МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕГО СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ

И.Е. Жуковская

ОБЪЕКТНО−ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

(Учебное пособие)

ТАШКЕНТ−2006

Учебное пособие по предмету «Объектно-ориентированное программирование» предназначено для студентов бакалавриата, обучающихся по направлению 5521900−Информатика и информационные технологии. Оно содержит пять глав, расположенных в логической последовательности и подразумевает использование новых информационных, и педагогических технологий.

Составитель: © Жуковская И.Е. предмет «Объектно-ориентированное программирование». (Учебное пособие). - Т.: ТГЭУ, 2006.

Жуковская Ирина Евгеньевна в 1987 г. закончила Ташкентский институт народного хозяйства по специальности «Организация машинной обработки экономической информации», после работала старшим инженером учебно – методического отдела этого же института. Далее занималась педагогической деятельностью.

В настоящее время она старший преподаватель кафедры «Экономическая информатика» ТГЭУ. Является автором ряда научных статей.

Рецензент:
Доц. кафедры «Экономическая информатика» **Ш.Х. Хошимходжаев.**

ОГЛАВЛЕНИЕ

Тема 1. Объектно-ориентированное проектирование …….....6
Тема 2. Объектно-ориентированный подход в программировании…....…..12
Тема 3. Объектно-ориентированное мышление ……...….17
Тема 4. Borland Delphi …….….31
Тема 5. Среда программирования Delphi ……...44
Тема 6. Управление проектом в Delphi ……...60
Тема 7. Обзор Палитры компонент …….....73
Тема 8. Рисование и закраска …….....79
Тема 9. Печать текстовая и графическая ……84
Тема 10. Свойства в Delphi ……...……...88
Тема 11. Методы в Delphi ……...…....104
Тема 12. Обработка исключительных ситуаций в Delphi ……..……124
Тема 13. События в Delphi ……..…...148
Тема 14. Библиотека управления динамическим обменом данными (DDE) в Delphi …….....151
Тема 15. Использование DLL в Delphi ….....155
Литература ……..…….....176

ВВЕДЕНИЕ

О стремительности развития информационных технологий мы читаем и слышим почти каждый день. Темпы действительно впечатляют. Однако, в компьютерном мире остается наиболее важная область, изменения в которой происходят крайне медленно, например программирование, кодирование и составление исходных текстов ключевой элемент в создании любого приложения ныне происходят так же, как и 40 лет тому назад. Сегодня разработчик применяет крайне ограниченный набор логических конструкций (условный оператор и операторы присваивания и цикла) и небольшое число стандартных типов данных. Причем такой подход с тех пор ничуть не изменился, хотя сменилось уже не одно поколение языков программирования, и оно безусловно, уже давно превратилось из искусства в ремесло. Ныне в программировании всех интересуют прежде всего скорость и качество создания программ в коллективе, а эти характеристики может обеспечить только среда визуального проектирования, способная взять на себя значительные объемы рутинной работы по подготовке приложений, а также согласовывать деятельность группы постановщиков, кодировщиков, тестеров и технических писателей. Возможности ООП полностью отвечают подобным требованиям и подходят для создания систем любой сложности.

Объектно-ориентированное программирование существенно отличается от традиционного и считается трудно осваиваемым. Для того чтобы технологически грамотно использовать ООП, необходимо хорошо понимать его основные концепции и научиться мыслить при разработке программ в понятиях ООП. В данном учебном пособии как раз и представлены основные понятия ООП, а средой приложения выбрана среда Delphi.

Система Delphi позволяет писать как крохотные программы и утилиты, так и корпоративные системы, работающие с базами данных на разных платформах. В этой системе Delphi существуют сотни готовых компонентов, и при решении многих задач бывает полезно предварительно найти нужный компонент, вместо того чтобы выполнять работу по программированию, возможно, уже сделанную другими людьми. Таким образом, компонентный подход к созданию способствует повторному использованию готовых разработок. Delphi представляет следующие новые свойства и усовершенствования: в язык Object Pascal среды Delphi включены динамические массивы, методы обработки переполнения, установка значения параметров по умолчанию, и м.д. Менеджер проекта помогает объединять проекты, функционирующие вместе в одну проектную группу. Это позволяет организовать как работу взаимозависимых проектов, таких, например как одно и многозадачные приложения или DLL, так и совместную работу исполняемых программ.

