тела цикла, то эти операторы заведомо будут выполнены хотя бы один раз, даже если выражение сразу истинно. Отметим, что точка с запятой перед ключевым словом **`until`** может быть опущена.

3.4.7. Цикл while … do

Структура организации циклов **`while...do`** имеет следующий синтаксис.

```
while <выражение условия> do <оператор>;
```

Структура работает следующим образом. Сначала вычисляется `<выражение условия>` логического типа. Если выражение возвращает `true`, то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет `false`. После этого цикл завершается и управление передается оператору, следующему за структурой **`while...do`**. Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза. В этом основное отличие структуры **`while...do`** от структуры **`repeat...until`**, в котором тело цикла заведомо выполняется хотя бы один раз.

3.4.8. Дополнительные операторы организации циклов

В некоторых случаях желательно прервать выполнения цикла, проанализировав какие-то внутренние условия. Это реализуется с помощью оператора `break`. Он прерывает выполнение тела любого цикла **`for`**, **`repeat`** или **`while`** и передает управление следующему за циклом выполняемому оператору.

Отметим, что еще один способ прерывания – использование оператора **goto**, передающего управление на метку оператора, расположенного вне тела цикла.

Для прерывания циклов, размещенных в процедурах или функциях, можно воспользоваться процедурой `exit`. В отличие от оператора `break`, процедура `exit` прервет не только выполнение цикла, но и выполнение той процедуры или функции, в которой расположен цикл. Прервать выполнение цикла, а заодно – и блока, в котором расположен цикл, можно также генерацией какого-то исключения. Наиболее часто в этих целях используется процедура `abort`. Наконец, процедура `halt` инициирует не только выход их цикла, но и аварийно завершает исполняемую программу.

Имеется также процедура `Continue`, которая прерывает только выполнение текущей итерации текущего выполнения тела цикла и передает управление на следующую итерацию.

3.4.9. Оператор with...do

Оператор **with...do** используется для *сокращения кода* при обращении к полям записей или к свойствам и методам объекта. Применение этого оператора позволяет избежать повторных ссылок на объект в последующих операторах. Оператор **with...do** может записываться следующим образом:

```
with <объект> do <оператор>;
```

В операторе, следующем за ключевым словом **do**, можно для полей, свойств и методов объекта, указанного как `<объект>`, не включать ссылки на этот объект. При этом каждый идентификатор в операторе, который совпадает с именем поля, свойства, метода объекта, трактуется как относящийся к этому объекту, и к нему неявно добавляется ссылка на этот объект.

Возможна другая форма этого оператора, соответствующая множеству вложенных друг в друга конструкций **with**.

```
with <объект 1>, ..., <объект n> do <оператор>;
```

3.5. Обработка исключительных ситуаций

Важным и принципиальным отличием Object Pascal от Pascal является *механизм обработки исключительных ситуаций* (исключений).

Исключение – это объект специального вида, возникающий, как правило, во время функционирования ПО как реакция на какие-либо ошибки. Это может быть деление на ноль, выход операнда за диапазон и т.д., – словом, *ошибки времени выполнения программы*. Исключения могут генерироваться *автоматически* или *целенаправленно*. Могут также содержать в виде свойств некоторую уточняющую информацию, т.е. характеризовать исключительную ситуацию. Особенностью исключений является то, что это сугубо временные объекты, и после обработки каким-то обработчиком они разрушаются.

Стандартной реакцией ПО на большинство исключений является выдача пользователю краткой информации в окне сообщений и уничтожение экземпляра исключения (обеспечивается методом `HandleException`). После этого работа программы может быть продолжена.

Механизм обработки исключений предназначен для реализации иной реакции приложения на ошибки времени выполнения. Для этого используются структуры **`try...except`** и **`try...finally`**.

Первая имеет следующий синтаксис.

```
try
  ...{Исполняемый код}
except
on ...<реакция на событие 1>;
on ...<реакция на событие 2>;
   ...
else
  ...<обработчик всех не перехваченных ранее событий>;
end;
```

Операторы, расположенные в разделе **except**, выполняются только в том случае, если в операторах раздела **try** (или в процедурах и функциях, вызванных этими операторами) возникла ошибка и, следовательно, было сгенерировано исключение. Раздел **except** может использоваться двумя способами. При первом способе в нем могут располагаться любые выполняемые операторы, кроме обработчиков исключений **on...do**. Это могут быть операторы сообщений об ошибке, операторы освобождения ресурсов и т.д. Второй и главный способ использования раздела **except** – обработка исключений. В этом случае в него могут включаться только обработчики исключений: операторы **on...do** и необязательный обработчик любых исключений, предваряемый ключевым словом **else**, как показано в примере.

Оператор **on...do** перехватывает исключения указанного класса и всех производных от него классов. Поэтому при его использовании важно представлять себе иерархию классов исключений. Структура работает следующим образом. Если при выполнении операторов раздела **try** генерируется исключение, выполнение этого раздела прерывается и управление передается операторам раздела **except**. Если среди обработчиков встретился соответствующий сгенерированному исключению, выполняется оператор этого обработчика, исключение разрушается и управление передается оператору, следующему за блоком **on...do** (т.е. оператору, следующему за последним оператором блока **end**). Если же обработчик не найден, то управление передается на следующий уровень, т.е. разделу **except** следующего обрамляющего блока **try...except** (если таковой есть). То же самое происходит, если в процессе обработки генерируется новое исключение (при обработке ошибки произошла новая ошибка). Если в результате прохода по всем уровням подходящий обработчик так и не будет найден, произойдет обработка системным обработчиком исключений.

Оператор **on...do** имеет две формы:

```
on < класс исключения> do <оператор>;

on <имя>: <класс исключения>
        do <операторы, в которых можно использовать
конструкцию <имя>.<имя свойства>>
```

Первая форма оператора не позволяет получить доступ к свойствам исключения. Обрабатывается только указанный класс исключений (или исключения, производные от этого класса), обеспечивая индивидуальную реакцию на каждый вид исключений. Таким образом, поскольку семейство исключений построено по иерархическому принципу, можно, обратившись к базовому классу некоторой группы родственных исключений, обработать сразу всю группу.

Вторая форма создает временное имя исключения и позволяет через него с помощью конструкции `<имя>.<имя свойства>` иметь доступ к свойствам исключительной ситуации.

Обработчик, начинающийся с ключевого слова **else**, обрабатывает исключения всех классов. Если он используется, то помещается *последним* в разделе **except** блока **try...except**, так как после него любое исключение разрушается и, следовательно, никакой другой обработчик исключений не сработает. Обработчик **else** позволяет не вводить обработчики **on...do** всех возможных исключений, которые, следовательно, могут быть введены только для тех исключений, которые требуется обработать особым образом. Остальные могут обрабатываться одним общим обработчиком. При этом с помощью функций `ExceptObject`, `ExceptAddr`, `ShowException`, `ClassName` и других может быть получен доступ к объекту исключения и выдана информация о типе исключения.

Следует предупредить о некоторой опасности применения обработчика **else**. Поскольку он обрабатывает все исключения, то, если при этом не выдается никаких сообщений, возможна маскировка ошибок, что может существенно затруд-

нить отладку приложения. Следует также отметить, что **else** – не единственный способ реализации обработчика любых исключений. Те же функции можно реализовать с помощью обработчика **on...do**, примененного к базовому классу исключений `Exception`. Все предопределенные в Delphi классы исключений являются прямыми или косвенными наследниками этого класса. Пользователь может создавать и свои собственные, производные от `Exception`, классы исключений.

В структуре **try...finally** программный код, относящийся к секции **finally**, выполняется при любых условиях, даже если при выполнении операторов раздела **finally** случилась ошибка, операторы раздела все-таки выполняются до конца. Обычно в этот раздел помещают необходимые операторы "зачистки": освобождения динамически выделенной памяти, закрытия ненужных файлов, удаления временных файлов и т.д. Если не помещать такие операторы в защищенный раздел **finally**, то из-за прерывания выполнения вследствие возможных ошибок они могут оказаться не выполненными, что приведет к "утечке ресурсов".

Отметим, что блоки **try...except** никак не связаны с блоками **try...finally**. На практике эти виды блоков часто используются совместно в виде следующей структуры:

```
try       {начало блока try...finally}
.............
  try     {начало блока try...except}
   .............
  except
   .............
  end;    {конец блока try...except}
finally
.............
end;      {конец блока try...finally}
```

3.6. Процедуры и функции

Процедуры и *функции* представляют собой блоки программного кода, имеющие точно такую же структуру, как и программа (единственное отличие заключается в том, что процедуры и функции не могут содержать раздел `uses`).

В общем виде структура программной реализации процедуры следующая.

```
procedure MyProcedure(<список параметров>);
const // раздел описания локальных констант
type  // раздел описания локальных типов
var   // раздел описания локальных переменных
begin
  ... // программный код процедуры
end;
```

Функции отличаются от процедур только тем, что их идентификатор возвращает некоторое значение, поэтому при описании необходимо через двоеточие указать тип возвращаемого результата.

```
function MyFunc(<список параметров>):<тип результата>;
  ... // разделы описаний
begin
  ... // программный код функции
end;
```

Заголовок процедуры (функции) состоит из зарезервированного слова **`procedure`** (**`function`**), идентификатора и списка параметров, заключенного в круглые скобки (список параметров не обязателен, можно создавать процедуры или функции без параметров). Указываемые в заголовке параметры называются *формальными* и предназначены для обмена данными между основной программой и процедурой (функцией). В списке указывается один или несколько идентификаторов параметров и через двоеточие – тип. Параметры разных типов отделяются друг от друга точкой с запятой.

```
procedure prog_id(par1,par3:integer; param3:real);
```

Количество передаваемых формальных параметров не ограничено, причем внутри процедуры или функции они представляют собой обычные переменные или константы.

Вызов осуществляется с помощью оператора вызова, состоящего из идентификатора процедуры (функции) и списка *фактических* параметров, перечисляемых через запятую, которые представляют собой переменные или константы, описанные в разрабатываемой программе. Передача параметров производится через стек и может выполняться *по значению*, *по ссылке* или объявлением параметра как *выходного*.

При *передаче параметра по значению* в стек копируется значение переменной, соответствующей фактическому параметру. Подобный способ передачи принят по умолчанию, т.е. нет необходимости в использовании никаких дополнительных ключевых слов. Значение параметра, переданного по значению, можно изменять внутри процедуры или функции, однако в программу это изменение передаваться не будет. При передаче параметров по значению необходимо соблюдать осторожность, так как при передаче больших структур данных возможно переполнение стека, т.е. возможно возникновение ошибки времени выполнения `Stack overflow`.

В случае второго способа передачи параметров – *по ссылке* – в стек заносится не значение параметра, а его адрес. Таким образом, независимо от объема памяти, занимаемого переменной, в стеке будет занято только 4 байта. При передаче параметра по ссылке перед его идентификатором в списке формальных параметров указывается одно из ключевых слов: **`var`** или **`const`**. В первом случае параметр называется *параметром-переменной,* во втором – *параметром-константой.*

Значения параметров-переменных в тексте процедуры (функции) можно изменять, и эти изменения будут передаваться в программу. Таким образом, параметры-переменные используются для передачи данных в вызывающую програм-

му. Значения параметров-констант внутри структуры изменять нельзя. Попытка присвоить параметру-константе какое-нибудь значение приведет к ошибке компиляции.

Еще одним способом изменения внутри процедуры или функции параметров, объявленных в ПО, является использование ключевого слова **out**. Оно помечает формальный параметр как *выходной*. Отличие от передачи по ссылке в том, что при этом не гарантируется передача начального значения. Память, занимаемая аргументом, очищается в момент обращения к процедуре. Таким образом, параметр, передаваемый как выходной, указывает на некоторое место в памяти, являющейся просто контейнером, куда процедура должна занести соответствующее выходное значение.

Тип формального параметра в заголовке можно не указывать, например: **procedure** `MyProg(`**var** `A)`. В этом случае в качестве фактического параметра может быть использована переменная любого типа. Параметры без типа передаются только по ссылке. При использовании параметров без типа компилятор не может произвести проверку соответствия типов. Поэтому программист должен сам следить за соответствием типов во избежание возникновения ошибок во время выполнения программы (`runtime error`).

Формальные параметры могут быть открытыми массивами. В этом случае можно передавать в качестве фактического параметра массивы разной длины. Параметр, имеющий тип открытого массива, объявляется следующим образом:

```
procedure MyProc(var param:array of Mytype);
```

К открытому массиву внутри процедуры можно обращаться только *поэлементно*. Открытые массивы можно передавать как по ссылке, так и по значению. В последнем случае фактический массив записывается в стек, что может привести к ошибке `Stack overflow`.

В качестве примера работы с открытым массивом приведем функцию [1], которая возвращает строку при передаче в нее аргументов различных типов – строк, целых и действительных чисел, булевых величин, указателей.

```
function ToStr(A: array of const):string;
var i:integer;
begin
 Result:='';
 for i:=0 to High(A) do With A[i] do
  case VType of
    vtString  :Result:=Result+VString^+'  ';
 vtAnsiString:Result:=Result+String(VAnsiString)+'  ';
    vtPChar   :Result:=Result+VPChar+'  ';
    vtChar    :Result:=Result+VChar;
vtExtended:Result:=Result+FloatToStr(VExtended^)+'  ';
    vtInteger :Result:=Result+IntToStr(VInteger)+'  ';
    vtBoolean :if VBoolean
                  then Result:=Result+'true'+'  '
                  else Result:=Result+'false'+'  ';
    vtObject  :Result:=Result+VObject.ClassName+'  ';
    vtClass   :Result:=Result+VClass.ClassName+'  ';
 vtCurrency:Result:=Result+CurrToStr(VCurrency^)+'  ';
    vtVariant :Result:=Result+string(VVariant^)+'  ';
    vtInt64   :Result:=Result+IntToStr(VInt64^)+'  ';
  end; {case VType}
end;
```

Вызывать функцию необходимо, используя конструктор открытого массива, в который можно включать самые разные элементы.

Для параметров в объявлениях функций и процедур могут задаваться *значения по умолчанию*. Этот механизм позволяет передавать не все необходимые аргументы. Значения по умолчанию задаются добавлением в конце объявления параметра знака равенства, после которого записывают константное выражение. Аргументы по умолчанию рекомендуется назначать последними в списке параметров, т.к. при вызове процедуры или функции пропускать параметры нельзя.

Идентификатор функции обычно используется в выражении. В теле функции для возвращения значения используется как сам идентификатор функции, так и неявно определенный идентификатор `Result`, с которым внутри функции можно работать как с обычной переменной. Идентификатору функции можно только присваивать значения, но нельзя использовать в выражениях в теле функции, иначе функция будет повторно вызывать сама себя. Подобный прием в программировании носит название *рекурсия*, но пользоваться им надо осторожно, ибо, например, бесконечный вызов функции может привести к переполнению стека.

Процедуры и функции могут содержать не только разделы объявления констант, типов переменных, но и включать в себя объявления функций и процедур. Все эти идентификаторы, называемые *локальными*, видны только внутри тех процедур и функций, в которых они объявлены.

В одной и той же области видимости можно определить *перезагружаемые* функции или процедуры. Они имеют одинаковые имена, но различаются по числу и типу параметров. Для объявления функции (процедуры) как перезагружаемой используется ключевое слово **`overload`**. Рассмотрим пример.

```
Function Divide(X,Y:real):real; overload;
Begin
  Result:=Y/X;
End;

Function Divide(X,Y:integer): integer; overload;
Begin
  Result:=Y div X;
End;
```

Обе функции объявлены как перезагружаемые с именем `Divide`. Первая получает действительные аргументы, вторая – целые. Соответственно вызов `Divide(3,7)` приведет к вызову второго варианта. Первый вариант будет вызван при использовании другого программного кода – `Divide(3.0,7.0)`.

4. ИНТЕГРИРОВАННАЯ СРЕДА DELPHI

4.1. Общий внешний вид и основные возможности

В Delphi интегрированная среда разработки (IDE – *Integrated Development Environment*) – это удобная среда быстрой разработки сложных прикладных программ, в которой есть все необходимые инструменты (*tools*) для проектирования, запуска и тестирования разрабатываемых программных средств и все нацелено на облегчение этих процессов. IDE интегрирует в себе *редактор кодов*, *отладчик*, *инструментальные панели*, *визуальный редактор форм*, *редактор графических ресурсов*, *инструментарий баз данных* и многое другое – все, с чем приходится работать при проектировании ПО. Эта интеграция предоставляет разработчику хорошо сбалансированный набор инструментов, дополняющих друг друга. Естественно, существует возможность расширять возможности (меню) IDE, включая в нее необходимые дополнительные инструменты, в том числе и разработанные самостоятельно.

Запуск Delphi осуществляется файлом `delphi32.exe` или пиктограммой в разделе меню Windows `Пуск | Программы`. Внешний вид IDE Delphi версии 7.0 показан на рис. 4.1. В верхней части находится *главное меню*. Ниже расположены две *инструментальные панели.* Левая панель содержит *быстрые кнопки,* дублирующие некоторые наиболее часто используемые команды меню. Правая панель содержит *палитру библиотеки визуальных компонентов* VCL (*Visual Component Library*), состоящую из ряда страниц, закладки которых видны в верхней части палитры. Правее полосы главного меню размещена панель сохранения и выбора различных конфигураций, содержащая выпадающий список и две быстрые кнопки.

В основном поле окна слева находится окно *инспектора объектов* (*Object Inspector*), с помощью которого задаются свойства компонентов (вкладка `Properties`) и обработчики событий (вкладка `Events`).

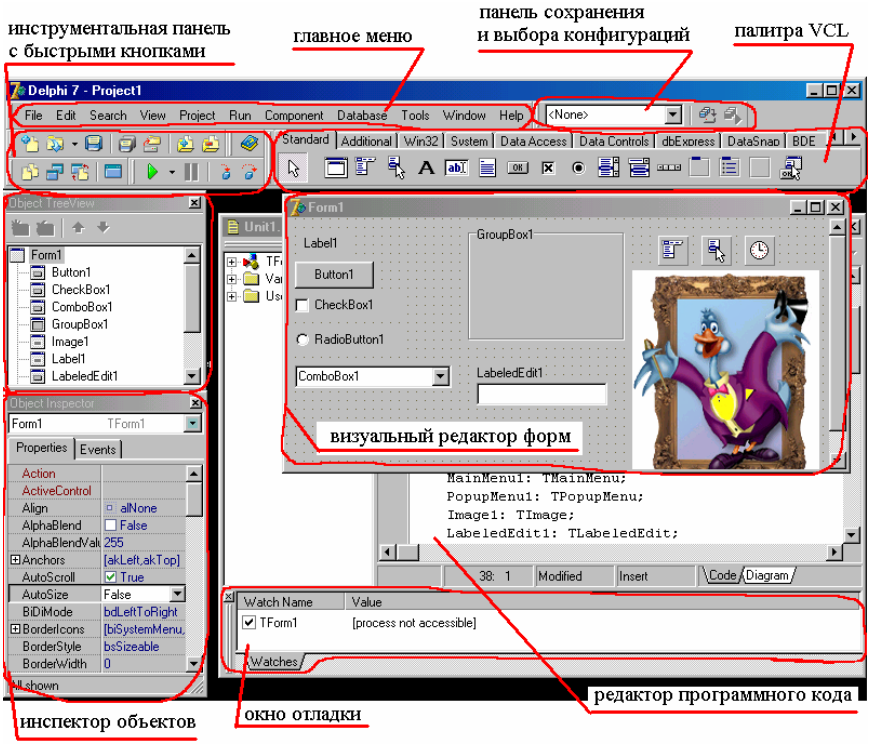


Рис. 4.1. Внешний вид IDE Delphi версии 7.0

Правее расположено окно *редактора программного кода* (*Code Explorer*), который является обычным текстовым редактором, ориентированным на написание текстов программ.