Интегрированная среда разработки содержит более перенастраиваемую конфигурацию окон инструментов, которые можно закреплять с редактором кода. Проводник кода и менеджер проекта можно как закреплять, так и не закреплять. Интегрированный отладчик имеет множество свойств, включая

удаленную и многопроцессорную отладку, просмотр кода центрального процессора, инспекторов, усовершенствованные точки прерывания, отладчик специфических подменю и закрепленных окон. Иерархия объектов Delphi неизменно расширяется с тем, чтобы поддерживать drag-and-drop перетаскивания, обеспечивать дополнительный контроль над размещением окна, и м.д.[29].

Материала в данном учебном пособии изложение следующим образом.

Так, в первой главе представлены основные концепции и механизмы ООП. Описание выполнено сугубь теоретически, без учета особенностей конкретных реализаций объектных моделей на каком-либо языке программирования. При изучении данной главы весьма важно понять: чем рассматриваемый подход отличается от традиционного, что подразумевается под объектной декомпозицией, чем, в свою очередь, она отличается от процедурной и какие основные механизмы реализуются средствами объектно-ориентированных языков программирования.

Вторая глава содержит сведения о средствах объектно-ориентированного программирования в Delphi. Здесь представлены основные характеристики продукта Delphi, рассмотрены составляющие данной среды, Delphi, расписано управление пунктами меню Проекта Delphi, приведен постраничный обзор Палитры компонентов, уделено внимание работе с графическими компонентами а так же печати в текстовом и графическом режимах.

В третьей главе рассмотрены свойства, методы и события в Delphi.

Четвертая глава данного учебного пособия посвящена работе с библиотекой упраления динимическим обменом данными (DDE) в Delphi и вопросам использования DLL в Delphi.

В пятой главе уделено внимание вопросом структурной обработки исключительных ситуаций в Delphi, рассмотрена их модель и синтаксис, а также приведены конкретные примеры.

Этого этого, учебное пособие снабжено глоссарием, отражающим основные моменты технологии ООП.

Рекомендуется для студентов ВУЗов, а также всех желающих изучать ООП в среде Delphi.

Глава 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

1.1.Основные понятия объектно-ориентированного проектирования

Деловые компьютерные программы, используемые в бизнесе и научных исследованиях, строятся на основе моделей реального мира. В таких моделях реальным процессам и системам ставится в соответствие совокупность величин, называемых переменными состояния. Изменение состояния исследуемого процесса или системы отображается изменением переменных состояния модели. В общем случае математическая модель описывается набором переменных состояния и отношениями (связями) между этими переменными. Переменные состояния могут быть как числовыми, так и не числовыми, в том числе словами и предложениями естественного языка. Проектирование и разработка программ, реализующих модели сложных процессов и явлений, достаточно сложны и трудоемки. Одним из подходов, обеспечивающих структурирование математической модели и упрощение ее программирования, является объектный подход, в котором реальный процесс или система представляются совокупностью объектов, взаимодействующих друг с другом. Понятию “объект” сопоставляют ряд дополняющих друг друга определений. Ниже приводим некоторые из них:

- Объект - это осязаемая реальность, характеризующаяся четко определяемым поведением;

- Объект - особый опознаваемый предмет, блок или сущность (реальная или абстрактная), имеющая важное функциональное назначение в данной предметной области.

Объект может быть охарактеризован структурой, состоянием объекта, его поведением и индивидуальностью.