Важным инструментом разработки интерфейса приложений является *визуальный редактор форм* (*Form Designer*). С его помощью можно разместить на форме проектируемого приложения различные *компоненты*, как показано на рис. 4.1, модифицировать некоторые *свойства компонентов* и самой формы, установить типовые *обработчики событий*. Напомним, что компоненты являются основой компонентной архитектуры построения приложений (см. пп. 1.1.5). *Обработчик события* – это процедура, предназначенная для создания реак-

ции формы или компонента на какое-либо воздействие. Возможные события объектов перечислены на вкладке *Events* инспектора объектов.

В интегрированной среде разработки Delphi существует достаточно много различных окон, используемых в процессе разработки ПО. Например, окно *менеджера проекта* (*Project Manager*) или показанное на рис. 4.1 окно *наблюдаемых величин* (*Watch List*). Поэтому в IDE, как и в оконных компонентах Delphi, широко используется технология *Drag&Doc* – перетаскивание и встраивание окон. Встраивание окон позволяет экономить площадь экрана. Такое окно можно отличить от обычного по следующим признакам:

- сокращенная полоса системного меню, включающая обычно только кнопку закрытия окна;
- наличие в контекстном меню, всплывающем при щелчке в окне правой кнопкой мыши, переключателя *Dockable* – встраиваемое (если снять метку с этого переключателя, окно перестанет быть встраиваемым);
- при перетаскивании встраиваемого окна размеры его рамки изменяются, если окно перемещается в пределах другого окна.

4.2. Главное меню

Главное меню (рис. 4.1) содержит полный набор команд, необходимых для работы в интегрированной среде Delphi. Для полной характеристики всех команд лучше обратиться к справочной литературе [1] или выделить конкретный пункт меню и с помощью клавиши F1 получить контекстную справку.

Поскольку Delphi является визуальной средой проектирования, то частого обращения к командам главного меню, как правило, не требуется. Наиболее востребованные команды в Delphi вынесены на инструментальные панели быстрых кнопок (рис. 4.1). Их назначение можно узнать из ярлычков, появляющихся при задержке курсора мыши над соответствующей

кнопкой. Разработчик может изменить набор инструментов на панелях с помощью команды `Vien | Toolbars`.

Приведем описание основных команд главного меню и (при наличии) соответствующие им пиктограммы быстрых кнопок и *"горячие"* клавиши активации команды с клавиатуры.

4.2.1. Меню File

Часть команд меню `File` – это обычные команды работы с файлами: `Open` (), `Save` (, `Ctrl+S`), `Save as` (), `Close` (), `Exit` (), назначение которых пояснять не требуется.

Команды `Open Project` () и `Save Project As` (, `Ctrl+F11`) используются для открытия проектов и их сохранения под другим именем. Под *проектом* в Delphi понимается набор файлов, необходимых для разработки программного обеспечения или динамически связываемой библиотеки.

Команды `Save All` (, `Shift+Ctrl+S`) и `Close All` () предназначены для сохранения и закрытия всех открытых файлов, относящихся к активному проекту.

Команда `Use Unit` (, `Alt+F11`) используется для подключения к проекту дополнительных модулей, открытых в редакторе кода.

Команда `Print` () предназначена для печати формы или программы.

Для создания новых проектов и других файловых элементов программных средств используется команда `New`, с которой связано подменю, позволяющее создать, например, приложение – `Application` (), форму – `Form` (), программный модуль `Unit` (), модуль данных – `Data Module` (), фрейм – `Frame` () и т.п. Выполнение команды `Other` () приводит к вызову окна диалога (рис. 4.2), которое позволяет создать файл любого типа. Здесь разработчику предлагается набор объектов, содержащихся в специальном *депозитарии – хранилище объектов* (*Object Repository*).

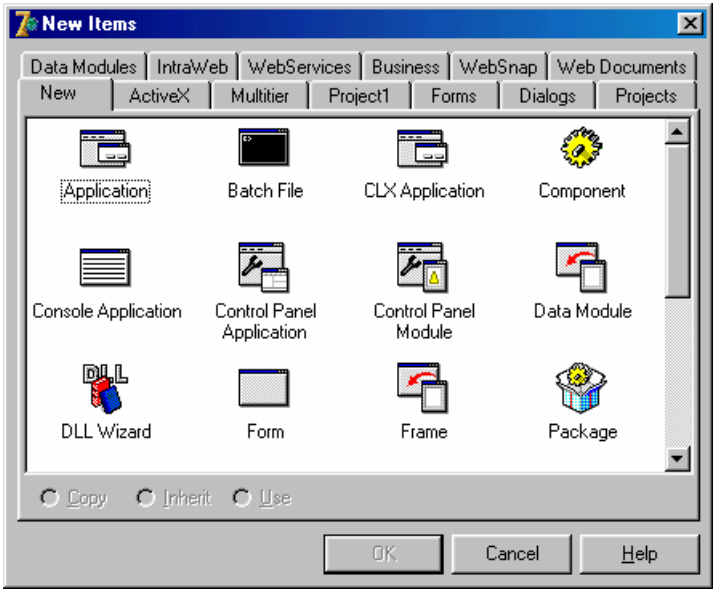


Рис. 4.2. Окно хранилища объектов

4.2.2. Депозитарий – хранилище объектов

Среда проектирования программных средств Delphi позволяет разрабатывать самые различные проекты – исполняемые приложения, библиотеки кода, компоненты Delphi, элементы `ActiveX` и т.п. *Депозитарий* – это специальное хранилище объектов, содержащее шаблоны программного кода, которые могут быть использованы в качестве основы при разработке программных средств.

Шаблоны объектов хранилища разделены на группы, отображаемые на разных вкладках окна `New Items`. При выборе некоторых объектов (например, расположенных на вкладках `Forms`, `Dialogs` или `Data Modules`) активизируется один из переключателей, расположенных в нижней части окна. Таким образом задается способ использования выбранного шаблона:

➢ `Copy` – в проект добавляется копия выбранного объекта. При этом изменения, вносимые в экземпляр объекта в проекте, не отражаются на объекте, содержащемся в хранилище.

Также и последующее изменение шаблона объекта хранилища не повлияет на функциональность объекта в проекте.

- `Inherited` – в проект добавляется копия выбранного объекта. Изменения встроенного в проект объекта не отражаются на шаблоне депозитария, но изменение шаблона изменит объект, используемый в проекте.
- `Use` – используется для модификации объектов депозитария. Изменения в проекте сразу же вносятся в шаблон объекта из хранилища и во все другие проекты, использующие данный шаблон.

В депозитарий можно добавлять свои шаблоны и удалять существующие (только на вкладках `Forms`, `Dialogs`, `Projects`, `Data Modules` и `Business`). С помощью команды `Properties` из контекстного меню, отображаемого нажатием правой кнопки мыши в окне диалога `New Items`, можно создавать новые вкладки.

4.2.3. Меню Edit и команды контекстного меню визуального редактора форм

Меню `Edit` содержит ряд стандартных команд редактирования, обычно используемых в прикладных программах в операционной среде Windows:

- `Undo` (, `Ctrl+Z`) – отменить последнее действие;
- `Redo` (, `Shift+Ctrl+Z`) – повторить;
- `Cut` (, `Ctrl+X`) – вырезать выделенный текст, объект или группу объектов;
- `Copy` (, `Ctrl+C`) – копировать;
- `Paste` (, `Ctrl+V`) – вставить;
- `Delete` (, `Ctrl+Del`) – удалить;
- `Select All` (`Ctrl+A`) – выделить все.

Остальные команды используются при проектировании интерфейса форм и дублируют команды контекстного меню визуального редактора форм.

Команда `Align to Grid` выравнивает размещенные на форме компоненты по узлам сетки.

Команды `Bring to Front` () – переместить наверх, и `Send to Back` () – переместить вниз, служат для изменения *Z-последовательности* компонентов. Первая команда помещает выделенный компонент поверх всех остальных (в начало *Z-последовательности*), вторая, соответственно, наоборот.

При выборе команды `Align` () появляется соответствующее диалоговое окно, опции которого позволяют выбрать ряд вариантов выравнивания компонентов на форме по горизонтали и вертикали.

Выбор раздела меню `Size` () позволяет изменять размеры компонента или группы компонентов до заданных значений ширины и высоты. Другая команда – `Scale` () – позволяет пропорционально изменять масштаб (увеличивать или уменьшать до ста раз) всего расположенного на форме. Для этого необходимо в появляющемся диалоговом окне задать в процентах масштабирующий коэффициент `Scaling factor`.

При выполнении программ в среде Windows нажатие клавиши `Tab` обычно приводит к последовательной смене активных компонентов формы. С помощью команды `Tab Order` () можно формировать эту последовательность в специальном окне, что удобнее, чем задавать свойства вручную.

Команда `Creation Order` () позволяет управлять последовательностью создания невизуальных компонентов, что может быть важно, если одни из этих компонентов используют свойства других, предполагая, что они существуют и инициализированы.

Раздел команд `Flip Children` позволяет зеркально преобразовать размещение (слева направо) компонентов формы.

После размещения и выравнивания компонентов их местоположение полезно заблокировать или зафиксировать командой `Lock Controls` (). Повторное использование команды приведет к разблокировке компонентов, но даже для забло-

кированных компонентов их местоположение можно изменить непосредственным заданием свойств в инспекторе объектов.

4.2.4. Меню Search

Команды меню `Search` используют для быстрого поиска и замены фрагмента программного кода. Доступны, например, следующие команды:

- `Find` ( , `Ctrl+F`) – найти;
- `Find in Files` () – поиск текстовых строк в нескольких файлах, в том числе относящихся не только к текущему проекту;
- `Relace` ( , `Ctrl+R`) – заменить;
- `Search Again` ( , `F3`) – повторный поиск;
- `Go to Line Number` ( , `Alt+G`) – перейти на строку с указанным номером.

При возникновении ошибки времени выполнения с помощью команды `Find Error` () можно найти соответствующую строку программного кода. Но эта возможность доступна, только если ошибка возникла в модуле разработчика.

4.2.5. Меню View

Команды меню `View` предназначены для управления отображением информации. Например, команда `Project Manager` ( , `Ctrl+Alt+F11`) открывает окно менеджера проекта, в котором отображаются все файлы, входящие в текущую группу проекта. Доступна возможность добавлять, удалять, сохранять целые проекты и отдельные файлы в проектах, активизировать конкретный проект.

Для активизации часто используемого окна инспектора объектов используется команда `Object Inspector` ( , `F11`).

Если на проектируемой форме находится сложная иерархия компонентов, то для их поиска удобно использовать специальное окно *Object TreeView* (рис. 4.1), которое вызывается командой `Object TreeView` ( , `Shift+Alt+F11`).

При разработке ПО в Delphi существует возможность *журнализации*, т.е. ведения списка задач, которые могут относиться к проекту в целом, к отдельному модулю проекта, к динамической библиотеке, входящей в группу проектов, и т.д. Для активизации соответствующего окна используется команда `To-Do List` (). Список задач не влияет на компиляцию проекта и используется как записная книжка, помогая планировать разработку программы и разделять задачи между разработчиками при коллективной разработке приложений.

Команда `Alignment Palette` () открывает панель инструментов для разнопланового выравнивания как отдельного компонента, так и группы компонентов в визуальном редакторе форм. Используется при разработке интерфейса ПО.

Команда `Browser` (, `Ctrl+Shift+B`) открывает окно *браузера проекта*, в котором приведена структура приложения, используемые классы, модули, переменные и т.д.

Команда `Code Explorer` () активизирует окно *браузера кода*, в котором визуализирована структура программы. По умолчанию это окно отображается совместно с *редактором кода*, как показано на рис. 4.1.

Активизировать любое из открытых окон Delphi можно с помощью соответствующего списка, вызываемого командой `Window List` (, `Alt+O`).

Группа команд `Debug Windows` используется для отладки приложений путем визуализации соответствующих окон.

Группа команд `Desktops` позволяет сохранить текущую конфигурацию среды разработки (команда `Save Desktop...`,) – т.е. информацию об открытых окнах и их расположении. Отметим, что с помощью команды `View | Desktops | Set Debug Desktop` () можно создать специальную конфигурацию *Debug Desktop*, загружаемую в режиме отладки.

Группа команд `Toolbars` используется для настройки панели инструментов.

Команда Toggle Form/Unit ( , F12) используется для переключения между окнами редакторов кода и форм.

Для вызова окна, содержащего список всех модулей проекта, используется команда Units ( , Ctrl+F12). Аналогичное действие для списка всех форм проекта реализует команда Forms (, Shift+F12).

4.2.6. Меню Project

Команды данного меню предназначены для создания и редактирования проектов. Например, команда Add to Project ( , Shift+F11) используется для добавления к проекту файла модуля, ресурсов и т.д. Обратная команда Remove from Project () удаляет из проекта ненужные модули.

Команда Import Type Library () открывает одноименное окно диалога (рис. 4.3), в котором отображаются библиотеки типов, зарегистрированные в системе. Доступны следующие элементы управления:

- кнопка Add используется для добавления новой библиотеки типов и регистрации ее в системе;
- кнопка Remove удаляет библиотеку типов из списка и регистрационную запись в системном реестре;
- поле ввода Class names отображает имена импортируемых классов, которые можно изменять;
- список Palette page определяет страницу *палитры компонентов*, на которой будут находиться компоненты, соответствующие классам, указанным в поле Class names;
- поля ввода Unit dir name и Search path задают пути к папке модуля, использующего текущую библиотеку типов, и к библиотечным модулям, используемым при создании пакета;
- кнопка Install добавляет новые компоненты, соответствующие классам поля Class names, как в существующий пакет, так и во вновь создаваемый;
- кнопка Create Unit создает и добавляет к текущему проекту модуль с описанием импортируемых классов.

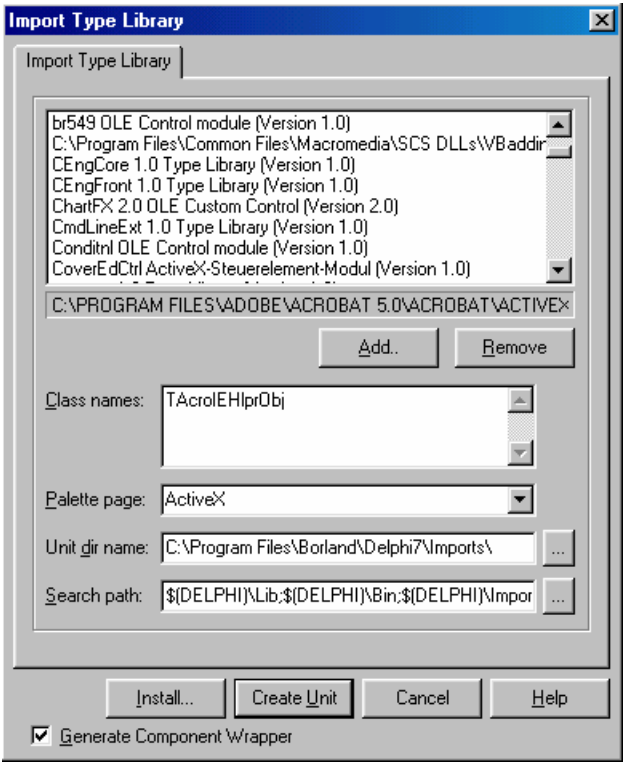


Рис. 4.3. Окно диалога Import Type Library

Продолжим рассмотрение команд меню `Project`.

Команда `Add to Repository` () добавляет текущий проект в *хранилище объектов* в качестве шаблона. `Add New Project` () добавляет в текущую группу проектов новый проект, а команда `Add Existing Project` () добавляет существующий проект в текущую группу проектов.

С помощью команды `View Source` () можно открыть в *редакторе кода* главный файл проекта, который начинается со служебного слова `program`.

Компиляция текущего проекта осуществляется командой `Compile <имя проекта>` ( , `Ctrl+F9`), компиляция всех про-

ектов и библиотек пакета реализует команда Compile All Projects (). Команда Build <имя проекта> () осуществляет перекомпиляцию проекта вместе со всеми используемыми модулями. Соответственно, Build All Projects () выполняет аналогичные действия для всех проектов пакета.

После компиляции командой Information () можно получить информацию о текущем проекте. В одноименном окне отображается следующая информация: количество строк программного кода (Source compiled); размер исполняемого файла или динамической библиотеки без отладочной информации (Code size); объем памяти, занятый глобальными переменными (Data size); объем памяти, отведенный для локальных переменных (Initial stack size); размер выходного файла (File size); список всех используемых в проекте пакетов (Package Used).

Последняя команда Options (, Shift+Ctrl+F11) вызывает окно диалога свойств (опций) проекта. Закладки этого окна (рис. 4.4) имеют следующее назначение.

- Forms – работа с формами проекта: назначение главной формы (Main form), автоматически создаваемых форм (Auto-create forms) и остальных форм (Available forms).
- Application – определяет для откомпилированного приложения следующие атрибуты: Title – надпись, идентифицирующая программу при минимизации окна приложения; Help file – файл справки, автоматически ассоциированный с разрабатываемым программным обеспечением; Icon – *иконка* или *пиктограмма* приложения.
- Linker – параметры компоновщика программного кода.
- Directories/Conditionals – пути к входным и выходным файлам текущего проекта, файлам библиотек сторонних производителей и т.п.
- Compiler Messages – сообщения компилятора, флаг Show Hints разрешает генерацию замечаний компилятора (*hint*), например, об объявленной, но неиспользованной пере-

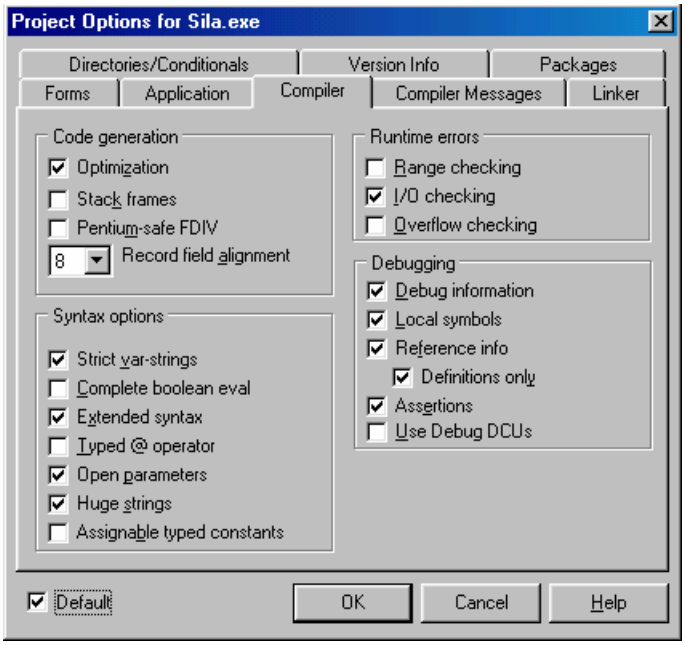


Рис. 4.4. Окно для задания параметров компиляции

менной; флаг `Show Warnings` вызывает генерацию предупреждений компилятора (*warning*).

Наиболее важной из вкладок свойств проекта является `Compiler` – настройка компилятора (рис. 4.4), которая позволяет установить ряд *директив* компилятора.

Директивы компиляции кода (`Code generation`):

- `Optimizations {$O}` – компиляция с оптимизацией, иногда сокращает размер выполняемого файла и повышает эффективность программы, но затрудняет отладку по причине устранения из кода некоторых операторов и переменных;
- `Stack frames {$W}` – компилятор генерирует стек для всех процедур и функций;
- `Pentium-Safe FDIV {$U}` – генерация кода для проверки ошибок деления, свойственных ранним версиям Pentium;

➢ `Record field alignment {$A}` – выравнивание элементов структур данных по границам заданного числа битов (1 (соответствует выключенной директиве `{$A}`), 2, 4, 8 (соответствует включенной директиве `{$A}`)), данная директива позволяет повысить скорость работы со структурированными данными за счет увеличения памяти, занимаемой ими физически.

Директивы ошибок времени выполнения (`Runtime errors`). Включение опций этой группы незначительно увеличивает размер выходного файла, но они необходимы при отладке программы. Доступны следующие директивы:

➢ `Range Checking {$R}` – проверка допустимых значений индексов массивов и строк;

➢ `I/O Checking {$I}` – проверка ошибок ввода/вывода после соответствующей операции;

➢ `Overflow Checking {$Q}` – проверка переполнения при целочисленных операциях.