Состояние объекта определяется перечнем всех возможных (обычно статических) свойств и текущими значениями (обычно динамическими) каждого из этих свойств. Свойства объекта характеризуются значениями его параметров. Поведение объекта описывает, как воздействует на другие объекты или подвергается воздействию со стороны других объектов с точки зрения изменения его собственного состояния и состояния других объектов. Также говорят, что поведение объекта определяется его действиями.[26, 30]

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию именуют операцией. В объектно-ориентированных языках программирования операции называют методами. Выделяются пять типов операций:

- конструктор, создание и инициализация объекта;

- деструктор, разрушающий объект;

- модификатор, изменяющий состояние объекта;

- селектор для доступа к переменным объекта без их изменения;

- итератор для доступа к содержанию объекта по частям в определенной последовательности.

Известна и другая классификация методов объекта, когда выделяют функции управления, реализации, доступа, а также вспомогательные функции.[25]

Под индивидуальностью объекта понимают свойство объекта, позволяющее отличать этот объект от всех остальных.

Объекты могут находиться в определенных иерархическими отношениях друг к другу. Основные иерархические отношения - это отношения использования и включения.

Так, отношение использования реализуется посылкой сообщений от объекта А к объекту В. При этом объект А может выступать в роли:

- активного или воздействующего объекта, когда он воздействует на другие объекты, но сам воздействию не подвергается;

- пассивного или исполняющего, когда объект подвергается воздействию, но сам на другие объекты не воздействует;

- посредника, если объект и воздействует и сам подвергается воздействию.

Отношение включения имеет место тогда, когда составной объект содержит другие объекты.

Структура и поведение схожих объектов определяют их класс, между которыми также могут быть установлены следующие отношения:

- отношение разновидности (кошка - вид определенного биологического семейства или кошка - домашнее животное);

- включения или составной части (лапа - часть кошки);

- ассоциативности, когда между классами есть сгубо смысловая связь (кошки и собаки - домашние животные).

Объект, обладающий перечисленными характеристиками, в общем случае служит моделью реальной сущности, поскольку при его описании пренебрегают второстепенными или несущественными в конкретной ситуации свойствами.[25,26,27]

Выделение надлежащим образом совокупности объектов и отношений между ними позволяет построить объектную модель определенной предметной области, а на основе такой модели разработать программные средства для исследования этой предметной области и принятия решений.

Приведенная выше характеристика объектного подхода соответствует его применению для построения и программирования имитационных моделей реальных процессов и систем.

С точки зрения собственно программирования объектный подход можно рассматривать как развитие понятия типов данных. Тип данного определяет множество значений, которые может принимать данное, и набор операций, которые могут быть выполнены над данными этого конкретного типа. В языках программирования предусматриваются некоторые предопределенные (базовые) типы данных, обычно это целые и вещественные числовые типы, символьные и строковые типы, а в ряде случаев и преобразования данных одного типа в другой. В языках программирования со строгой типизацией каждое данное принадлежит только одному типу и разрешаются только явные преобразования

данных одного типа в другой. Строгая типизация повышает надежность разрабатываемых программ, позволяя выявить многие ошибки еще на этапе отладки программы. В языках программирования предусматривается и возможность построения новых типов данных, определяемых программистом, но для таких типов, как правило, отсутствует перечень допустимых операций.[30]

Объектно-ориентированные языки программирования позволяют распространить требования строгой типизации на типы данных, определяемых программистом.

Объектно-ориентированный подход к проектированию программных изделий предполагает:

- проведение объектно-ориентированного анализа предметной области;

- объектно-ориентированное проектирование;

- разработку программного изделия с использованием объектно-ориентированного языка программирования.

1.1.1. Объектно-ориентированный анализ

Объектно-ориентированный анализ (ООА) - это метод отождествления важных сущностей реального мира для понимания и объяснения того, как они взаимодействуют между собой. Говорят также, что ООА - это моделирование проблемы с целью формирования словаря предметной области, определения объектов и классов.

Известны несколько подходов к проведению ООА. Так, в работе С. Шлеер и С. Меллора "Объектно-ориентированный анализ: моделирование мира в состояниях" выделены три этапа ООА:[50]

- построение информационной модели, абстрагирование реальных сущностей в терминах объектов и атрибутов.

- построение модели состояний для формализации жизненных циклов объектов и отображение этой модели диаграммами и таблицами переходов, взаимодействие между объектами осуществляется путем передачи сообщений о происходящих с ними событиях.

- разработка модели процессов, в которой действия в моделях состояний расчленяются на фундаментальные и многократно используемые процессы.

В работе Г. Буча “Объектно-ориентированное проектирование с примерами применения” отмечаются следующиеальтернативные подходы к ООА:

- Метод неформального описания, в котором выделяются существительные и глаголы в описании предметной области.[27] Существительные здесь рассматриваются в качестве кандидатов для образования классов, а глаголы – как кандидаты в операции над классами;

- структурный анализ, при котором на основе модели системы, представленной диаграммами потоков данных, выделяются внешние события и объекты, база данных, поток управления, преобразования потока управления. Далее, на основе анализа потока данных и потока управления, выделяются классы и методы классов.

1.1.2. Пример объектно-ориентированного анализа

Пусть требуется разработать пакет программ для решения расчетных задач, настраиваемый на конкретную предметную область. Каждая предметная область определяется совокупностью переменных и связей между этими переменными - формулами и алгоритмами для вычисления некоторых переменных по известным значениям других переменных. Такой пакет должен состоять из двух частей, универсальной, обеспечивающей ввод и представления данных пользователям и общее управление работой пакета, а также функциональной части, содержащей алгоритмы вычислений для конкретной предметной области.[26,27,30]

Предметная область ППП для решения расчетных задач формально может быть описана совокупностью трех множеств:

- множества данных *X*;

- множества функциональных связей (задач, решаемых с использованием пакета) *F*;

- множества связей по определению *R*.

При использовании пакета пользователь может вводить некоторые данные из заранее определенного их набора и запрашивать решение одной или нескольких задач, а затем выводить вычисленные данные на экран, в файл или на принтер. Пакет должен настраиваться на конкретную предметную область путем определения множеств *X*, *F* и *R* и подключения соответствующего набора обрабатывающихся модулей (подпрограмм).

Объектно-ориентированный анализ проще всего начать с неформального описания и построения словаря предметной области. В данном случае такой словарь может включать следующие понятия:

Данные (множество *X*): множество данных, элемент множества данных.

Элемент множества данных должен описываться его типом, уровнем агрегирования и наличием или отсутствием конкретного значения;

Задачи (множество *F*): множество задач, элемент множества задач (конкретная задача), набор аргументов (входных и выходных данных задачи), аргумент задачи;

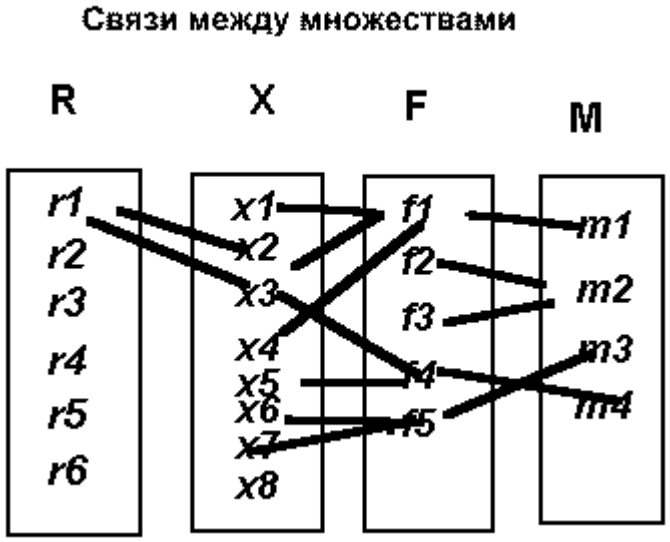
Связи по определению (множество *R*): множество связей, элемент множества связей, тип связи (старший - младший, функция-предикат);

Поскольку каждая задача должна реализовываться вызовом некоторого обрабатывающего модуля (подпрограммы), добавим соответствующие понятия.