Директивы синтаксиса (`Syntax options`):

➢ `Strict Var-Strings {$V}` – проверка параметров типа строк (не действует при директиве `Open parameters {$P}`);

➢ `Complete Boolean Eval {$B}` – полное вычисление булевских выражений, за исключением очень редких специальных случаев включать эту опцию не рекомендуется;

➢ `Extended Syntax {$X}` – позволяет использовать вызов функций как процедур (с игнорированием возвращаемого результата), поддерживает тип `PChar`;

➢ `Type @ Operator {$T}` – проверка типа оператора, возвращаемого операцией `@`;

➢ `Open parameters {$P}` – разрешение передачи параметров для процедур и функций в виде открытых строк (тип **`string`** эквивалентен `OpenString`), которые обычно более надежны и эффективны;

➢ `Huge Strings {$H}` – при включении этой опции тип **`string`** эквивалентен `AnsiString`, при выключенной – короткой строке `ShortString`;

➢ Assignable Typed Constants {$J} – разрешения операций присваивания для типизированных констант.

Директивы отладки (Debugging):

➢ Debug Information {$D} – размещение отладочной информации в объектных файлах модулей *.*dcu*;

➢ Local Symbols {$L} – генерация информации о локальных символах;

➢ Reference Info/Definitions Only {$Y}/{$YD} – генерация информации о ссылках на объявленные идентификаторы, необходимой для работы *редактора кода* (данные директивы не нужны при установке двух предыдущих опций);

➢ Assertions {$C} – включает в код программы директивы проверки утверждений (при отключении для удаления добавленного кода необходима перекомпиляция);

➢ Use Debug DCUs – разрешает использовать отладочную версию компонентов VCL.

4.2.7. Меню Run

Команды этого меню предназначены для запуска и отладки проектируемых программных средств.

Важнейшей является команда Run ( , F9), запускающая компиляцию (по необходимости) и выполнение текущего проекта. Команда продублирована на инструментальной панели.

Команда Attach to Process () позволяет выбрать из списка всех работающих на данный момент программ процесс для режима отладки в среде Delphi. Текст выбранной программы отображается в отладчике в командах ассемблера.

Для запуска программы с параметрами предназначена команда Parameters ().

Для пошагового выполнения программы служат команды Step Over ( , F8) и Trace Into ( , F7). Последняя называется *отладка с заходом* в вызываемые процедуры и функции. Команда Trace to Next Source Line ( , Shift+F7) обеспечивает пошаговую отладку подпрограмм косвенного вызова.

`Run to Cursor` ( , `F4`) обеспечивает запуск ПО и выполнение до той строки программного кода, на которой расположен текстовой курсор.

Команда `Run Until Return` ( , `Shift+F8`) выполняет отлаживаемое приложение до возврата из выполняемой функции или процедуры. Остановка произойдет на операторе, следующем за вызовом.

Команда `Show Execution Point` () открывает окно *редактора кода* и показывает строку программы, выполняемую в данный момент. Команда доступна только в режиме пошаговой отладки.

Остановить выполнение запущенной программы без завершения можно с помощью команды `Program Pause` ().

Команда `Program Reset` ( , `Ctrl+F2`) завершает работу проектируемого приложения.

Команды `Inspect` (), `Evaluate/Modify` ( , `Ctrl+F7`) и `Add Watch` ( , `Ctrl+F5`) используются для просмотра и изменения значений переменных в режиме отладки.

Группа команд `Add Breakpoint` предназначена для установки () и снятия точек останова в программе, а также для изменения их параметров. Отметим, что щелчок мышью в *редакторе кода* на поле, расположенном слева от текста программы, также позволяет установить (снять) точку останова.

4.2.8. Меню Component и палитра компонентов

Команды меню `Component` используются для создания (`New Component` – ) и установки новых компонентов (`Install Component` – ) или элементов `ActiveX` (`Import ActiveX Control` – ). Команда `Create Component Template` становится доступной при выделении одного или нескольких компонентов. Она позволяет создать шаблон компонента или группы и включить его в библиотеку.

Команда `Install Packages` () позволяет просмотреть список всех имеющихся пакетов и указать, какие из них долж-

ны быть подкомпилированы в код приложения. Также можно увидеть текущий список компонентов каждого пакета и отредактировать состав пакетов.

Команда `Configure Palette` открывает окно диалога `Palette Properties` (рис. 4.5), с помощью которого производится настройка палитры, используемой для отображения компонентов, содержащихся в библиотеке Delphi.

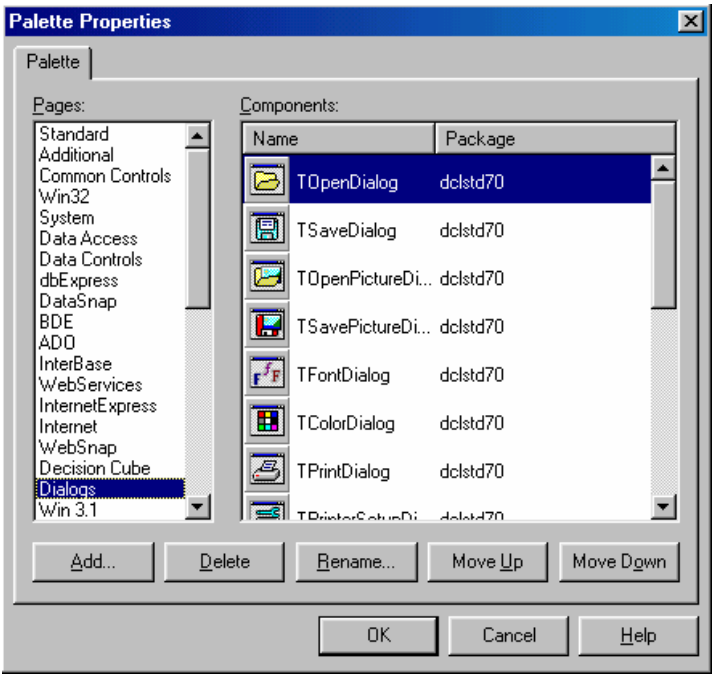


Рис. 4.5. Окно настройки палитры компонентов

В соответствии с выполняемыми функциями все расположенные в палитре компоненты разделены на группы, каждая из которых размещается на отдельной странице. Палитра компонентов полностью конфигурируется пользователем. Можно создавать (`Add`) или удалять (`Delete`) новые страницы, переименовывать (`Rename`) и менять (`Move Up`, `Move Down`) их расположение. Подобные действия применимы и к отдельным

компонентам, причем удаление компонента из палитры (кнопка Hide, заменяющая кнопку Delete при выборе конкретного компонента) не приводит к его физическому удалению из библиотеки компонентов Delphi.

В стандартной конфигурации палитры, например, содержатся следующие страницы компонентов и элементов управления (рассмотрим только основные из них):

- Standard – стандартные элементы управления оконного интерфейса Windows;
- Additional – специализированные элементы управления интерфейса Windows;
- Win32 – элементы интерфейса, содержащиеся в 32-битных системных библиотеках Windows 95 и Win32s;
- System – специализированные системные элементы управления;
- Dialogs – стандартные диалоги открытия файла, сохранения, печати и т.п.;
- Samples – примеры разработанных компонентов;
- Servers – компоненты для организации взаимодействия с приложениями Microsoft Office;
- Data Access, Data Controls, dbExpress, BDE, ADO, InterBase – компоненты для работы с базами данных;
- QReport – компоненты для подготовки отчетов.

Напомним, что для получения справки по интересующему компоненту необходимо выбрать его в палитре и нажать F1.

4.2.9. Меню Database, Tools, Windows, Help

Команды меню Database предназначены для работы с базами данных. Команда Explore запускает утилиту *SQL Explorer*, позволяющую просматривать и редактировать существующие базы данных. Команда SQL Monitor также запускает одноименную утилиту, которая позволяет отслеживать взаимодействие между процессором баз данных *BDE* (*Borland Database Engine*) и клиентской базой данных. Команда Form Wiz-

ard запускает мастер (*Wizard*) создания форм, на которых проектируется отображение данных локальной или удаленной БД.

Меню Tools содержит команды, вызывающие диалоговые окна настроек интегрированной среды Delphi (Environment Options – ), редактора кода (Editor Option – ), отладчика (Debugger Options – ) и хранилища объектов (Repository – ). Инструментарий наиболее важных диалоговых окон будет рассмотрен далее.

Кроме того, в данном меню содержатся команды, позволяющие запускать некоторые внешние программы (утилиты). Среди них отладчик Web-приложений (Web App Debugger – ), программа для проектирования, просмотра и редактирования таблиц баз данных (Database Desktop), графический редактор (Image Editor) и другие. Естественно, существует возможность настройки списка вызываемых программ – команда Configure Tool ().

Меню Windows содержит список открытых окон. Выбор окна осуществляется командой Next Window ( , Alt+End).

Разделы меню Help позволяют работать со справочной системой Delphi.

4.3. Инспектор объектов

Важнейшим инструментом в интегрированной среде проектирования Delphi (см. рис. 4.1) является *инспектор объектов* (Object Inspector). Его используют для настройки опубликованных (секции **public** и **published** при объявлении классов) свойств компонентов.

Окно инспектора объектов содержит выпадающий список и две вкладки. На первой (Properties) отображается список свойств выделенного объекта, на второй (Events) – список событий, на которые реагирует объект. Выпадающий список содержит перечень всех компонентов, размещенных в актив-

ном на данный момент *контейнере компонентов* (форме, фрейме, модуле данных).

Каждая вкладка разделена на две колонки. В левой перечислены имена свойств или событий, в правой – их значения, которые можно редактировать. Редактирование осуществляется или непосредственным заданием значения свойства, или выбором из предопределенного списка, или с помощью специального редактора.

Настройка инспектора объектов выполняется с помощью команд контекстного меню. Наиболее используемыми являются следующие команды.

- Команда `View` определяет категории отображаемых свойств, т.е. свойства всех компонентов разделены на ряд категорий (например, `Visual` – свойства, связанные с положением, цветом и т.п. компонента, или `Border Style` – стиль внешнего интерфейса), отображением которых можно управлять.
- Команда `Arrange` определяет способ сортировки отображаемых свойств – по алфавиту (`by Name`) или по категориям (`by Category`).
- Команда `Stay on Top` располагает окно инспектора объектов поверх всех остальных окон.
- `Hide` закрывает окно инспектора объектов.
- `Help`, соответственно, вызывает файл справки.
- Команда `Properties` вызывает диалоговое окно для редактирования и настройки свойств инспектора объектов.

4.4. Редактор кода и его настройка

Редактор программного кода также является важнейшим инструментом среды Delphi. Настройка его осуществляется выполнением команды `Editor Option` () меню `Tools`. Окно настройки разделено на несколько страниц (вкладок).

Страница `General` показана на рис. 4.6. Она позволяет установить следующие опции редактора (`Editor Options`):

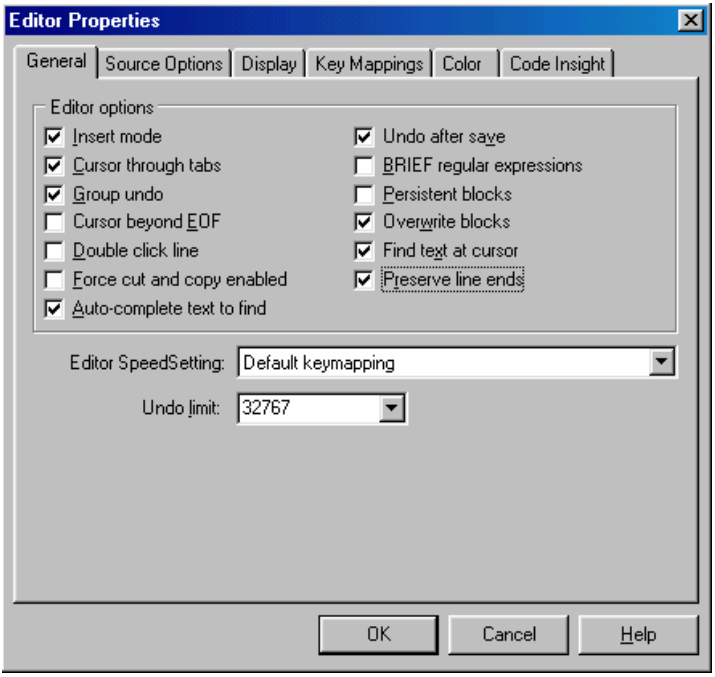


Рис. 4.6. Страница `General` окна настройки редактора кода

➢ `Insert Mode` – установка по умолчанию режима вставки, а не замены символа (изменяется пользователем с помощью клавиши "`Insert`");

➢ `Cursor Through Tabs` – клавиши со стрелками перемещают курсор на следующую позицию табуляции;

➢ `Group Undo` – при нажатии клавиш `Alt+Backspace` или выполнении команды `Edit | Undo` (↶, `Ctrl+Z`) восстанавливается состояние, которое было до последней последовательности команд одного типа;

➢ `Cursor Beyond EOF` – курсор может позиционироваться после символа конца файла;

➢ `Double Click Line` – при двойном клике на какомлибо символе выделяется вся строка, иначе выделяется слово;

➢ `Force Cut And Copy Enabled` – команды `Cut` ( , `Ctrl+X`) и `Copy` ( , `Ctrl+C`) меню `Edit` выполняются даже при невыделенном фрагменте текста;

➢ `Auto-complete text to find` – невозможно автозавершение диалога поиска;

➢ `Undo after save` – позволяет восстанавливать изменения после команды сохранения;

➢ `BRIEF regular expressions` – использование регулярных выражений редактора `BRIEF`;

➢ `Persistent blocks` – сохранение выделения блока до нового выделения при сдвиге курсора;

➢ `Overwrite blocks` – замещение выделенного блока очередным нажатым символом;

➢ `Find text at cursor` – текст, на котором стоит курсор, помещается при выполнении команд поиска меню `Search` в окно задания текста;

➢ `Preserve line ends` – сохранять символы конца строки.

Все эти опции можно устанавливать независимо друг от друга. Но можно осуществить и быструю смену стиля редактора, выбрав один из предопределенных стилей в выпадающем списке `Editor SpeedSetting`.

Объем текста, который может быть восстановлен, определяется параметром `Undo Limit`.

Страница `Display` окна настроек показана на рис. 4.7.

Группа опций `Display and file options` позволяет определить следующие настройки:

➢ `BRIEF cursor shapes` – установить курсор в стиле редактора BRIEF;

➢ `Create backup file` – при сохранении редактируемых файлов сохранять резервную копию (расширение резервных файлов начинается с символа "~");

➢ `Zoom to full screen` – при развертывании окна редактора кода устанавливать его размер на весь экран, заслоняя

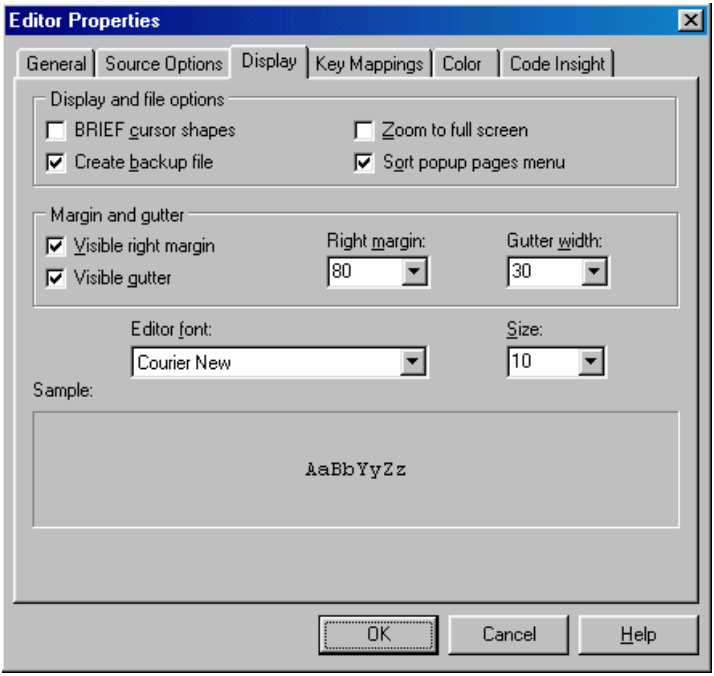


Рис. 4.7. Страница `Display` окна настройки редактора кода

таким образом полосу главного меню, палитру компонентов и инструментальные панели;

- `Sort popup pages menu` – определяет, каким образом (по алфавиту или по времени создания закладок) упорядочить список открытых в редакторе кода документов, на которые можно переключиться посредством раздела `Pages` контекстного меню закладок окна редактора кода.

Группа опций отображения текста `Margin and gutter` определяет следующие параметры:

- `Visible right margin`– делает видимой линию правого поля в окне редактора кода;
- `Visible gutter` – делает видимой полосу установок точек прерывания слева от текста программного кода;

- Right margin – длина строки в окне редактора кода, по умолчанию – 80 символов, максимально – до 1024;
- Gutter width – ширина полосы для установки точек прерывания, по умолчанию – 30;
- Editor Font – шрифт, используемый в редакторе кода (рекомендуется использовать шрифт постоянной ширины типа Courier, также актуально проследить за тем, чтобы используемый шрифт поддерживал символы кириллицы);
- Size – размер используемого шрифта;
- Sample – пример текста, отображенного выбранным шрифтом и размером.

Страница Key Mappings позволяет установить комбинации управляющих клавиш редактора кода, которые имеют ряд предопределенных значений, соответствующих стилям, устанавливаемым на странице General.

На странице Color (рис. 4.8) можно определить виды визуального выделения различных синтаксических элементов (выбираются в списке Element) программного кода. Аналогично предыдущим страницам имеется возможность быстрого выбора настройки из предопределенных цветовых схем (список Color SpeedSetting). Для элементов также имеется возможность выбора атрибутов шрифта и цвета фона (Background Color). Результаты изменений можно увидеть в имитации окна редактора кода.

4.5. Общие настройки среды проектирования

Многостраничное окно общих настроек среды проектирования Delphi вызывается командой Environment Options () меню Tools.

На рис. 4.9 показана вкладка Preferences.

Опции раздела автосохранения (Autosave options) определяют автоматическое сохранение файлов проекта при ка-

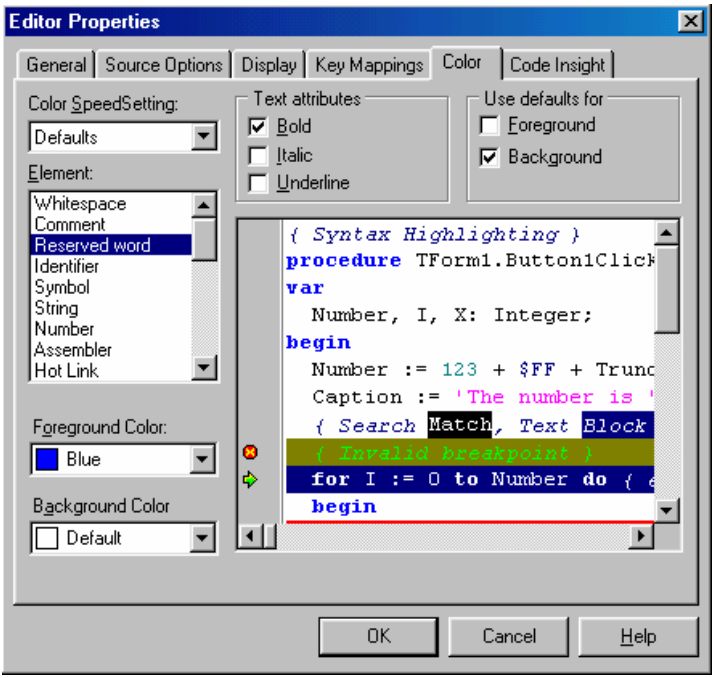


Рис. 4.8. Страница `Color` окна настройки редактора кода

ждом запуске приложения (`Editor files`) и сохранение информации об экране (`Project Desktop`) при выходе из Delphi.