Модули (множество *M*): множество модулей, элемент множества модулей, список параметров модуля, параметр модуля.

Элементы перечисленных множеств находятся между собой в определенных отношениях. Элемент множества задач использует элементы множества данных для передачи аргументов связанному с задачей модулю. Элементы множества связей либо связывают между собой пары данных, либо описывают некоторый предикат, аргументами которого служат элементы множества данных.

Кроме понятий - существительных следует выделить понятия - глаголы (действия), относящиеся к понятиям - существительным. В рассматриваемом примере действиями могут быть:



- определить новый элемент множества данных, задач или связей;

- добавить новый элемент в множество (данных, задач, связей);

- удалить элемент из множества (данных, задач, связей);

- ввести значение элемента множества данных;

- вывести на экран (в файл, на принтер) значение элемента множества данных;

- выполнить задачу.

Предполагается, что перечисленные действия должны выполняться по командам пользователя. Следовательно, все предыдущие рассуждения необходимоповторить для процесса диалога с пользователем. Отличие заключается лишь в том, что для организации диалога, как правило, применяются готовые инструментальные средства и свойственные им основные понятия: диалоговое окно, объект ввода строкового данного, независимые и зависимые, списки строк.

Результаты ООА документируются наборами диаграмм и таблиц, отражающих как структуру выделенных классов объектов, так и отношения между ними.

Понятия, которые принимаются как кандидаты в используемые классы, более подробно описываются по специальному шаблону, который включает следующие характеристики класса:

- имя класса, т.е. присвоенный ему идентификатор;

- множественность экземпляров класса (0/1/n);

- иерархия класса (базовые классы);

- структура и интерфейс класса.

Здесь под структурой понимаются как атрибуты класса (элементы-данные), так и действия (методы). Данные и методы разбиваются по уровням доступа.

Под интерфейсом класса понимаются элементы-данные, доступные из экземпляров других классов, и методы, которые могут быть вызваны из других классов.

Для классов желательно определить жизненные циклы экземпляров класса: когда, при каких условиях создается экземпляр класса, когда изменяется его состояние и когда он уничтожается.[28]

Поскольку в объектно-ориентированной системе экземпляры классов обмениваются сообщениями, для каждого класса следует определить поступающие его экземплярам сообщения и на их основе построить диаграммы перехода (описать класс как конечный автомат). В ряде случаев для каждого объекта целесообразно построить модели состояний и определить списки событий, изменяющих состояние объектов.

После выделения классов и их неформального описания могут быть построены модели процессов, которые должны быть реализованы в будущем программном изделии. В такой модели отражаются внешние события (действия пользователя) и вызываемые этими событиями действия с экземплярами классов.

Например, в рассматриваемом примере процесс определения нового данного должен включать ввод имени данного и его описания, включающего тип данного, уровень агрегирования, границы индексов для массива. После ввода пользователем имени данного нужно проверить, является ли это имя уникальным.[27,28]

После ввода границ индекса для массива также нужно проверить их допустимость. Если пользователь указал все характеристики данного правильно, нужно построить экземпляр объекта “данное” и включить его в множество данных.

1.1.3. Процесс объектно-ориентированного проектирования

Объектно-ориентированное проектирование (Object-Oriented Design - OOD) - это поступательный итеративный процесс. Граница между объектно-ориентированным анализом и проектированием расплывчата и построение проекта программного изделия состоит из ряда циклов, в которых уточняются описания классов и взаимодействия между ними, разрабатываются реализующие их программы, проводится их отладка и тестирование и по результатам каждого этапа уточняются рабочие документы предыдущих этапов, дорабатываются описания классов и программы. Эти циклы повторяются до получения требуемого результата.

В рассмотренном выше примере были выделены классы “множество данных” и “данное”. Пусть классу “множество данных” присвоено имя TXSet.