Уточнение понятия информации об экране определяются другой группой опций – `Desktop contents`. В частности, флаг `Desktop Only` задает сохранение информации о каталогах и файлах, открытых в редакторе кода, и об открытых окнах. Если необходимо сохранить также информацию о символах последней успешной компиляции, то необходимо задать опцию `Desktop And Symbols`.

Флаг `Auto drag docking` разрешает или запрещает автоматическое встраивание окон друг в друга. В процессе проектирования существует возможность временно изменить значение этой опции на противоположное с помощью нажатой и удерживаемой клавиши `Ctrl`.

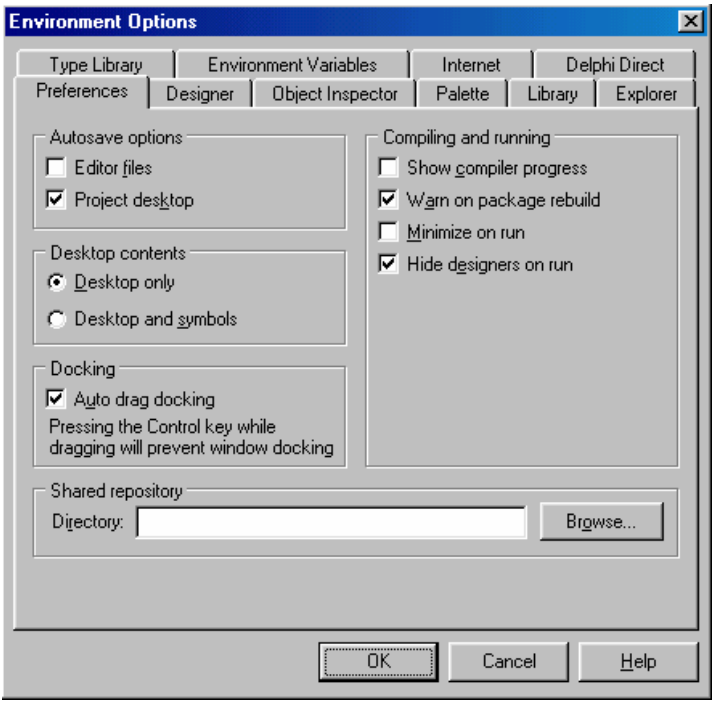


Рис. 4.9. Страница `Preferences` окна общих настроек среды проектирования Delphi

Группа опций компиляции и выполнения (`Compiling and running`) определяет следующие параметры:

- `Show Compiler Progress` – показывать ход выполнения компиляции;
- `Warn on package rebuild` – показывать замечания при построении пакетов во время компиляции;
- `Minimize on run` – сворачивать или *минимизировать* Delphi при выполнении приложения (по завершении работы программы окна Delphi восстанавливаются);
- `Hide designers on run` – делать невидимыми окна проектирования (инспектора объектов и визуального редактора форм) при выполнении приложения.

Поле ввода Directory в нижней части страницы Preferences окна общих настроек среды проектирования определяет местонахождение файла депозитария delphi32.dro (по умолчанию расположен в директории BIN.)

Страница Designer задает опции проектирования:

- Display grid – делает видимыми узлы сетки;
- Snap to grid – автоматически привязывает компоненты, помещенные на форму, к узлам сетки;
- Grid size x – шаг сетки по вертикали (от 2 до 128);
- Grid size y – шаг сетки по горизонтали (от 2 до 128);
- New Form as Text – определяет вид сохранения файла описания *.dfm атрибутов формы: текстовой (флаг установлен) или двоичный (бинарный);
- Auto create forms & data modules – определяет, будут ли новые формы проекта (кроме первой) рассматриваться как автоматически создаваемые (Auto Create) или как возможные (Available Forms);
- Show component captions – делает видимыми надписи компонентов;
- Show designer hints – делает видимыми ярлычки с именами классов компонентов;
- Show extended control hints – делает видимыми ярлычки с расширенной информацией: положением, размером, значением свойства TabStop и номером в последовательности табуляции.

На странице Library находятся списки каталогов, в которых ищутся используемые в проекте файлы:

- Library path – пути поиска исходных файлов проекта;
- BPL output directory – каталог для размещения откомпилированных файлов проектов **.bpl*;
- DCP output directory – каталоги файлов пакетов компонентов Delphi **.dcp*;

➢ `Browsing path` – каталог, в котором браузер кода `Code Browser` ищет файлы, содержащие информацию об идентификаторах, например, компонентах VCL.

Ряд важных настроек редактора программного кода и браузера объектов находятся на странице `Explorer` (рис. 4.10).

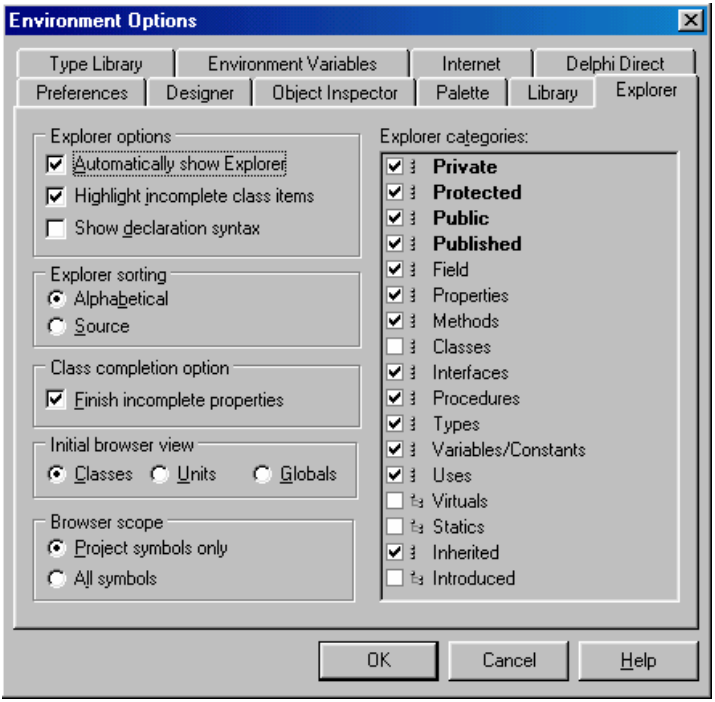


Рис. 4.10. Страница `Explorer` окна общих настроек среды проектирования Delphi

Автоматическое появление окна *исследователя кода*, встроенного в редактор кода, определяется опцией `Automatically show Explorer`. Флаг `Highlight incomplete class items` определяет появление выделения жирным шрифтом незавершенных свойств и методов. Установка опции `Show declaration syntax` приведет к отображению синтаксиса объявлений (по умолчанию отображаются только имена).

Раздел Explorer sorting определяет последовательность визуализации информации: по алфавиту (Alphabetical) или в порядке последовательности объявлений (Source).

Опция Finish incomplete properties позволяет автоматически включать в программный код шаблон реализации незавершенного свойства при нажатии клавиш Ctrl+Shift+C.

Раздел Explorer categories определяет классификацию отображаемых элементов дерева *исследователя кода*.

Группа переключателей (радиокнопок) Initial browser view определяет страницу информации, открываемой в окне браузера объектов. Радиокнопки группы Browser scope уточняют информацию о визуализируемых символах: Project symbols only – только о символах модулей текущего проекта, All symbols – о символах всех модулей, явно или неявно используемых в проекте, включая модули библиотеки VCL.

4.6. Некоторые дополнительные настройки

Завершая данную главу, приведем некоторые дополнительные настройки, облегчающие разработку программных средств в интегрированной среде Delphi.

Для конфигурации *средств подсказок и помощи* Code Insight необходимо перейти на одноименную страницу в окне настроек редактора кода (Editor Option) меню Tools. Доступны следующие функции:

- Code completion – подсказка в виде списка свойств, методов, событий, относящихся к данному компоненту (при отключении этой опции функция остается доступной при нажатии клавиш Ctrl+пробел);
- Code parameters – подсказка параметров функций, процедур, методов;
- Tooltip expression evaluation – оценка выражений во время останова программы или пошаговой отладки;

➢ Tooltip symbol insight – подсказка определений идентификаторов, над которыми перемещается курсор мыши;

➢ ползунок Delay устанавливает время задержки автоматического срабатывания средств подсказок и помощи.

Можно воспользоваться шаблонами типичных структур языка Object Pascal и определять собственные шаблоны.

Для использования готового шаблона необходимо в окне редактора кода нажать сочетание клавиш Ctrl+J и из появившегося списка шаблонов выбрать требуемый. Например, выбор стандартного шаблона fors добавит в окно редактора кода следующую заготовку – **for** := **to do**.

Для определения собственного шаблона (или редактирования стандартного) необходимо из меню Tools вызвать Editor Options, перейти на вкладку Source Options и нажать кнопку Edit Code Templates. Появляющееся после этих действий окно диалога показано на рис. 4.11. В нем для примера задана структура оператора **for** с уменьшающимся значением переменной цикла. В текст шаблона можно вставить вертикальную черту в том месте, где необходимо установить курсор при вводе данного шаблона в текст.

Настройка отладчика осуществляется командой Debugger Options () меню Tools. Самая важная опция Integrated debugging, обеспечивающая активацию отладчика, расположена внизу и видна на любой странице. Отметим наиболее востребованные опции этого инструмента.

➢ Allow function calls in new watches, страница General – разрешает вызов функций в выражениях, отображаемых в окне наблюдения.

➢ Stop on Delphi Exceptions, страница Language Exceptions – выключение этой опции отменяет появление дополнительных окон сообщений об исключительных ситуациях, то есть в процессе отладки проектируемая программа ведет себя также, как и при обычном запуске.

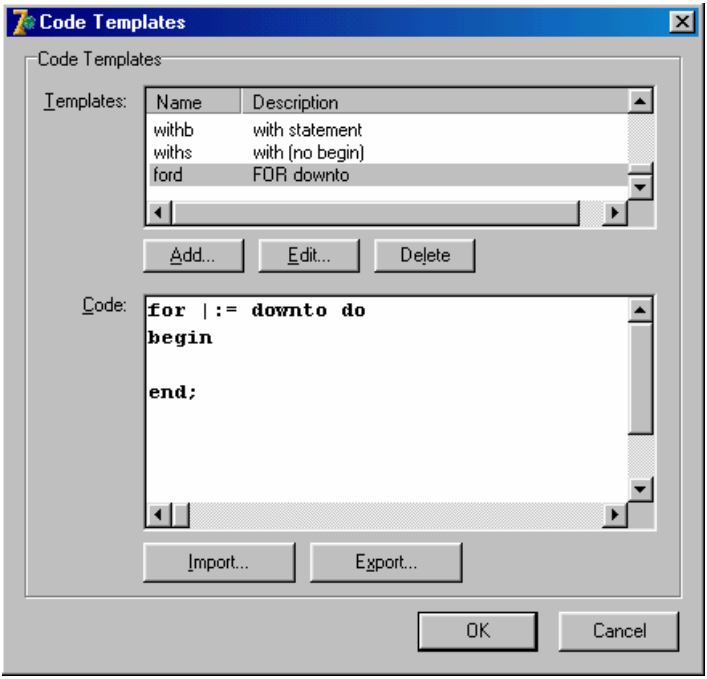


Рис. 4.11. Диалоговое окно ввода нового шаблона типичной структуры языка Object Pascal

Отметим, что отключение флага `Stop on Delphi Exceptions` отключает при отладке установку *курсора выполняемой строки* на фрагменте кода, сгенерировавшего ошибку. Таким образом, отладка приложения затрудняется.

5. ОСНОВНЫЕ ЭЛЕМЕНТЫ ПОСТРОЕНИЯ ИНТЕРАКТИВНОГО ИНТЕРФЕЙСА ПРИКЛАДНЫХ ПРОГРАММ

5.1. Формы и фреймы – основа визуализации интерфейсных элементов

Форма представляет собой окно приложения на этапе разработки. Как правило, прикладная программа или приложение, работающее в операционной среде Windows, имеет, по крайней мере, одно окно. Поэтому проект Delphi также должен содержать хотя бы одну форму. Формы обеспечивают визуальное проектирование интерфейса на этапе разработки и, при работе приложения, инициализацию и визуализацию разработанного интерфейса. Происходит это благодаря тому, что форма является своего рода *контейнером*, содержащим в себе другие элементы интерфейса.

Фрейм, как и форма, может содержать различные компоненты, но его основное назначение в том, чтобы обеспечить возможность разделения формы на несколько самостоятельных участков, облегчая тем самым процесс проектирования.

Хотя фрейм и форма входят в состав VCL и являются стандартными компонентами Delphi, но в палитре компонентов их нет. Для создания новой формы (или фрейма) используют команды меню `File` – `Form` () и `Frame` (). Как правило, этим элементам соответствуют отдельные модули. После создания фрейма с помощью команды из палитры компонентов `Frames` ( , страница `Standard`) его можно разместить на форме.

При создании нового приложения всегда автоматически создается форма, соответствующая главному окну приложения. Если в ПО присутствуют другие (*вторичные*) формы, то можно оставить предлагаемый Delphi по умолчанию механизм автоматического создания форм, а можно и выполнять эти

действия программно, т.е. *динамически* создавать и уничтожать вторичные формы.

5.2. Наиболее общие свойства, методы и события компонентов

Все компоненты Delphi составляют определенную иерархию, т.е. являются потомками базовых, наиболее общих классов. Благодаря этому они обладают подобными по имени и по назначению *свойствами*, *методами* и *событиями*.

Свойства, соответственно, характеризуют конкретные параметры компонента как объекта проектирования. По типу значений они могут быть отнесены к *числовым*, *ранжируемым* и *атрибутивным*.

Числовые свойства принимают значения из некоторой числовой области. Для них правильной является постановка вопроса: "Во сколько раз (насколько) одно значение данного свойства больше (меньше) другого значения этого свойства?" Также различные объекты могут сравниваться между собой по значениям одноименного свойства.

Для *ранжируемого* свойства на множестве его значений можно установить, какое из них предпочтительнее, но в то же время нельзя сказать, во сколько раз (насколько) одно значение больше (меньше) другого. По существу, ранжируемые значения, даже если они выражаются числами, есть не что иное, как некоторый способ кодирования нечисловых свойств.

Для *атрибутивного* свойства на множестве его значений нельзя устанавливать отношение предпочтения. Например, атрибутивное свойство – цвет, значения – красный, синий, зеленый. Для любых двух значений атрибутивного свойства, даже если они выражены числами, нельзя говорить о том, что одно из значений предпочтительнее другого (во всяком случае, без глубокого специального рассмотрения), и тем более нельзя говорить о том, на сколько (во сколько раз) одно значение больше (меньше) другого. Иногда из атрибутивных свойств выде-

ляют подкласс *бинарных* свойств, которые могут принимать лишь два значения: есть в наличии данное свойство или нет.

Важнейшими свойствами компонентов Delphi являются:

- `Align` – определяет способ выравнивания компонента внутри *родительского компонента* (контейнера), т.е. определяет, остается ли компонент неизменным при изменении размеров контейнера (`alNone`) или он изменяется, занимая всю доступную площадь (`alClient`), ее верхнюю (`alTop`), нижнюю (`alBottom`), левую (`alLeft`) или правую (`alRight`) часть;
- `Anchors` – определяет привязку данного компонента к родительскому при изменении размеров последнего: к верхней (`akTop`), к нижней (`akBottom`), к левой (`akLeft`) и к правой (`akRight`) границе контейнера. Например, если в множестве `Anchors` компонента `A1` присутствуют привязки к противоположным сторонам контейнера, то при изменении родительского компонента происходит растяжение или сжатие (вплоть до полного уничтожения изображения компонента) `A1`, причем выдерживаются расстояния от сторон контейнера;
- `AutoSize` – определяет, будет ли высота компонента автоматически адаптироваться к размеру его содержимого;
- свойство `Caption` связывает с компонентом некоторую строку текста, поясняющую его назначение, чаще всего это надписи на кнопках, метках, тексты разделов меню и т.д.;
- `Cursor` определяет изображение курсора мыши в области компонента: `crAppStart` (), `crArrow` (), `crCross` (), `crDefault` (курсор, принятый по умолчанию), `crDrag` (), `crHandPoint` (), `crHelp` (), `crHourGlass` (), `crHSplit` (), `crIBeam` (), `crMultiDrag` (), `crNo` (), `crNoDrop` (), `crSizeAll` (), `crSizeNESW` (), `crSizeNS` (), `crSizeNWSE` (), `crSizeWE` (), `crSQLWait` (), `crUpArrow` (), `crVSplit` (), `crNone` (изображение отсутствует), также возможно сформировать собственное изображение курсора;

- свойство `DragCursor` отвечает за изображение курсора (принимаемые значения аналогичны предыдущему свойству) при перемещении его в области компонента в процессе перетаскивания мышью;
- свойство `Enabled` определяет, реагирует ли компонент на события, связанные с мышью, клавиатурой и таймером, т.е. доступность компонента для пользователя (недоступный компонент отображается серым цветом);
- атрибуты используемого компонентом шрифта определяются свойством `Font`, которое, в свою очередь, также является стандартным классом Delphi и имеет собственные свойства, такие, как `Color` (цвет), `Name` (имя шрифта), `Size` (размер), `Style` (стиль) и др.;
- `Height` – определяет высоту компонента или формы в пикселях;
- `HelpContext` – номер *контекстно-зависимой встроенной справки* по элементу интерфейса, для использования этого свойства разработчику необходимо спроектировать файл справки (обычно или в *hlp*- или в *chm*-форматах);
- свойство `Hint` обеспечивает текст подсказки, появляющийся во всплывающем окне подсказки при задержке курсора над компонентом или в заданном месте окна, например, в строке состояния;
- свойство `Left` определяет координату левого края в пикселях, для компонентов за начало отсчета берется левая граница клиентской области родителя;
- `Name` – имя компонента;
- `Tag` – это свойство системой не используется и равно нулю по умолчанию, разработчик ПО может определить и использовать его по своему усмотрению, помещая в него любую необходимую информацию, совместимую с типом `integer`;
- `Top` – координата верхнего края компонента;
- свойство `Transparent` определяет прозрачность фона компонента;

- `Visible` – определяет видимость компонента;
- `Width` – горизонтальный размер компонента.

Методы, во-первых, используются для чтения или записи свойств классов Delphi. Во-вторых, любой класс имеет два специальных метода:

- **`Constructor`** – конструктор, обычно имеющий имя `Create`, предназначен для выделения памяти под объект и начальной инициализации свойств;
- **`Destructor`** – деструктор, как правило, идентифицируемый именем `Destroy`, по назначению является обратным конструктору и служит для уничтожения объекта и освобождения занимаемой им памяти.

Из других методов, типичных для различных компонентов, стоит выделить:

- `Clear` – очистка полей объекта, списков, удаление текстовых данных;
- копирование данных из одного объекта в другой реализуется методом `Assign`, иногда для этих целей используют методы, начинающиеся с `Copy<…>`;
- методы чтения и записи обычно называются `SaveTo<устройство>` и `LoadFrom<устройство>`, например, метод чтения из файла идентифицируется как `LoadFromFile` (заметим, что в некоторых случаях для аналогичных процедур используются методы `Open` и `Close`);
- с помощью метода `Add` обычно создают новый объект (например, пункт меню, элемент списка, новую текстовую строку) и добавляют его в список объектов;
- удаление элементов, соответственно, обычно реализовано методом `Delete`;
- при необходимости полной перерисовке визуального компонента используются методы `Invalidate` (сообщает Windows о необходимости перерисовки после того, как будут об-

работаны другие важные сообщения) или `Repaint` (немедленная перерисовка изображения);

- для предотвращения мерцания экрана, т.е. при необходимости временного отключения (включения) перерисовки элементов при изменении одного или нескольких визуальных компонентов используются методы `EndUpdate` (`BeginUpdate`);
- метод `BringToFront` перемещает компонент в начало Z-последовательности;
- метод `ClassName` возвращает имя типа объекта;
- `Hide` – делает компонент невидимым, метод `Show` осуществляет обратную операцию;
- `SetFocus` – передает компоненту фокус.