С учетом имеющихся инструментальных средств класс TXSet может быть построен на основе класса Array из библиотеки CLASSLIB, т.е. это множество может быть интерпретировано массивом. Массив представляет собой упорядоченную совокупность однотипных элементов, в то же время данные могут принадлежать различным типам и каждому типу соответствует свой набор характеристик. Это противоречие можно преодолеть, если элементами массива TXSet будут указатели на экземпляры данных.

Чтобы использовать указатели на экземпляры данных как элементы массива, все классы, определяющие типы данных, должны быть образованы из общего базового класса.[27,28]

Пусть требуется обеспечить возможность использования числовых скалярных данных и массивов (векторов и прямоугольных матриц), а также данных типа строк и массива строк. Естественно определить для каждого такого типа свой класс: TDScal, TDArray, TDString, TDStringArray. В каждом из этих классов должно быть поле идентификатора данного ident, поле описания данного head и, возможно, поле flags, представляющее собой набор битов, дополняющих описание данного. Может оказаться удобным иметь и поля, содержащие количество знаков при представлении скаляра или элементов массивов (width) и количество цифр в дробной части для представления чисел (dec). Все эти данные можно объединить в классе TData, базовом для остальных классов данных. Таким образом, вместо одного класса “данное”, выделенного на этапе анализа, появилось пять классов. После этого следует вернуться к этапу анализа и оформить рабочие документы анализа для новых классов.

Аналогичным образом следует уточнить состав и определения остальных классов, выбранных на этапе анализа.

После определения перечня классов следует разработать семантику каждого класса - установить состав и назначение методов класса. При этом также может возникнуть необходимость выделения новых классов и, следовательно, повторение отдельных частей этапа анализа и новое уточнение ранее описанных классов.

Таким образом, процесс объектно-ориентированного проектирования состоит из циклического выполнения четырех основных шагов:[26,27]

- определение классов и объектов на том или ином уровне абстракции;

- выявление семантики классов;

- установление (идентификация) связей между классами и объектами;

- Реализация классов.

На каждом повторении этого цикла уточняются описания классов и перерабатываются проектные документы.

1.2. Объектно-ориентированный подход в программировании

1.2.1. Технологии программирования

Технология программирования - это совокупность методов и средств разработки (написания) программ, а также порядок применения этих методов и средств.

На ранних этапах развития программирования, когда программы писались в виде последовательностей машинных команд, какая-либо технология программирования отсутствовала. Первые шаги в ее разработке состояли в представлении программы в виде последовательности операторов. Написанию последовательности машинных команд предшествовало составление операторной схемы, отражающей последовательность операторов и переходы между ними. Операторный подход позволил разработать первые программы для автоматизации составления программ - так называемые составляющие программы.

С увеличением размеров программ стали выделять их обособленные части и оформлять их как подпрограммы. Часть таких подпрограмм объединялась в библиотеки, из которых подпрограммы можно было включать в рабочие программы, а затем из них вызывать, что и положило начало процедурному программированию - большая программа представлялась совокупностью процедур-подпрограмм. Одна из подпрограмм являлась главной и с нее начиналось выполнение программы.

В 1958 г. были разработаны первые языки программирования, Фортран и Алгол-58. Программа на Фортране состояла из главной программы и некоторого количества процедур - подпрограмм и функций. Программа на Алголе-58 и его последующей версии Алголе-60 представляла собой единое целое, но имела блочную структуру, включающую главный блок и вложенные блоки подпрограмм и функций. Компиляторы для Фортрана обеспечивали раздельную трансляцию процедур и последующее их объединение в рабочую программу, первые компиляторы для Алгола предполагали, что транслируется сразу вся программа, раздельная трансляция процедур не обеспечивалась.[51]

Процедурный подход потребовал структурирования будущей программы, ее разделения на отдельные процедуры. При разработке отдельной процедуры о других процедурах требовалось знать только их назначение и способ вызова. Появилась возможность перерабатывать отдельные процедуры, не затрагивая остальной части программы, сокращая при этом затраты труда и машинного времени на разработку и модернизацию программ.[21]