События являются основой событийного программирования и предназначены для формирования реакции компонента на действия пользователя, операционной системы, других прикладных программ. Как правило, основным программным кодом в Delphi является именно код *обработчиков событий*, в которых напрямую или с помощью методов изменяются свойства компонентов проектируемого ПО. Идентификаторы событий обычно начинаются с `On<…>`. Рассмотрим основные события, свойственные компонентам Delphi:

- `OnChange` – событие наступает после изменения основных смысловых данных компонента, например, текста;
- `OnClick` – событие соответствует щелчку мыши на компоненте и некоторым другим действиям пользователя;
- `OnClose` – событие соответствует закрытию, например, окна диалога или формы;
- `OnContextPopup` – событие наступает при щелчке правой кнопкой мыши на компоненте или при любом другом способе вызова контекстного всплывающего меню, обработчик события может использоваться для изменения меню, в частности, в зависимости от позиции курсора мыши;
- `OnDblClick` – двойной щелчок мыши;

- `OnEnter` – событие наступает в момент активизации элемента (*получения фокуса*), например, при выборе его с помощью мыши;
- событие `OnExit` наступает в момент потери элементом фокуса;
- событие `OnKeyDown` наступает при нажатии пользователем любой клавиши, включая функциональные и вспомогательные, такие, как `Shift`, `Alt` и `Ctrl`;
- при нажатии пользователем символьных клавиш происходит событие `OnKeyPress`;
- событие `OnKeyUp` наступает при отпускании пользователем любой клавиши;
- обработка событий `OnMouseDown` и `OnMouseUp` используется для операций, требуемых при нажатии и отпускании пользователем какой-нибудь кнопки мыши;
- при перемещении курсора мыши над компонентом генерируется событие `OnMouseMove`;
- событие `OnPaint` наступает, когда приходит сообщение Windows о необходимости перерисовать испорченное изображение (может испортиться, например, из-за временного перекрытия изображения другим окном) визуального элемента интерфейса прикладной программы;
- `OnResize` – наступает сразу после изменения размера компонента или формы;
- `OnShow` – событие, наступающее при открывании диалогового окна или формы.

Конечно, в данном разделе были рассмотрены не все свойства, методы и события компонентов. Их полное и подробное описание можно найти в справочной литературе [1] или в контекстной справке интегрированной среды Delphi.

5.3. Типы пользовательского интерфейса

Программные продукты, работающие в операционной среде Windows, по типу используемого интерфейса пользователя можно классифицировать следующим образом:

1. *Консольные* приложения;
2. *Одно-документный* интерфейс (*SDI-приложение*).
3. *Многодокументный* интерфейс (*MDI-приложение*).
4. *Форма со вкладками.*

Консольные приложения, т.е. программы, работающие в текстовом режиме и не имеющие собственного окна. Например, программы, написанные для MS DOC, как правило, работают в консольном режиме. Приведем пример простого приложения, которое выводит в консольном окне текст.

```
program ConsoleProg;
{$APPTYPE CONSOLE} // директива компилятора,
                   // определяющая приложение как консольное
{$R *.RES}
begin
  WriteLn ('Выводимый текст');
end;
```

5.3.1. SDI-интерфейс

Термин *SDI* (*Single Document Interface*) дословно означает одно-документный интерфейс. SDI-приложения способны загрузить и использовать одновременно только один документ. Ярким представителем такого класса программ является стандартное ПО просмотра текстовых файлов в Windows – программа "Блокнот".

Следует сказать несколько слов о термине *документ*. За время развития многозадачных операционных систем приложения, функционирующие в них, становятся все более *объекто-центричными*, т.е. они работают с неким центральным объектом (документом), в который могут быть внедрены другие внешние объекты, причем в общем случае внешние объекты могут создаваться и редактироваться отдельным специали-

зированным ПО. Но исходная программа остается при этом SDI-приложением, так как может работать только с одним объектом (или документом в широком смысле этого слова).

Способность одновременно работать только с одним объектом не мешает приложению использовать дополнительные формы, например диалоговые окна, панели инструментов и прочее. Примером может служить сама Delphi – огромное количество панелей инструментов, меню, разнообразных библиотек компонентов, взаимодействующих между собой форм... Но в целом она остается SDI-приложением, так как может загрузить и использовать одновременно только один объект.

5.3.2. MDI-интерфейс

Термин *MDI* (*Multiple Document Interface*) дословно означает многодокументный интерфейс и описывает приложения, способные загрузить и использовать одновременно несколько документов или объектов.

Обычно MDI-приложения состоят минимум из двух форм – *родительской* и *дочерней*. Свойство родительской формы `FormStyle` устанавливается равным `fsMDIForm`. Для дочерней формы устанавливается стиль `fsMDIChild`. Родительская форма служит контейнером, содержащим дочерние формы, которые могут перемещаться, изменять размеры, минимизироваться или максимизироваться только в *клиентской* области родительской формы. В приложении могут быть дочерние формы разных типов, например одна – для обработки изображений, а другая – для работы с текстом.

В MDI-приложении, как правило, требуется выводить несколько экземпляров объектов одного класса, каждый из которых должен быть создан перед использованием, т.е. под объект должна быть выделена память, и уничтожен (память должна быть освобождена), когда программа в объекте больше не нуждается.

В случае форм в Delphi это можно сделать автоматически с помощью вкладки `Forms` в опциях настройки проекта (меню

Project, команда Options ( , Shift+Ctrl+F11)). При этом автоматически создаваемые формы (Auto-create forms) автоматически будут уничтожаться и освобождать память.

Для динамического создания объекта необходимо использовать конструктор Create, в частности, для динамического создания формы Form1 необходим следующий код.

```
Form1:= TForm1.Create(Application);
```

Конструктор Create получает в качестве параметра потомка класса TComponent (базовый класс всех компонентов Delphi), который и будет владельцем формы. Можно передать параметр nil, создав форму без владельца, но тогда придется освобождать занимаемую память вручную с помощью программного кода Form1.Free. В примере (как обычно и делается), в качестве владельца выступает глобальный объект разрабатываемого приложения Application. Дело в том, что процесс удаления форм подчиняется *концепции владельцев объектов*: когда объект уничтожается, автоматически уничтожаются все объекты, которыми он владеет. Созданная описанным образом форма принадлежит объекту Application и уничтожается при закрытии приложения.

Класс TForm имеет несколько свойств, специфичных для MDI-приложений.

ActiveMDIChild – свойство возвращает дочерний объект TForm, имеющий в текущее время фокус ввода. Оно полезно, когда родительская форма содержит панель инструментов или меню, команды которых распространяются на открытую дочернюю форму.

Свойство MDIChildren является массивом объектов, предоставляющих доступ к созданным дочерним формам. MDIChildCount возвращает количество элементов в этом массиве. Обычно эти свойства используется при выполнении какого-либо действия над всеми открытыми дочерними формами. Вот, например, код миминизации всех дочерних форм.

```
for i:= MDIChildCount-1 downto 0 do
  MDIChildren[i].WindowState := wsMinimized;
```

Отметим, что если минимизировать дочерние формы в порядке возрастания элементов массива, цикл будет работать некорректно, так как после сворачивания каждого окна массив `MDIChildren` обновляется и пересортировывается, и возможен пропуск некоторых элементов.

Профессиональные MDI-приложения позволяют активизировать необходимое дочернее окно, выбрав его из списка в меню. Свойство `WindowMenu` определяет объект `TMenuItem`, который Delphi будет использовать для вывода списка доступных дочерних форм.

Из специфических MDI-событий класса `TForm` отметим, что событие `OnActivate` (отобразить, показать форму) родительской формы имеет место при переключении между дочерними формами.

Имеются также специфичные для MDI-форм методы. `Arrangelcons` выстраивает пиктограммы минимизированных дочерних форм в нижней части родительской формы. `Cascade` располагает дочерние формы каскадом, так что видны все их заголовки. `Next` и `Previous` вызывает переход от одной дочерней формы к другой. `Tile` выстраивает дочерние формы так, что они не перекрываются.

5.3.3. Форма со вкладками

Если информация, с которой должен работать пользователь программного продукта, разбивается на группы, то имеет смысл использовать иной тип интерфейса – *форма с несколькими вкладками* или *многостраничное окно*, как, например, в настройках интегрированной среды программирования Delphi.

Для создания форм со вкладками предназначен специальный элемент управления `PageControl` ( , страница `Win32`). На каждой странице данного элемента (каждая создаваемая

страница является объектом типа TTabSheet) можно размещать любые другие компоненты.

Для добавления или удаления вкладки при проектировании используются команды контекстного меню New Page или Delete Page, соответственно. Во время работы программы аналогичные действия выполняются вызовом конструктора Create или метода Free класса TTabSheet.

Основные свойства класса TPageControl:

- ActivePage – имя активной страницы (вкладки);
- ActivePageIndex – индекс активной страницы;
- PageCount – количество страниц;
- Pages[Index:integer] – индексированный список страниц-объектов типа TTabSheet, обычно используется для прямого доступа ко вкладкам;
- свойство MultiLine определяет, будут ли закладки размещаться в несколько рядов, если все они в один ряд не помещаются;
- TabPosition определяет место расположения ярлычков закладок: сверху (tpTop), снизу (tpBottom), слева (tpLeft) или справа (tpRight);
- стиль внешнего вида заголовков вкладок (ярлычков) определяется свойством Style: обычный (tsTabs), в виде кнопок (tsButtons), в виде плоских кнопок (tsFlatButtons);
- TabHeight и TabWidth – высота и ширина заголовков вкладок (при нулевом значении определяются автоматически);
- свойство ScrollOpposite определят способ перемещения закладок при размещении их в несколько рядов.

В компоненте имеется ряд методов, позволяющих оперировать страницами:

- CanShowTab – определяет, может ли пользователь выбрать указанную закладку;
- FindNextPage – возвращает следующую или предыдущую страницу по отношению к текущей;
- RowCount – возвращает число рядов закладок;

➢ SelectNextPage – делает активной следующую или предыдущую видимую страницу.

Основные события компонента – OnChanging и OnChange. Первое из них происходит непосредственно перед переключением на другую страницу после щелчка пользователя на новой закладке. При этом в обработчик события передается по ссылке параметр AllowChange – разрешение переключения. Если в обработчике задать AllowChange := false, то переключение не произойдет. Событие OnChange происходит сразу после переключения.

5.4. Основные стандартные компоненты

5.4.1. Надписи

Надписи (компонент Label, **A**, страница Standard палитры компонентов) используются для отображения текста без возможностей редактирования. Класс в библиотеке VCL, соответствующий компоненту Label, носит имя TLabel. Он получается добавлением символа "T" к имени компонента (верно и в отношении других компонентов).

Основное свойство класса TLabel, в котором задается выводимый текст – Caption. Следует иметь в виду, что оно имеет строковый тип, и для вывода численных значений необходимо воспользоваться функциями преобразования:

➢ IntToStr – преобразует целое число в строку;

➢ FloatToStr – преобразует действительное число в строковый тип ;

➢ FloatToStrF – аналогична предыдущей функции, но с возможностью формирования представления числа в виде строки (*форматирования*).

Естественно, для класса TLabel наиболее важными являются свойства форматирования текста. Alignment – определяет способ выравнивания текста по горизонтали: по левому краю (taLeftJustify), по правому (taRightJustify), по цен-

тру (taCenter). Свойство Layout определяет выравнивание текста по вертикали: tlTop – вверху, tlCenter – в середине, tlBottom – внизу. Оно имеет смысл, если свойство AutoSize равно false. Текст может быть автоматически отформатирован в несколько строк (перенос по словам), если свойство WordWrap установлено в true и размеры метки это позволяют.

5.4.2. Текстовое поле ввода

Текстовое поле ввода (компонент Edit,  , страница Standard) позволяет отобразить и редактировать при помощи клавиатуры строковые данные (свойство Text). Для ввода чисел необходимы функции преобразования: StrToInt (преобразует строку в число целого типа) или StrToFloat (преобразует строку в действительное число).

При изменении содержимого поля ввода происходит событие OnChange. Из свойств можно отметить ReadOnly – разрешает или запрещает возможность редактирования текста, отображаемого в свойстве Text.

5.4.3. Класс TCheckBox

Элемент *флажок* или *индикатор с флажком* (компонент CheckBox,  , страница Standard) используется для выбора одного из двух вариантов. Соответственно, элемент может находиться в одном из двух состояний: *включен* (свойство Checked равно true, флажок помечается галочкой) или *выключен*. Если свойство AllowGrayed равно true, то возможно и третье состояние флажка – *включен и закрашен серым*. Третье состояние обычно используется для того, чтобы показать, что имеются вложенные флажки, из которых установлена только часть, как, например, в опциях установок Windows. Для идентификации трех состояний используется свойство State, принимающее значения cbChecked (установлен), cbGrayed (установлен и закрашен серым), cbUnchecked (снят).

Класс `TCheckBox` имеет свойство `Caption`, с помощью которого можно задать поясняющую надпись. При изменении состояния наступает событие `OnClick`.

5.4.4. Списки

Элемент интерфейса типа *список* предназначен для визуализации выбора одного или нескольких вариантов из множества возможных. В Delphi для реализации подобной задачи существует несколько компонентов. Рассмотрим их.

`ListBox` ( , страница `Standard`) – отображает список элементов и позволяет пользователю выбрать из него необходимые строки. Свойство `Style`, установленное в `lbStandard` (значение по умолчанию) соответствует списку строк, другие значения позволяют отображать в списке не только текст, но и изображения. В список автоматически добавляются полосы прокрутки. Есть свойство `Columns` , которое определяет число столбцов, в которых будет отображаться список, если он не помещается целиком в окне компонента. Свойство `Sorted` позволяет упорядочить список по алфавиту.

Основное свойство компонента, содержащее массив списка строк – `Items`. Используется для очистки списка (метод `Clear`), а также для добавления (`Add`), вставки (`Insert`), перемещения (`Exchange`, `Move`) и удаления (`Delete`) элементов списка, в качестве которых могут выступать не только строки, но и объекты. Индекс выбранной строки можно определить по свойству `ItemIndex`. По умолчанию `ItemIndex = -1`, это означает, что ни один элемент списка не выбран. Если есть необходимость установить выбор по умолчанию (который будет показан в момент начала работы приложения), то сделать это можно, например, в обработчике события `OnCreate` формы, введя в него оператор вида

```
ListBox1.ItemIndex:=0;
```

В классе TListBox имеется свойство MultiSelect, разрешающее множественный выбор в списке. В этом случае проверить выбор элемента списка можно с помощью свойства Selected.

Компонент CheckListBox ( , страница Additional) аналогичен компоненту списка строк ListBox, за исключением того, что рядом с каждый элементом находится окно с флажком – индикатор, который пользователь может включать и выключать, помечая элементы списка. Имеется событие OnClickCheck, возникающее при изменении пользователем состояния каждого индикатора.

Компонент ComboBox ( , страница Standard) объединяет функции компонентов ListBox и Edit (окна редактирования). Он отображает список строк в развернутом виде или в виде выпадающего списка и позволяет выбрать необходимую строку непосредственно из списка или задать в качестве выбора собственный текст. Отличие ComboBox от схожего по функциям компонента ListBox заключается в следующем:

- ComboBox разрешает пользователю редактировать список, а ListBox не разрешает;
- в ComboBox список может быть развернут или свернут, а в ListBox он всегда развернут;
- ListBox допускает множественный выбор, а в ComboBox выбирается всегда только один элемент.

Внешний вид списка определяется свойством Style:

- csDropDown – выпадающий список со строками одинаковой высоты и с окном редактирования, позволяющий вводить или редактировать текст;
- csSimple – развернутый список со строками одинаковой высоты и с окном редактирования, также позволяющий вводить или редактировать текст;
- csDropDownList – выпадающий список со строками одинаковой высоты, не содержащий окна редактирования;

➢ csOwnerDrawFixed – выпадающий список типа csDropDown с графической прорисовкой элементов одинаковой высоты, задаваемой свойством ItemHeight;

➢ csOwnerDrawVariable – аналогичен предыдущему, но элементы списка могут иметь различную высоту.

Выбранный элемент или введенный текст можно определить по значению свойства Text. Индекс выбранного элемента списка определяются свойством ItemIndex.

Основное событие компонента OnChange наступает при изменении текста в окне редактирования в результате прямого редактирования текста или в результате выбора из списка.

5.4.5. Радиокнопки

Радиокнопки или *переключатели* (компонент RadioButton, , страница Standard) предназначены для выбора одного из нескольких альтернативных вариантов.

Основным свойством компонента является Checked – определяет, выбрана ли данная кнопка или нет. Поскольку в начале выполнения приложения обычно необходимо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство Checked надо установить в true в процессе проектирования.

Свойство Caption содержит надпись около кнопки. Значение свойства Alignment определяет, с какой стороны от кнопки появится надпись: слева (taLeftJustify) или, как принято по умолчанию, справа (taRightJustify).

Событие OnClick для компонента RadioButton вызывается только при выборе переключателя, т.е. если радиокнопка уже выбрана, то нажатие на ней левой кнопки мыши к наступлению события OnClick не приводит.

5.4.6. Кнопки

Одним из наиболее распространенных элементов управления являются *кнопки*. Свойства самой обычной кнопки инкапсулированы в классе TButton. (компонент Button, , стра-

ница Standard). Все основные свойства класса были перечислены в п. 5.2. Добавим описание еще трех свойств:

- если свойство Cancel = true, то при нажатии на клавишу Escape (Esc) вызывается событие OnClick данной кнопки (отметим, что на форме может быть только одна кнопка со свойством Cancel = true);
- если свойство Default = true, то при нажатии на клавишу Enter вызывается событие OnClick данной кнопки (отметим, что на форме может быть только одна кнопка со свойством Default = true);
- при создании окон диалога используется свойство ModalResult, которое обеспечивает при нажатии на кнопку установку значения аналогичного свойства родительской формы равным свойству ModalResult кнопки.

Из методов, присущих кнопкам, имеет смысл отметить один – Click. Выполнение этого метода эквивалентно щелчку на кнопке, т.е. вызывает событие кнопки OnClick. Метод используют для дублирования щелчка на кнопке какими-то другими действиями.

Кроме класса TButton, в Delphi есть компоненты кнопок с еще большим набором возможностей. Это BitBtn ( , страница Additional) и SpeedButton ( , страница Additional). Эти элементы интерфейса способны содержать в себе графические изображения (свойство Glyph), находиться в нажатом зафиксированном состоянии, работать в составе группы и др.

5.4.7. Панели

При разработке интерфейса ПО несколько функционально связанных элементов часто объединяют в одну группу. Для визуализации такого объединения обычно используют специальные контейнерные элементы – *панели*.

Первый из компонентов такого типа – Bevel ( , страница Additional) – не является контейнером. Он создает чисто зрительный эффект объединения элементов. Стиль отображе-

ния определяется свойством Style, которое может принимать значения bsLowered (утопленный) и bsRaised (приподнятый). Контур компонента определяется свойством Shape: прямоугольник (bsBox), рамка (bsFrame), пунктирная рамка (bsSpacer), верхняя, нижняя, левая и правая линии (соответственно – bsTopLine, bsBottomLine, bsLeftLine, bsRightLine).

Компонент Panel ( , страница Standard) представляет собой панель, которая служит контейнером, объединяющим в группу управляющие компоненты, компоненты ввода и отображения информации и другие, меньшие, контейнеры. Панель можно использовать также для построения полос состояния и инструментальных панелей. Объединение панелью таких компонентов, как радиокнопки (класс TRadioButton) обеспечивает их функционирование как единой группы – включение одной выключает остальные (отметим, что существует также компонент RadioGroup ( , страница Standard) – панель, содержащая радиокнопки, регулярно расположенные столбцами и строками). Внешний вид компонента определяется свойствами BorderStyle, BevelInner, BevelOuter, BevelWidth, BorderWidth.