Следующим шагом в углублении структурирования программ стало так называемое структурное программирование, при котором программа в целом и отдельные процедуры рассматривались как последовательности канонических структур: линейных участков, циклов и разветвлений. Появилась возможность читать и проверять программу как последовательный текст, что повысило производительность труда программистов при разработке и отладке программ. С целью повышения структурности программы были выдвинуты требования к большей независимости подпрограмм, подпрограммы должны связываться с вызывающими их программами только путем передачи им аргументов, использование в подпрограммах переменных, принадлежащих другим процедурам или главной программе, стало считаться нежелательным.[53]

Процедурное и структурное программирование затронули, прежде всего, процесс описания алгоритма как последовательности шагов, ведущих от варьируемых исходных данных к искомому результату. Для решения

специальных задач стали разрабатываться языки программирования, ориентированные на конкретный класс задач: системы управления базами данных, имитационное моделирование и т.д.

При разработке трансляторов все больше внимания стало уделяться обнаружению ошибок в исходных текстах программ, обеспечивая, этим, сокращение затрат времени на отладку программ.[21,22]

Применение программ в самых разных областях человеческой деятельности привело к необходимости повышения надежности всего программного обеспечения. Одним из направлений совершенствования языков программирования стало повышение уровня типизации данных. Теория типов данных исходит из того, что каждое используемое в программе данное принадлежит одному и только одному типу данных. Тип данного определяет множество возможных значений данного и набор операций, допустимых над этим данным. Данное конкретного типа в ряде случаев может быть преобразовано в данное другого типа, но такое преобразование должно быть явно представлено в программе. В зависимости от степени выполнения перечисленных требований, можно говорить об уровне типизации того или иного языка программирования. Стремление повысить уровень типизации языка программирования привело к появлению языка Паскаль, который считается строго типизированным, хотя и в нем разрешены некоторые неявные преобразования типов, например, целого в вещественное. Применение строго типизированного языка при написании программы позволяет еще при трансляции исходного текста выявить многие ошибки использования данных и этим, повысить надежность программы. Вместе с тем строгая типизация сковывала свободу программиста, затрудняла применение некоторых приемов преобразования данных, часто используемых в системном программировании. Практически одновременно с Паскалем был разработан язык Си, в большей степени ориентированный на системное программирование и относящийся к слабо типизированным языкам.[26]

Все универсальные языки программирования, несмотря на различия в синтаксисе и используемых ключевых словах, реализуют одни и те же канонические структуры: операторы присваивания, циклы и разветвления. Во всех современных языках присутствуют предопределенные (базовые) типы данных (целые и вещественные арифметические типы, символьный и, возможно, строковый тип), имеется возможность использования агрегатов данных, в том числе массивов и структур (записей). Для арифметических данных разрешены обычные арифметические операции, для агрегатов данных как правило предусмотрена только операция присваивания и возможность обращения к элементам агрегата. Вместе с тем при разработке программы для решения конкретной прикладной задачи желательна возможно большая концептуальная близость текста программы к описанию задачи. Например, если решение задачи требует выполнения операций над комплексными числами или квадратными матрицами, желательно, чтобы в программе явно присутствовали операторы сложения, вычитания, умножения и деления данных типа комплексного числа, сложения, вычитания, умножения и обращения данных

типа квадратной матрицы. Решение этой проблемы возможно несколькими путями:

- построением языка программирования, содержащего как можно больше типов данных, и выбором для каждого класса задач некоторого подмножества этого языка. Такой язык иногда называют языком-оболочкой. На роль языка-оболочки претендовал язык ПЛ/1, оказавшийся настолько сложным, что так и не удалось построить его формализованное описание. Отсутствие формализованного описания, однако, не помешало широкому применению ПЛ/1 как в Западной Европе, так и в бывшем СССР;[52]

- построением расширяемого языка, содержащего небольшое ядро и допускающего расширение, дополняющее язык типами данных и операторами, отражающими концептуальную сущность конкретного класса задач. Такой язык называют языком-ядром. Как язык-ядро были разработаны языки Симула и Алгол-68, не получившие широкого распространения, но оказавшие большое влияние на разработку других языков программирования.