Практически аналогичный компонент GroupBox ( , страница Standard) не имеет широких возможностей задания различных стилей оформления, но может иметь заголовок.

Инструментальная панель (компонент ToolBar,  , страница Win32) используется для быстрого доступа к часто используемым функциям приложения с помощью инструментальных быстрых кнопок и других компонентов. Размещаемые компоненты автоматически располагаются рядами и упорядочиваются по размерам. Для проектирования кнопок необходимо щелкнуть на ToolBar правой кнопкой мыши и выбрать из всплывшего меню команду New Button. На форме появится

очередная кнопка – объект типа TToolButton, по свойствам и внешнему виду подобный компоненту SpeedButton.

Специальный контейнер инструментальных панелей (компонент CoolBar,  , страница Win32) позволяет проектировать перестраиваемые панели, состоящие из *полос*, в которые могут включаться инструментальные панели и любые другие оконные компоненты (окна редактирования, панели и т.п.), каждый из которых автоматически снабжается средствами перемещения в пределах окна CoolBar. В полосы могут вставляться и не оконные компоненты, например, надписи (класс TLabel), но они не будут перемещаемыми.

Еще большими возможностями обладает компонент ControlBar ( , страница Additional). Отличие от CoolBar заключается в широком применении техники перетаскивания и встраивания Drag&Doc. Каждый компонент, попадая на ControlBar, получает полосу захвата, свойственную этой технологии. В дальнейшем возможно перемещать по ControlBar все эти компоненты и даже вынимать их, превращая в самостоятельные плавающие окна, что определяется свойством AutoDrag. Но чтобы воспользоваться этой возможностью, надо у компонентов, размещенных на ControlBar, установить свойства DragMode = dmAutomatic и DragKind = dkDock, что означает автоматическое выполнение операций Drag&Doc.

Свойства RowSize и RowSnap определяют *процедуру встраивания*: RowSize задает размеры полос, в которые могут встраиваться компоненты, а RowSnap определяет захват полосами встраиваемых компонентов. Свойство AutoDock, установленное в true, обеспечивает временное встраивание компонента, перетаскиваемого над панелью, в ControlBar. Это позволяет во время работы приложения наглядно представлять результат перетаскивания, и относится не только к компонентам, первоначально находившимся на панели ControlBar, но и к любому перетаскиваемому и встраиваемому компоненту.

5.4.8. Меню

Строка *главного меню* (компонент `MainMenu`,  , страница `Standard`) располагается в верхней части главной формы приложения. Доступ к командам осуществляется либо непосредственным выбором мышью пункта меню, либо с помощью специально назначенных клавиш клавиатуры. Проектирование производится с помощью *конструктора меню*, вызываемого двойным щелчком на компоненте. Команды `Create Submenu` позволяет ввести подменю в выделенный раздел.

Свойство `Items` содержит массив разделов меню типа `TMenuItem`, обладающих своими свойствами, методами, событиями. Свойство `Caption` обозначает надпись раздела, свойство `ShortCut` определяет клавиши быстрого доступа к разделу. Свойство `Default` определяет, является ли данный раздел разделом по умолчанию своего подменю, т.е. разделом, выполняемым при двойном щелчке пользователя на родительском разделе. Свойство `Break` используется в длинных меню, чтобы разбить список разделов на несколько столбцов. Свойство `Checked`, установленное в `true`, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран. Еще одним свойством, позволяющим вводить маркеры в разделы меню, является `RadioItem`. Это свойство, установленное в `true`, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства `GroupIndex`. Для каждого раздела могут быть установлены во время проектирования или программно во время выполнения свойства `Enabled` (доступен) и `Visible` (видимый). Предусмотрена возможность ввода в разделы меню изображений с помощью свойств `Bitmap` и `ImageIndex`. Первое из них позволяет непосредственно ввести изображение в раздел. Второе – указать индекс изображения во внешнем компоненте `ImageList`. Указание на этот компонент задается в свойстве `Images`.

Пункты меню в рамках одного раздела принято логически делить на группы. Друг от друга группы отделяются специальной линией – *разделителем*. Для реализации подобного эффекта достаточно присвоить пункту меню в качестве заголовка (`Caption`) одиночный символ "-".

Свойства и методы класса `TMainMenu` могут обеспечить объединение меню главной и вспомогательной форм. В MDI-приложениях меню дочерней формы всегда объединяется с меню главной формы. В SDI-приложениях при установке свойства `AutoMerge` в `true` меню вторичной формы также объединится с меню главной формы приложения. Положение добавляемых пунктов меню зависит от значения их свойств `GroupIndex`.

Основное событие раздела меню – `OnClick`, возникающее при выборе раздела с помощью мыши или при нажатии "горячих" клавиш и клавиш быстрого доступа.

Компонент `PopupMenu` ( , страница `Standard`) определяет *всплывающее контекстное меню*, появляющееся на экране при щелчке правой кнопкой мыши в поле практически любого компонента, который связан с данным меню своим свойством `PopupMenu`. Процесс проектирования контекстного меню и его свойства практически аналогичны компоненту `MainMenu`.

5.4.9. Таймер

Таймер (компонент `Timer`,  , страница `System`) позволяет задавать интервалы времени. Компонент не является визуальным, но тем не менее часто используется, например, при синхронизации мультипликации, для программного закрытия окон, для регулярного опроса источников информации, при контроле времени на ответ в обучающих программах и т.д.

Таймером можно управлять с помощью двух свойств: `Interval` – интервал времени в миллисекундах и `Enabled` – доступность. Если задать `Interval=0` и `Enabled=false`, то таймер перестанет работать, т.е. перестанет генерировать со-

бытие OnTimer через заданный интервал времени после предыдущего срабатывания, или после программной установки свойства Interval (или – альтернативный вариант – свойства Enabled), или после запуска приложения.

Заданный интервал времени выдерживается достаточно точно, только если он составляет сотни и тысячи миллисекунд. При меньших значениях, как правило, реальные интервалы времени оказываются существенно больше.

5.4.10. Визуализация больших текстовых фрагментов

Для визуализации больших текстовых фрагментов используются компоненты Memo ( , страница Standard, компонент визуализирует только текст) и RichEdit ( , страница Win32, визуализируется текст в обогащенном формате *rtf*).

Memo – многострочный текстовый редактор, позволяющий редактировать текст окна, в которое можно вводить множество строк. Формат всего текста одинаков и определяется свойством Font. Свойство Lines, доступное как во время проектирования, так и во время выполнения, имеет множество свойств и методов типа TStrings, которые обычно используются для формирования и редактирования текста. Весь текст содержится в свойстве Text.

Окно редактирования снабжено многими функциями, свойственными большинству редакторов. Например, в нем предусмотрены типичные комбинации "горячих" клавиш: Ctrl+C – копирование выделенного текста в буфер обмена Clipboard (команда Copy), Ctrl+X – вырезание выделенного текста в буфер Clipboard (команда Cut), Ctrl+V – вставка текста из буфера обмена в позицию курсора (команда Paste), Ctrl+Z – отмена последней команды редактирования.

Компонент RichEdit представляет собой многофункциональное средство редактирования текстов, позволяющее работать с обогащенным *rtf*-форматом, т.е. выбирать различные атрибуты форматирования для разных фрагментов текста. В

этом основное отличие RichEdit от подобного, но более простого компонента Memo.

Для изменения атрибутов вновь вводимого фрагмента текста задается свойство SelAttributes, которое в свою очередь имеет подсвойства: Color (цвет), Name (имя шрифта), Size (размер), Style (стиль) и ряд других. Например, если приложение имеет компонент RichEdit1 и диалог выбора шрифта FontDialog1, то следующий код позволит менять атрибуты вновь вводимого или выделенного текста:

```
if FontDialog1.Execute then
 with RichEdit1.SelAttributes do begin
  Color:=FontDialog1.Font.Color;
  Name :=FontDialog1.Font.Name;
  Size :=FontDialog1.Font.Size;
  Style:=FontDialog1.Font.Style;
 end;
RichEdit1.SetFocus;
```

Приведенный код можно сократить, воспользовавшись тем, что объекты SelAttributes и Font совместимы по типу.

```
if FontDialog1.Execute then
     RichEdit1.SelAttributes.Assign(FontDialog1.Font);
RichEdit1.SetFocus;
```

В компоненте имеется также свойство DefAttributes, содержащее атрибуты по умолчанию.

За выравнивание, отступы и т.д. в пределах текущего абзаца отвечает свойство Paragraph типа TParaAttributes. Этот тип в свою очередь имеет ряд свойств:

Alignment – определяет выравнивание текста;

FirstIndent – число пикселей отступа красной строки;

Numbering – управляет вставкой маркеров, как в списках;

LeftIndent – отступ в пикселях от левого поля;

RightIndent – отступ в пикселях от правого поля;

TabCount – количество позиций табуляции;

Tab – значения позиций табуляции в пикселях.

Значения подсвойств свойства `Paragraph` можно задавать только в процессе выполнения приложения. Значения подсвойств свойства `Paragraph` относятся к тому абзацу, в котором находится курсор. Например, каждый из следующих операторов осуществит соответственное выравнивание:

```
RichEdit1.Paragraph.Alignment:=taLeftJustify; // влево
RichEdit1.Paragraph.Alignment:=taCenter;     // по центру
RichEdit1.Paragraph.Alignment:=taRightJustify;// вправо
```

Следующий оператор приведет к тому, что текущий абзац будет отображаться как список, т.е. с маркерами:

```
RichEdit1.Paragraph.Numbering:=nsBullet;
```

Уничтожение списка в текущем абзаце осуществляется оператором

```
RichEdit1.Paragraph.Numbering:=nsNone;
```

Отметим, что компонент `RichEdit` также позволяет работать с графикой и OLE-объектами.

5.4.11. Визуализация структурированных данных

Более сложными элементами интерфейса являются компоненты визуализации структурированных данных, которые позволяют отображать данные в виде *дерева*, списков, таблиц, значков (*пиктограмм*).

Одним из таких компонентов является `ListView` ( , страница `Win32`), который позволяет отображать данные в виде списков, таблиц, крупных и мелких пиктограмм в стиле интуитивно понятного интерфейса Windows.

Стиль отображения информации определяется свойством `ViewStyle`, которое может устанавливаться в процессе проектирования или программно во время выполнения. Основное свойство компонента, описывающее состав отображаемой информации – `Items`. Во время проектирования оно может быть

установлено специальным редактором в окне Инспектора Объектов. В нем можно вводить новые узлы и дочерние узлы. Смысл дочерних узлов – это информация, которая появляется только в режиме `vsReport` (в виде таблицы). Для каждого нового узла можно указать ряд свойств. Свойство `Caption` – надпись, появляющаяся около пиктограммы (для дочерних узлов это свойство соответствует надписи, появляющейся в ячейках таблицы в соответствующем режиме). Свойство `ImageIndex` определяет индекс пиктограммы. Индексы соответствуют спискам изображений, хранящихся в отдельных компонентах `TImageList`, на которые указывают свойства `LargeImages` (для режима `vsIcon`) и `SmallImages` (для режимов `vsSmallIcon`, `vsList` и `vsReport`). Свойство узла `StateIndex` позволяет добавить вторую пиктограмму в данный объект для его дополнительной характеристики. Индекс соответствует списку изображений, хранящихся в отдельном компоненте `ImageList`, указанном в свойстве `StateImages` класса `TListView`.

Свойство `Columns` определяет список заголовков таблицы в режиме `vsReport` при свойстве `ShowColumnHeaders` (показать заголовки), установленном в `true`. Свойство `Checkboxes` определяет отображение индикатора с флажком около каждого элемента списка. Индикаторы можно устанавливать программно или их может изменять пользователь во время выполнения. Определить установку индикатора в некотором элементе `Items[i]` можно проверкой его свойства `Checked`. Свойства `HotTrack` и `HotTrackStyles` определяют появление выделения при перемещении курсора над элементом списка и стиль этого выделения. Свойство `HoverTime` задает в миллисекундах задержку появления такого выделения. Свойство списка `Selected` определяет выделенный пользователем элемент списка. Свойство `DragMode` определяет режим перетаскивания элементов списка.

Метод `Arrange` позволяет упорядочить пиктограммы в режимах `vsIcon` и `vsSmallIcon`. Упорядочивание пиктограмм

происходит в пределах той области, в которой они находятся. Способ упорядочивания определяется соответствующим заданием свойства SortType, определяющего характер сортировки.

Компонент TreeView ( , страница Win32) представляет собой окно для отображения иерархических данных в виде дерева, в котором пользователь может выбрать нужный ему узел или узлы. Иерархическая информация может быть самой разной – структура некоторого предприятия, структура документации учреждения, структура отчета и т.п. С каждым узлом дерева могут быть связаны некоторые данные.

Основным свойством класса TTreeView, содержащим информацию об узлах дерева, является индексированный список узлов Items. Каждый узел является объектом типа TTreeNodes, обладающим своими свойствами и методами, которые позволяют эффективно изменять внешний вид компонента и перестраивать дерево во время выполнения приложения.

Во время проектирования формирование дерева осуществляется в специальном редакторе узлов. Для каждого нового узла дерева можно указать ряд свойств: Text – надпись, появляющаяся в дереве около данного узла, ImageIndex и SelectedIndex – индексы пиктограмм, отображаемых для узла, который соответственно не выделен и выделен пользователем в данный момент (индексы соответствуют списку изображений, хранящихся в компоненте ImageList, на который указывает свойство Images класса TTreeView). Свойство узла StateIndex позволяет добавить вторую пиктограмму в данный узел, не зависящую от состояния узла, которая может служить дополнительной характеристикой узла. Свойство StateIndex соответствует списку изображений отдельного компонента ImageList (свойство StateImages).

Для отображения табличных данных используются компоненты DrawGrid (таблица, ячейки которой содержат графические изображения –  , страница Additional) и, если табли-

ца должна содержать текстовую или текстовую и графическую информацию, то следует использовать компонент StringGrid ( , страница Additional).

Основные свойства, доступные во время выполнения:

- Cells – строка, содержащаяся в ячейке с заданными индексами столбца и строки,
- Cols и Rows – списки строк, содержащихся соответственно в столбце или в строке с заданным индексом,
- Objects – объект, связанный со строкой, содержащейся в ячейке с заданными индексами столбца и строки.

Таблица может иметь полосы прокрутки (свойство ScrollBars), причем полосы прокрутки появляются и исчезают автоматически в зависимости от того, помещается таблица в соответствующий размер или нет. Заданное число первых строк и столбцов может быть фиксированным и не прокручиваться. Таким образом, можно задать заголовки столбцов и строк, постоянно присутствующие в окне компонента. Свойства ColCount и RowCount определяют соответственно число столбцов и строк, свойства FixedCols и FixedRows – число фиксированных столбцов и строк. Цвет фона фиксированных ячеек определяется свойством FixedColor. Свойства LeftCol и TopRow определяют соответственно индексы первого видимого на экране в данный момент прокручиваемого столбца и первой видимой прокручиваемой строки.

Свойство Options является множеством, определяющим многие свойства таблицы: наличие разделительных вертикальных и горизонтальных линий в фиксированных (goFixedVertLine и goFixedHorzLine) и не фиксированных (goVertLine и goHorzLine) ячейках, возможность для пользователя изменять с помощью мыши размеры столбцов и строк (goColSizing и goRowSizing), перемещать столбцы и строки (goColMoving и goRowMoving) и многое другое. Важным элементом в свойстве Options является goEditing – возможность редактировать содержимое таблицы.

Для отображения изображений компонент имеет свойство `Canvas` (некоторую область для рисования – *канву*). Имеется метод `CellRect`, возвращающий область канвы, отведенную под ячейку с заданными индексами столбца и строки.

5.4.12. Компоненты построения баз данных

Использование баз данных (БД) является одним из приоритетных направлений развития прикладного ПО. Среда Delphi всегда отличалась богатыми возможностями по поддержке механизмов доступа к базам данных и многоуровневым распределенным системам.

Изначально доступ к БД в Delphi обеспечивался *процессором баз данных BDE* (*Borland Database Engine*). Обращение к базам данных производится с помощью специальных компонентов, использующих функции BDE, обеспечивая доступ как к *локальным*, так и к *распределенным* БД. Взаимодействие осуществляется через *драйверы*. В поставку Delphi включены стандартные драйверы для работы с локальными базами данных dBase, Paradox и FoxPro, а также с SQL-серверами Oracle, Informix, Sybase, DB2 и InterBase. Кроме того, имеется возможность подключения любых драйверов ODBC.

Начиная с пятой версии, в Delphi добавлен новый механизм доступа к данным с использованием *технологии ADO* (*ActiveX Data Objects*), разработанной и поддерживаемой фирмой Microsoft и реализующей стандарт обращения этой фирмы к реляционным базам данных. Технология ADO аналогична BDE по назначению и довольно близка по возможностям.

При работе с базами данных удобно использовать специальные *модули данных*. Они представляют собой контейнеры, в которые можно помещать невизуальные компоненты доступа к данным. Новый модуль данных создается с помощью команды главного меню `File | New`, после чего из депозитария выбирается объект `Data Module`. При этом открывается окно *редактора модуля данных*, содержащее закладки `Components` и `Data Diagram`. Закладка `Components` предназначена для раз-

мещения компонентов доступа к данным. На странице `Data Diagram` отображается диаграмма связей между используемыми компонентами, которые можно проектировать визуально, а также сопровождать их комментариями. Для использования модуля данных он должен быть объявлен в разделе **uses** модуля проектируемого приложения.

Любое приложение, работающее с БД, должно обеспечивать ряд типовых функциональных возможностей:

- подключение к базе данных;
- считывание информации из таблиц этой БД;
- редактирование данных;
- навигацию по набору данных.

Набор данных представляет собой двумерную таблицу. Строки таблицы называются *записями*, а столбцы – *полями*.

Таблицы БД не загружаются в память полностью из-за их большого размера. Загружаются только значения полей, относящиеся к какой-либо записи таблицы. Запись, значения полей которой загружены в память, называется *текущей записью*. *Перемещение* по набору данных означает загрузку в выделенную для хранения текущей записи память новых значений полей из таблицы данных. С текущей записью связано понятие *курсора набора данных*, который представляет собой объект, содержащий значения текущей записи и инкапсулирующий методы, позволяющие загружать новые записи и при этом сохранять изменения, занесенные в текущую запись.

Для отображения и редактирования данных в VCL реализован ряд компонентов, специально ориентированных на работу с базами данных. Например, для организации доступа к БД используются следующие компоненты:

- для доступа к таблицам локальных баз данных и управление ими – `Table` ( , страница `BDE` палитры компонентов), `ADOTable` ( , страница `ADO`);
- компоненты `Query` ( , страница `BDE`) и `ADOQuery` ( , страница `ADO`) используют для доступа к БД SQL-запросы, по-

этому позволяет работать как с локальными, так и с распределенными базами данных.

При работе с реляционными БД важное значение имеет понятия *поля* – описание типа данных, которому соответствует значение в таблице базы данных. В наборе данных Delphi для представления полей используется базовый класс `TField`, обеспечивающий возможность работы с любыми типами данных, а на основе его определен ряд классов для представления типизированных полей:

- `TStringField` – строковое поле (длина строки – не более 8192 символа);
- `TSmallIntField` – целочисленное поле, хранящее данные в формате `ShortInt`;
- `TIntegerField` – целочисленное поле, хранящее данные в формате `Integer`;
- `TLargeField` – целочисленное поле, хранящее данные в формате `LongInt`;
- `TWordField` – целочисленное поле, хранящее данные в формате `Word`;
- `TBooleanField` – логическое поле для данных типа `Boolean`;
- `TFloatField` – поле для действительных чисел, тип данных `Double`;
- `TBLOBField` – поле, хранящее данные в формате большого двоичного массива, может содержать любые данные, представимые в виде двоичного объекта (физически в БД двоичные объекты хранятся в отдельных файлах, а поля содержат только ссылки на них);
- `TGraphicField` – поле для хранения изображений в формате BMP (фактически аналогично типу `BLOB`);
- `TArrayField` – массив полей любого типа кроме `TArrayField`;
- `TDataSetField` – поле, содержащее набор данных;
- `TMemoField` – поле для хранения списка строк.

Отметим, что существуют также *вычисляемые* поля, которые позволяют рассчитать свое значение на основе существующих данных, не изменяя при этом структуры таблицы БД.

Для полноценной работы с базами данных недостаточно обеспечить только доступ к информации. Необходимы также возможности визуализации и редактирования информации, хранящейся в БД.

На странице `Data Access` палитры компонентов VCL находится компонент `DataSource` (), который используется в качестве интерфейса для соединения набора данных с компонентами отображения данных. В качестве механизма взаимодействия используются как BDE, так и ADO.

Класс `TDataSource` содержит небольшое количество свойств и методов:

- `AutoEdit: Boolean` – если значение данного свойства равно `true`, то при попытке пользователя изменить значение поля в элементе отображения данных набор данных автоматически переводится в состояние редактирования (`dsEdit`);
- `DataSet: TDataSet` – указывает на набор данных, с которым связан объект `TDataSource`;
- `Enabled: Boolean` – определяет, отображать или нет данные в элементах отображения данных, связанных с данным объектом типа `TDataSource`;
- `State: TDataSetState` – содержит текущее состояние набора данных, связанного с компонентом `TDataSource`;
- **`procedure`** `Edit` – проверяет состояние набора данных перед переводом его в состояние `dsEdit`;
- **`function`** `IsLinkedTo(DataSet:TDataSet):Boolean` – проверяет, связан ли компонент типа `TDataSource` с набором данных `DataSet`, указанным в параметрах функции.

В классе `TDataSource` определена обработка событий:

- `OnDataChange` – вызывается при перемещении курсора набора данных, связанного с компонентом `DataSource`, если в текущую запись были внесены изменения;

➢ `OnStateChange` – вызывается при изменении состояния набора данных, связанного с компонентом `DataSource`;

➢ `OnUpdateData` – вызывается перед сохранением изменений в базе данных.

Как правило, элементы визуализации информации баз данных по свойствам, методам, событиям подобны аналогичным элементам управления Delphi, рассмотренным ранее. В наименовании компонентов обычно присутствует сочетание символов "`DB`", указывающее на то, что данный компонент ориентирован на работу с БД. Все элементы визуализации данных имеют общее свойство `DataSource`, указывающее на источник данных. Стандартные элементы расположены на странице `Data Controls` палитры компонентов.

Для визуализации набора данных в виде таблицы используется компонент `DBGrid` (). Структура таблицы компонента соответствует структуре набора данных: строки являются записями, а столбцы – полями. Отметим, что отображать все поля, содержащиеся в наборе данных, не обязательно.

Для доступа к отдельным полям БД также имеются соответствующие компоненты. Кроме свойства `DataSource`, они имеют свойство `DataField:`**`String`** – имя поля набора данных, из которого элемент отображения данных получает информацию. Кратко охарактеризуем эти компоненты.

`DBText` – компонент () отображает текущее значение поля набора данных без возможности редактирования. Аналогичен элементу `Label`.

`DBEdit` () – представляет собой обычную строку ввода, аналогичную `Edit`. В отличие от `DBText` с помощью данного компонента можно не только просматривать базу данных, но и редактировать ее.

`DBMemo` () – предназначен для отображения и редактирования полей, содержащих несколько строк текста (обычно это поля типа `TMemoField` или `TBLOBField`). Аналогичен компоненту `Memo`.

Для визуализации и редактирования сложного текста с различными стилями существует компонент DBRichEdit ().

DBCheckBox () – компонент (аналог CheckBox) предназначен для просмотра и редактирования данных, которые могут принимать только два значения. Состояние флажка определяется свойствами ValueChecked и ValueUnChecked, а также значением, содержащемся в поле, с которым связан компонент. По умолчанию ValueChecked=true и ValueUnChecked=false. Однако этим свойствам можно присваивать и строковые значения, причем одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой. Если значение поля соответствует одному из значений свойства ValueChecked, то флажок будет установлен; если поле соответствует одному из значений ValueUnChecked, то флажок будет снят. При изменении состояния флажка значение поля становится равным: при установке флажка – первому значению в списке ValueChecked, при сбросе флажка – первому значению в списке ValueUnChecked.

DBRadioGroup () – группа переключателей, состояние которых зависит от значения связанного с ним поля. Если текущее значение поля соответствует значению какого-либо переключателя, то он включается. При изменении состояния переключателей в поле заносится значение включенного переключателя. Свойство Items содержит список переключателей. Значения, на которые реагируют переключатели, определяются свойством-списком Values, причем каждому значению списка соответствует один переключатель. Текущее значение поля содержится в свойстве Value.

DBListBox () – компонент служит для отображения текущего значения поля данных и замены его на любое значение из списка. При этом значение поля должно совпадать с одним из элементов списка. В остальном компонент DBListBox ничем не отличается от подобного компонента ListBox.

DBComboBox () – отображает значение поля, связанного с данным компонентом, в строке редактирования. Текущее значение поля можно изменять, выбирая новое значение из выпадающего списка или редактируя текст в поле ввода. Аналогичен компоненту ComboBox.

DBImage () – используется для отображения графической информации, хранящейся в БД. Практически аналогичный компонент Image будет рассмотрен далее в пп. 5.7.

Компоненты, работающие с отдельными полями, не имеют встроенных средств для изменения положения курсора набора данных, добавления новых записей и удаления существующих записей. Поэтому при их использовании требуются дополнительные элементы управления, обеспечивающие *навигацию* по набору данных.

В палитре компонентов Delphi содержится компонент DBNavigator ( , страница Data Controls), позволяющий решить эту задачу. Компонент представляет собой набор кнопок, выполняющих следующие функции:

- перемещение курсора набора данных на следующую, предыдущую, первую и на последнюю записи;
- вставка новой пустой записи в текущую позицию курсора набора данных;
- удаление текущей записи;
- перевод набора данных в режим редактирования;
- запись изменений в набор данных;
- отмена изменений, внесенных в текущую запись;
- восстановление исходного значения записи.

Набор кнопок компонента DBNavigator определяется пользователем с помощью свойства VisibleButtons, имеющего тип TButtonSet. Связь компонента с набором данных устанавливается через свойство DataSource. Свойство ConfirmDelete позволяет включить запрос для подтверждения при удалении записи.

Нажатие кнопок в DBNavigator можно производить программно при помощи метода BtnClick(index:TNavigateBtn). Параметр index определяет "нажимаемую" кнопку.

Компонент обрабатывает два события:

- BeforeAction – вызывается *после* нажатия на кнопку элемента типа TDBNavigator, но *до выполнения* операции, связанной с этой кнопкой (обычно используется для запроса пользователя подтверждения на выполнение операции, приводящей к изменению данных);
- OnClick – вызывается *после* нажатия на кнопку и *после* выполнения операции, связанных с этой кнопкой.

5.5. Компоненты организации диалога

5.5.1. Окна сообщений

Наиболее простой тип диалога проектируемого приложения с пользователем можно организовать с помощью *окон сообщений*. Существует ряд функций для реализации подобных стандартных диалоговых окон. Из них рассмотрим подробно функцию MessageBox, которая является методом переменной Application типа TApplication (эта переменная доступна в любом проекте Delphi). Функция отображает диалоговое окно с заданными кнопками, сообщением и заголовком и позволяет проанализировать ответ пользователя. Во многих отношениях это окно подобно окнам, создаваемым такими функциями, как ShowMessage, ShowMessageFmt, MessageDlg, MessageDlgPos, CreateMessageDialog и др. Но функция MessageBox, в отличие от указанных функций, является наиболее удачным способом отображения полностью русифицированных диалоговых окон (соответственно, в русифицированных версиях Windows).

Функция MessageBox инкапсулирует одноименную функцию API Windows. Синтаксис ее объявления следующий:

```
function MessageBox(const Text, Caption: PChar;
                    Flags: Longint = MB_OK): Integer;
```

Параметр `Text` представляет собой текст сообщения, которое может превышать 255 символов (для длинных сообщений осуществляется автоматический перенос текста). Параметр `Caption` – это текст заголовка окна (также может превышать 255 символов, но не переносится).

Параметр `Flags` представляет собой множество флагов, определяющих вид и поведение диалогового окна. Этот параметр может комбинироваться операцией сложения по одному флагу из следующих групп:

- флаги кнопок, отображаемых в диалоговом окне;
- флаги пиктограмм в диалоговом окне;
- флаги, указывающие кнопку, которая в первый момент находится в фокусе;
- флаги модальности;
- дополнительные флаги.

Рассмотрим указанные группы более подробно.

Флаги кнопок, отображаемых в диалоговом окне (в скобках указаны надписи, которые будут отображаться в русифицированных версиях Windows):

- `MB_ABORTRETRYIGNORE` – кнопки Abort (Стоп), Retry (Повтор) и Ignore (Пропустить);
- `MB_OK` – кнопка OK (флаг принят по умолчанию);
- `MB_OKCANCEL` – кнопки OK и Cancel (Отмена);
- `MB_RETRYCANCEL` – кнопки Retry (Повтор) и Cancel (Отмена);
- `MB_YESNO` – кнопки Yes (Да) и No (Нет);
- `MB_YESNOCANCEL` – кнопки Yes (Да), No (Нет) и Cancel (Отмена).

Флаги пиктограмм, отображаемых в диалоговом окне:

- `MB_ICONEXCLAMATION`, `MB_ICONWARNING` – восклицательный знак (замечание, предупреждение);
- `MB_ICONINFORMATION`, `MB_ICONASTERISK` – буква i в круге (подтверждение);
- `MB_ICONQUESTION` – знак вопроса (ожидание ответа);

➢ MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND – знак креста на красном круге (запрет, ошибка).

Флаги, указывающие кнопку по умолчанию, т.е. кнопку, которая в момент визуализации окна находится в фокусе:

➢ MB_DEFBUTTON1 – первая кнопка (принято по умолчанию);

➢ MB_DEFBUTTON2 – вторая кнопка;

➢ MB_DEFBUTTON3 – третья кнопка;

➢ MB_DEFBUTTON4 – четвертая кнопка;

Флаги модальности:

➢ MB_APPLMODAL – пользователь должен ответить на запрос, прежде чем сможет продолжить работу с приложением. Разрешена работа с окнами другого приложения и со всплывающими окнами текущего ПО. Флаг принят по умолчанию.

➢ MB_SYSTEMMODAL – то же самое, что MB_APPLMODAL, но окно диалога отображается в стиле WS_EX_TOPMOST, то есть всегда остается поверх других окон, даже если пользователь перешел к другим приложениям. Используется для предупреждения о серьезных ошибках, требующих немедленного вмешательства.

➢ MB_TASKMODAL – то же самое, что MB_APPLMODAL, но окно диалога отображается как принадлежащее рабочему столу Windows. Флаг используется для сообщений из библиотеки или когда проектируемое ПО не имеет собственного окна.

Дополнительные флаги (могут задаваться оба флага):

➢ MB_HELP – добавляет в окно кнопку Help (Справка), щелчок на которой или нажатие клавиши F1 генерирует событие Help;

➢ MB_TOPMOST – помещает окно сообщения всегда поверх других окон (в стиле WS_EX_TOPMOST).

Возможны еще некоторые флаги, определяющие характер поведения окна при работе в сети нескольких пользователей, позволяющие отображать тексты справа налево (для восточных языков) и т.п.

Функция возвращает нуль, если не хватает памяти для создания диалогового окна. Если же функция выполнена успешно, то ее значение идентифицирует выбранную кнопку (табл. 5.1).

Таблица 5.1.

Значение	Численное значение	Пояснение
`IDABORT`	3	выбрана кнопка Abort (Стоп)
`IDCANCEL`	2	выбрана кнопка Cancel (Отмена) или нажата клавиша `Esc`
`IDIGNORE`	5	выбрана кнопка Ignore (Пропустить)
`IDNO`	7	выбрана кнопка No (Нет)
`IDOK`	1	выбрана кнопка OK
`IDRETRY`	4	выбрана кнопка Retry (Повтор)
`IDYES`	6	выбрана кнопка Yes (Да)

В качестве примера приведем код, предусматривающий проверку существования файла (путь содержится в переменной `FName='C:/MyFile.txt'`) и вывод соответствующих сообщений, а на рис. 5.1 показан внешний вид спроектированных диалоговых окон. Отметим, что если визуализируемый текст сообщения является результатом "склеивания" строк, то в этом случае необходимо применять операцию прямого приведения типов – `PAnsiChar(<переменная типа` **`string`**`>)`.

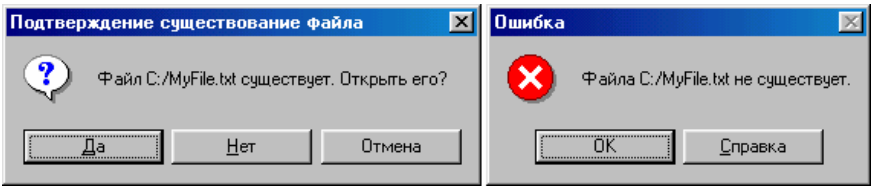


Рис. 5.1. Внешний вид диалоговых окон, спроектированных с помощью функции `MessageBox`

```
if FileExists(FName) // проверяем существование файла
 then Begin
  if (Application.MessageBox(
      PAnsiChar('Файл '+FName+' существует. Открыть
      его?'), 'Подтверждение существования файла',
      MB_YESNOCANCEL + MB_ICONQUESTION) <> IDYES)
   then begin
          ... // операторы открытия файла
        end
      End
 else begin
       Application.MessageBox(
        PAnsiChar('Файла '+FName+' не существует.'),
        'Ошибка', MB_ICONSTOP+MB_HELP);
      end;
```

5.5.2. OpenDialog, SaveDialog и другие компоненты стандартных диалоговых окон

Кроме сообщений, в прикладных программах широко используются диалоговые окна открытия и сохранения файлов, печати, выбора параметров шрифта и т.п. Для реализации диалогов подобного типа в интегрированной среде Delphi существуют специальные стандартные компоненты, расположенные на странице `Dialogs` палитры компонентов.

Компоненты вызова стандартных диалогов Windows для открытия и сохранения файлов `OpenDialog` (), `SaveDialog` () и `OpenPictureDialog` (), `SavePictureDialog` () отображают соответствующие модальные диалоговые окна Windows. Компоненты `OpenDialog` и `SaveDialog` работают с файлами любого типа, а компоненты `OpenPictureDialog` и `SavePictureDialog` – с файлами изображений.

Открытие соответствующего диалога осуществляется методом `Execute`. Если в диалоге пользователь нажмет кнопку "Открыть" ("Сохранить"), диалог закрывается, метод `Execute` возвращает `true` и выбранный файл отображается в свойстве компонента-диалога `FileName`. Если же пользователь отказал-

ся от диалога (нажал кнопку "Отмена" или клавишу Esc), то метод Execute возвращает false.

Значение свойства FileName можно задать и перед обращением к диалогу. Тогда оно появится в диалоговом окне как значение по умолчанию. Таким образом, например, код выполнение команды "Сохранить как ...", в результате выполнения которой в файле с выбранным именем необходимо сохранить текст окна редактирования Memo1, может иметь вид:

```
With SaveDialog1 do BEGIN
  FileName:=FName; // имя по умолчанию
  if Execute then begin
   FName:=FileName;
   Memo1.Lines.SaveToFile(FName);
                  end;
                    END;
```

В этом коде предполагается, что имя файла хранится в строковой переменной FName. Перед вызовом диалога это имя передается в него как имя файла по умолчанию, а после выбора пользователем файла, его выбор запоминается в той же переменной, и текст сохраняется в выбранном файле методом SaveToFile класса TMemo.

Другой компонент FontDialog () вызывает стандартный диалог Windows для выбора шрифта. Открытие диалога также осуществляется методом Execute. Если в диалоге выбрана кнопка "ОК" (или нажата клавиша Enter), диалог закрывается, метод Execute возвращает true и выбранные атрибуты шрифта передаются в свойство Font компонента-диалога. Если же пользователь отказался от диалога (нажал кнопку "Отмена" или клавишу Esc), то метод Execute возвращает false.

Значение свойства Font также можно задать перед вызовом диалогового окна. Тогда оно определит значения атрибутов шрифта по умолчанию, которые увидит пользователь в момент открытия диалога. Таким образом, например, выпол-

нение команды "Шрифт", по которой пользователь может задать текущее значение шрифта для компонента RichEdit1, может иметь вид:

```
// Задание в качестве атрибутов по умолчанию
// атрибутов шрифта текущей позиции курсора в тексте
FontDialog1.Font.Assign(RichEdit1.SelAttributes);
// Открытие диалога
if(FontDialog1.Execute) then
  RichEdit1.SelAttributes.Assign(FontDialog1.Font);
```

Свойство Device определяет, из какого списка возможных шрифтов будет предложен выбор в диалоговом окне: fdScreen – из списка экрана (по умолчанию), fdPrinter – из списка принтера, fdBoth – из обоих.

Свойство Options содержит множество опций диалога. Например, если установить опцию fdApplyButton, то в диалоговом окне появится еще одна кнопка "Применить", при нажатии которой возникает событие OnApply, в обработчике которого можно написать код, применяющий выбранные атрибуты без закрытия диалогового окна. Например:

```
RichEdit1.SelAttributes.Assign(FontDialog1.Font);
RichEdit1.Invalidate; // перерисовать компонент
```

Тогда можно, не прерывая диалога, наблюдать изменения непосредственно в окне RichEdit1, нажимая в диалоговом окне кнопку "Применить". При работе с компонентом класса TMemo аналогичный оператор может иметь вид:

```
Memo1.Font.Assign(FontDialog1.Font);
```

Свойства MaxFontSize и MinFontSize устанавливают ограничения на максимальный и минимальный размеры шрифта (действуют только при включенной опции fdLimitSize).

Компонент ColorDialog () отображает диалоговое окно Windows для выбора цветов. Диалог активизируется мето-

дом Execute. Когда пользователь выбирает цвет и нажимает "OK", диалог закрывается, и выбранный цвет сохраняется в свойстве Color.

Для визуализации диалоговых окон, связанных с печатью, в Delphi существуют компоненты PrinterSetupDialog () для отображения диалога установки параметров принтера и PrintDialog (), который отображает стандартное диалоговое окно для посылки заданий на принтер. В последнем можно выбрать принтер и установить его свойства, указать число копий и последовательность их печати, печатать в файл или непосредственно на принтер, выбрать печатаемые страницы или печатать только выделенный фрагмент и т.д. Отметим, что компонент PrintDialog не осуществляет печать. Он только позволяет пользователю задать атрибуты печати, которые могут быть прочитаны приложением как ряд свойств. А сама печать должна осуществляться программно с помощью объекта типа TPrinter или иным путем.

Задачи поиска и замены текстовых фрагментов реализуются с помощью компонентов FindDialog ( , отображает диалоговое окно поиска в тексте заданного фрагмента) и ReplaceDialog ( , диалог контекстного поиска и замены).

Диалоги вызываются методом Execute. Сами по себе компоненты не осуществляют ни поиска, ни замены, они только обеспечивают интерфейс с пользователем. Поиск и замена должна осуществляться программно с помощью обработки событий OnFind (происходит при нажатии в диалоговом окне кнопки "Найти далее") и OnReplace (использование кнопок "Заменить" или "Заменить все", причем идентификация осуществляется по значениям флагов frReplace и frReplaceAll).

5.6. Средства управления конфигурацией

Задачи *управления конфигурацией*, т.е. начальная инициализация или настройки проектируемого приложения изначально (в 16-разрядных версиях Windows 3.x) решались с помощью *текстовых INI-файлов*. Работу с такими файлами проще всего осуществлять с помощью создания в приложении объекта типа `TIniFile`.

В них сохраняемая информация о различных настройках логически группируется в разделы, каждый из которых начинается оператором заголовка, заключенным в квадратные скобки (например, `[Desktop]`). В строках, следующих за заголовком, содержится информация, относящаяся к данному разделу, в форме: `<ключ>=<значение>` или `<keyname>=<value>`.

Начиная с Windows 95, использование *INI*-файлов не рекомендовано и вместо типа `TIniFile`, инкапсулирующего свойства подобных файлов, используются типы `TRegistry`, `TRegIniFile` и `TRegistryIniFile`, инкапсулирующие свойства *системного реестра* (*registry*).

Реестр – это база данных для хранения информации о системной конфигурации аппаратуры, о Windows и о приложениях Windows. Реестр имеет иерархическую организацию, которая содержит много уровней ключей, субключей и параметров. В реестре данные делятся на две категории: *характеризующие компьютер* и *характеризующие пользователя*. Характеристики компьютера включают в себя все, связанное с техническими средствами, а также с установленными приложениями и их конфигурацией. Характеристики пользователя включают в себя установки по умолчанию для экрана, пользовательские конфигурации, информацию о выбранных пользователем принтерах, установки сети и т.д.

Шесть основных ключей реестра содержат все субключи и записи низшего уровня. Четыре ключа верхнего уровня предназначены для характеристик компьютера:

- `Hkey_Local_Machine` – информация о компьютере, включая конфигурацию установленной аппаратуры и программного обеспечения;
- `Hkey_Current_Config` – информация о текущем оборудовании;
- `Hkey_Dyn_Data` – динамические данные о состоянии, используемые процедурами plug-and-play;
- `Hkey_Classes_Root` – информация об OLE, drag&drop, клавишах быстрого доступа и пользовательском интерфейсе.

Два ключа верхнего уровня, предназначенные для характеристик пользователя:

- `Hkey_Users` – информация о пользователях, включая установки экрана и приложений;
- `Hkey_Current_User` – информация о пользователе, зарегистрированном в данный момент.

Реестр хранится в файле `SYSTEM.DAT` в каталоге `Windows`. Просмотр и редактирование реестра осуществляется стандартным в Windows *редактором реестра* `REGEDIT.EXE`.

Хотя разработчики Windows и не рекомендуют использование *INI*-файлов, тем не менее их логически естественная структура и доступность для редактирования в любом текстовом редакторе предопределили их предпочтительность по сравнению с использованием реестра, особенно для прикладных программных средств. Отметим, что логическая организация *INI*-файлов подходит и для хранения расчетных данных.

Когда в приложении создается объект типа `TIniFile`, ему передается как `FileName` (и это единственное свойство данного класса) имя файла, с которым он связан, причем расширение может быть любым. Методы `TIniFile` позволяют читать информацию из файла, записывать, удалять разделы и т.п.

Например, оператор создания объекта `Ini` типа `TIniFile` и связывания его с файлом `MyConfig.cfg`, находящемся в одном каталоге с проектируемым программным обеспечением, имеет следующий вид.

```
var Ini:TIniFile;
...
Ini:=TIniFile.Create('MyConfig.cfg');
```

Следующий оператор проверяет наличие в файле раздела `My Section`, содержащего ключ `MyKey`.

```
if Ini.ValueExists('My Section','MyKey') then ...
```

Следующий оператор заносит в ключ `MyKey` раздела `My Section` значение '5'.

```
Ini.WriteInteger('My Section','MyKey',5);
```

Следующий оператор удаляет ключ `MyKey` раздела `My Section`.

```
Ini.DeleteKey('My Section','MyKey');
```

Следующие операторы сохраняют содержимое объекта `Ini` в файле на диске и разрушают объект `Ini`.

```
Ini.UpdateFile;
Ini.Free;
```

Вместо класса `TINIFile` возможно применение объектов типа `TStringList`, близких по возможностям.

5.7. Работа с графикой

Работа с графическими изображениями в прикладных программах имеет большое значение, т.к. возможность использования графики является основной отличительной особенностью интерфейса операционных систем типа Windows по сравнению с MS DOS.

Отобразить на форме графическое изображение возможно с помощью компонента `Image` ( , страница `Additional`). Свойство `Picture` типа `TPicture` содержит отображаемый

графический объект типа битовой матрицы, пиктограммы, метафайла или определенного пользователем типа.

Чтобы создать на форме или модуле данных изображение, которым управляют другие элементы, необходимо использовать другой компонент – `ImageList` ( , страница `Win32` палитры компонентов). Это набор изображений одинаковых размеров, на которые можно ссылаться по индексам (начинающимся с нуля) в меню, инструментальных панелях и других компонентах. Изображения в `ImageList` могут загружаться и удаляться в процессе проектирования с помощью специального *редактора списков изображений*.

Многие компоненты Delphi имеют свойство `Canvas` (канва) типа `TCanvas`, которое позволяет создавать и редактировать графические изображения в области, определяемой свойством `ClipRect`. Канва используется для рисования *пером* `Pen` (класс `TPen`) и *кистью* `Brush` (класс `TBrush`), для модификации изображения или наложения друг на друга нескольких изображений. В классе `TImage` канва может использоваться только в случае, если в свойство `Picture` загружена битовая матрица или ничего не загружено. Если в этом свойстве находится объект, отличный по типу от `TBitMap`, то при обращении к `Canvas` генерируется исключение `EInvalidOperation`. Если же в `Picture` находится битовая матрица, то `Canvas` можно использовать для редактирования и создания изображений с помощью методов, которые укрупненно возможно разделить на методы рисования графических примитивов (линий, окружностей, прямоугольников и.т.д.), методы работы с областью графического изображения и методы вывода текста.

Приведем достаточно полный список методов `TCanvas`:

- **`procedure`** `Arc(X1,Y1,X2,Y2,X3,Y3,X4,Y4: Integer)` – рисует дугу окружности или эллипса; `(X1,Y1)` и `(X2,Y2)` определяют описанный прямоугольник, `(X3,Y3)` и `(X4,Y4)` – точки, через которые проходят радиусы, отмечающие начало и конец дуги;

➢ **procedure** Chord(Xl,Y1,X2,Y2,X3,Y3,X4,Y4: Integer) – рисует замкнутую фигуру, ограниченную дугой окружности или эллипса и хордой; (X1,Y1) и (X2,Y2) определяют описанный прямоугольник, (X3,Y3) и (X4,Y4) – точки, через которые проходит хорда;

➢ **procedure** Draw(X,Y: Integer; Graphic: TGraphic) – рисует графическое изображение Graphic в указанную позицию канвы (X,Y – левый верхний угол);

➢ **procedure** Ellipse(Xl,Y1,X2,Y2: Integer); **overload**; **procedure** Ellipse(**const** Rect: TRect); **overload** – рисует окружность или эллипс; (XI,Y1) и (X2,Y2) или Rect определяют прямоугольную область, в которую будет вписан рисуемый элемент;

➢ **procedure** FillRect(**const** Rect: TRect) – заполняет указанный прямоугольник канвы, используя текущее значение кисти Brush;

➢ **procedure** LineTo(X,Y: Integer) – рисует на канве прямую линию, начинающуюся с текущей позиции пера и кончающуюся указанной (X,Y) точкой (исключая ее);

➢ **procedure** MoveTo(X,Y: Integer) – изменяет текущую позицию пера на заданную без рисования;

➢ **procedure** Pie(X1,Y1,X2,Y2,X3,Y3,X4,Y4: Longint) – рисует сектор окружности или эллипса; (X1,Y1) и (X2,Y2) определяют описанный прямоугольник, а (X3,Y3) и (X4,Y4) определяют точки, через которые проходят радиусы, ограничивающие сектор;

➢ **procedure** PolyBezier(**const** Points: **array of** TPoint) – рисует на канве текущим пером кусочную кривую третьего порядка, сглаживающую заданное множество точек Points, число точек должно быть на единицу больше числа, кратного 3 (т.е. i*3+l);

➢ **procedure** PolyBezierTo(**const** Points: **array of** TPoint) – рисует на канве текущим пером кусочную кривую

третьего порядка, сглаживающую заданное множество точек Points, число точек должно быть кратно 3 (т.е. i*3);

- **procedure** Polygon(Points: **array of** TPoint) – рисует замкнутую фигуру с кусочно-линейной границей;
- **procedure** Polyline(Points: **array of** TPoint) – рисует разомкнутую кусочно-линейную кривую;
- **procedure** Rectangle(Xl,Y1,X2,Y2: Integer); **overload**; **procedure** Rectangle(**const** Rect: TRect); **overload** – рисует прямоугольник, заданный углами (X1,Y1) и (X2,Y2) или Rect;
- **procedure** RoundRect(Xl,Y1,X2,Y2,X3,Y3: Integer) – рисует прямоугольник со скругленными углами; (X1,Y1) и (X2,Y2) – прямоугольник, X3 и Y3 – ширина и высота эллипса скругления;
- **procedure** StretchDraw(**const** Rect: TRect: Graphic: TGraphic) – рисует графическое изображение Graphic в указанную прямоугольную область канвы Rect, подгоняя размер изображения под заданную область;
- **function** TextExtent(**const** Text: **string**): TSize – возвращает длину и высоту в пикселях текста, который предполагается написать на канве текущим шрифтом;
- **function** TextHeight(**const** Text: **string**): Integer – возвращает высоту в пикселях текста, который предполагается написать на канве текущим шрифтом;
- **procedure** TextOut(X,Y: Integer; **const** Text: **string**) – пишет указанную строку текста Text на канве, начиная с указанной (X,Y) позиции;
- **procedure** TextRect(Rect: TRect; X,Y: Integer; **const** Text: **string**) – пишет указанную строку текста Text на канве, начиная с указанной позиции и усекая текст, выходящий за пределы указанной прямоугольной области Rect;
- **function** TextWidth(**const** Text: **string**): Integer – возвращает длину в пикселях текста Text, который предполагается написать на канве текущим шрифтом.

Событий при изменении изображения возникает два: перед изменением – `OnChanging` и после изменения изображения – `OnChange`.

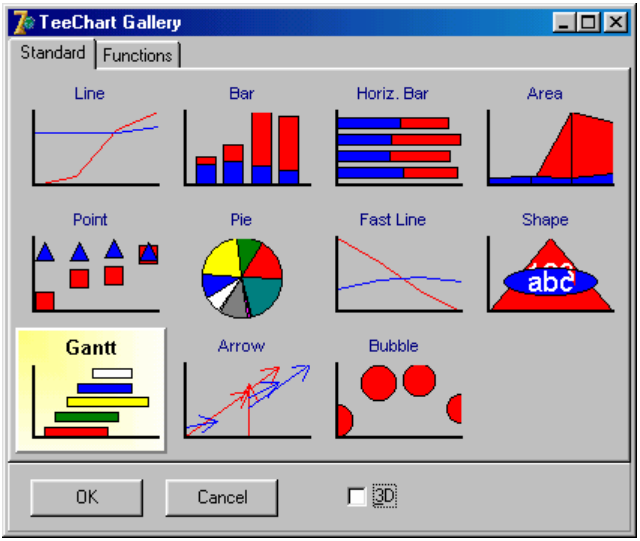
Отметим, что для рисования простых геометрических фигур в Delphi существует компонент `Shape` ( , страница `Additional`). Основное свойство компонента – `Shape` (форма), которое может принимать значения: `stRectangle` (прямоугольник), `stRoundRect` (прямоугольник со скругленными углами), `stSquare` (квадрат), `stRoundSquare` (квадрат со скругленными углами), `stEllipse` (эллипс), `stCircle` (круг). Свойство компонента `Brush` (кисть) типа `TBrush`, определяет заполнение фигуры: цвет (`Brush.Color`) и стиль (`Brush.Style`). Свойство `Pen` (перо) типа `TPen` определяет стиль линий.

Для создания диаграмм и графиков существует компонент `Chart` ( , страница `Additional`). Компонент является контейнером объектов `Series` типа `TChartSeries` – серий данных, характеризующихся различными стилями отображения. Каждый компонент может включать несколько серий. Множество свойств класса `TChart` определяют оформление графика, оси координат (они могут быть со всех четырех сторон), трехмерную имитацию отображения и т.п. На рис. 5.2 показаны стили визуализации графиков и диаграмм – в двухмерном варианте (рис. 5.2 а) и трехмерном (рис. 5.2 б).

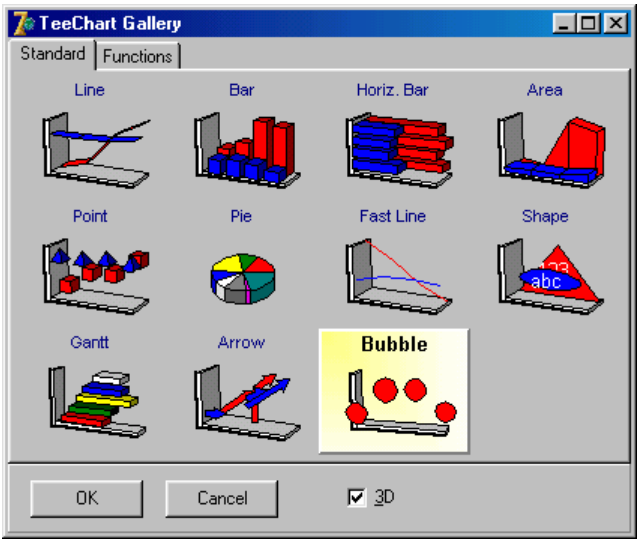
Свойства объектов типа `TChart` удобно устанавливать специальным *Редактором Диаграмм*, вызываемым из Инспектора Объектов нажатием кнопки с многоточием около соответствующего свойства или двойным щелчком на компоненте.

Для задания отображаемых значений используются методы серий `Series`. Основные из них:

- `Clear` – очищает серию от занесенных ранее данных;
- `Add` – позволяет добавить в диаграмму новую точку;
- `AddXY` – позволяет добавить новую точку в график функции.



а)



б)

Рис. 5.2. Стили визуализации графиков и диаграмм компонента `Chart`

Например, следующие операторы очищают серию Series1 и заносят в нее для отображения диаграммы четыре значения, задавая отображающие их цвета:

```
With Series1 do begin
  Clear;
  Add(A1,'Значение 1',clYellow);
  Add(A2,'Значение 2',clBlue);
  Add(A3,'Значение 3',clRed);
  Add(A4,'Значение 4',clPurple);
                end;
```

Следующие операторы заносят в серию Series2 значения, предназначенные для отображения красным цветом графика функции y=sin(x):

```
Series2.Clear;
for i:=0 to 100 do
  Series2.AddXY(0.02*Pi*i,sin(0.02*Pi*i),'',clRed);
```

Следующий оператор переносит данные серии Series1 в серию Series3, свойства которой, например, определяют отличный от Series1 тип диаграммы:

```
Series3.Assign(Series1);
```

Следующие операторы обеспечивают смену типа диаграммы, переключая видимость серий Series1 и Series3:

```
Series1.Active:= not Series1.Active;
Series3.Active:= not Series3.Active;
```

Свойство AllowPanning компонента TChart разрешает пользователю прокручивать графики и те типы диаграмм, в которых предусмотрены координатные оси. Прокрутку пользователь может осуществлять во время выполнения, нажимая правую кнопку мыши и "буксируя" ею график.

Свойство AllowZoom разрешает увеличение размера выбранного фрагмента графика или диаграммы с осями коорди-

нат, растягивая его на все видимое поле. Для этого необходимо с помощью левой кнопки мыши обвести рамкой требуемый фрагмент. Построение рамки вниз и вправо растягивает фрагмент на всю область изображения. Построение рамки вверх и влево восстанавливает исходный масштаб. Можно также восстановить исходный масштаб методом `UndoZoom`. Например, следующий оператор, вставленный в обработчик события `OnMouseDown`, восстанавливает масштаб, если пользователь нажимает кнопку мыши при нажатой клавише `Alt`:

```
if (ssAlt in Shift) then  Chart1.UndoZoom;
```

Масштаб можно также изменять методами `ZoomPercent` и `ZoomRect`.

По умолчанию весь график или диаграмма размещаются на одной видимой целиком странице. Если задать `MaxPointPerPage` (максимальное число точек на страницу), то изображение будет автоматически разбито на несколько страниц (число страниц – `NumPages`). На экране одновременно можно видеть одну страницу (определяется свойством `Page`). Перемещение по страницам возможно с помощью прокрутки графика пользователем (если оно разрешено свойством `AllowPanning`) или с помощью методов `PreviousPage` и `NextPage`.

Среди множества свойств серий можно отметить `Mark` – ярлычки, отображающие численные значения точек серии.

ЗАКЛЮЧЕНИЕ

Появление новых поколений ЭВМ обусловлено расширением сферы их применения. В настоящее время ведутся интенсивные работы по созданию ЭВМ пятого поколения как традиционной (неймановской) архитектуры, так и по созданию и апробации перспективных архитектур и схемотехнических решений. На формальном и прикладном уровнях исследуются архитектуры на основе параллельных абстрактных вычислителей (матричные и клеточные процессоры, систолические структуры, однородные вычислительные структуры, нейронные сети и др.). Вычислительные средства пятого поколения, кроме более высокой производительности и надежности при более низкой стоимости, что обеспечивается новейшими электронными технологиями, должны удовлетворять качественно новым *функциональным требованиям:*

- работать с базами знаний в различных предметных областях и, в частности, организовывать на их основе системы искусственного интеллекта;
- обеспечивать простоту применения ЭВМ путем реализации эффективных систем ввода-вывода информации голосом, диалоговой обработки информации с использованием естественных языков, устройств распознавания речи и изображения;
- упрощать процесс создания программных средств путем автоматизации синтеза программного обеспечения.

Все вышеперечисленное предполагает и новые требования к программированию на языках высокого уровня. Например, проблема создания эффективных систем параллельного программирования, ориентированных на высокоуровневое распараллеливание алгоритмов вычислений и обработки данных, представляется достаточно сложной и предполагает дифференцированный подход с учетом сложности распараллеливания и необходимости синхронизации процессов во времени.

Такое важное направление развития вычислительных средств пятого и последующих поколений, как интеллектуализация, связана с наделением ЭВМ элементами интеллекта, интеллектуализацией интерфейса с пользователем и др. Работа в данном направлении затрагивает, в первую очередь, программное обеспечение. ЭВМ при этом должна обладать способностью к обучению, к ассоциативной обработке информации, к ведению интеллектуального диалога при решении конкретных задач. Все это – перспективные направления для развития программирования на языках высокого уровня.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Архангельский А.Я. Delphi 6. Справочное пособие – М.: ЗАО "Издательство БИНОМ", 2001.

2. Архангельский А.Я. Программирование в Delphi 6. – М.: ЗАО "Издательство БИНОМ", 2001. – 1024 с.

3. Гультяев А.К. Help. Разработка справочных систем: Учебный курс. – СПб.: Питер, 2004. – 270 с.

4. Культин Н.Б. Программирование на Object Pascal в Delphi 5. – СПб: БВХ – Санкт-Петербург, 2000. – 464 с.

5. Кэнту М. Delphi 6 для профессионалов. – СПб: Питер, 2002. – 1088 с.

6. Марков А.С., Лисовский К.Ю. Базы данных. Введение в теорию и методологию: Учебник. – М.: Финансы и статистика, 2004. – 512 с.

7. Нортон П., Соухе Д. Язык ассемблера для IBM PC: Пер. с англ. – М.: Издательство "Компьютер", 1993. – 352 с.

8. Петров В.П. Информационные системы. – СПб.: Питер, 2002. – 688 с.

9. Тюкачев Н., Свиридов Ю. Delphi 5. Создание мультимедийных приложений. Учебный курс. – СПб.: Питер, 2001. – 400 с.

10. Фигурнов В.Э. IBM PC для пользователя. Краткий курс. – М.: ИНФРА, 1998. – 480 с.

11. http://www.informika.ru

12. http://www.rk6.bmstu.ru/ums/OPD/PrVysUr.htm

ОГЛАВЛЕНИЕ

Учебное издание

Романов Андрей Владимирович

ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ В СРЕДЕ DELPHI

Компьютерный набор А.В. Романов

Подписано к изданию 15.09.2006.
Уч.-изд. л. 8,6.

ГОУВПО «Воронежский государственный технический университет» 394026 Воронеж, Московский просп., 14