Дальнейшим развитием второго пути явился объектно-ориентированный подход к программированию, рассматриваемый в следующем параграфе.

1.2.2. Сущность объектно-ориентированного подхода к программированию

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

- программа представляет собой модель некоторого реального процесса, части реального мира;
- модель реального мира или его части может быть описана как совокупность взаимодействующих между собой объектов;
- объект описывается набором параметров, значения которых определяют состояние объекта, и набором операций (действий), которые может выполнять объект;
- взаимодействие между объектами осуществляется посылкой специальных сообщений от одного объекта к другому. Сообщение, полученное объектом, может потребовать выполнения определенных действий, например, изменения состояния объекта;
- объекты, описанные одним и тем же набором параметров и способные выполнять один и тот же набор действий, представляют собой класс однотипных объектов.[25,26,27]

С точки зрения языка программирования, класс объектов можно рассматривать как тип данного, а отдельный объект - как данное этого типа. Определение программистом собственных классов объектов для конкретного набора задач должно позволить описывать отдельные задачи в терминах самого класса задач (при соответствующем выборе имен типов и имен объектов, их параметров и выполняемых действий).

Таким образом, объектно-ориентированный подход предполагает, что при разработке программы должны быть определены классы используемых в

программе объектов и построены их описания, затем созданы экземпляры необходимых объектов и определено взаимодействие между ними.

Классы объектов часто удобно строить так, чтобы они образовывали иерархическую структуру. Например, класс “Студент”, описывающий абстрактного студента, может служить основой для построения классов “Студент 1 курса”, “Студент 2 курса” и т.д., которые обладают всеми свойствами студента вообще и некоторыми дополнительными свойствами, характеризующими студента конкретного курса. При разработке интерфейса с пользователем программы могут использовать объекты общего класса “Окно” и объекты классов специальных окон, например, окон информационных сообщений, окон ввода данных и т.п. В таких иерархических структурах один класс может рассматриваться как базовый для других, производных от него, классов. Объект производного класса обладает всеми свойствами базового класса и некоторыми собственными свойствами, он может реагировать на те же типы сообщений от других объектов, что и объект базового класса, а также на сообщения, имеющие смысл только для производного класса. Обычно говорят, что объект производного класса наследует все свойства своего базового класса.

Некоторые параметры объекта могут быть локализованы внутри объекта и недоступны для прямого воздействия извне объекта. Например, во время движения объекта-автомобиля объект-водитель может воздействовать только на ограниченный набор органов управления (рулевое колесо, педали газа, сцепления и тормоза, рычаг переключения передач) и ему недоступен целый ряд параметров, характеризующих состояние двигателя и автомобиля в целом.[30]

Очевидно для того, чтобы продуктивно применять объектный подход для разработки программ, необходимы языки программирования, поддерживающие этот подход, т.е. позволяющие строить описание классов объектов, образовывать данные объектных типов, выполнять операции над объектами. Одним из первых таких языков стал язык SmallTalk, в котором все данные являются объектами некоторых классов, а общая система классов строится как иерархическая структура на основе предопределенных базовых классов.

Опыт программирования свидетельствует о том, что любой методический подход в технологии программирования не должен применяться слепо с игнорированием других подходов. Это относится и к объектно-ориентированному подходу. Существует ряд типовых проблем, для которых его полезность наиболее очевидна, к таким проблемам, в частности, относятся, задачи имитационного моделирования, программирование диалогов с пользователем.[21] Существуют и задачи, в которых применение объектного подхода ни к чему, кроме излишних затрат труда, не приведет. В связи с этим наибольшее распространение получили объектно-ориентированные языки программирования, позволяющие сочетать объектный подход с другими методологиями. В некоторых языках и системах программирования применение объектного подхода ограничивается средствами интерфейса с пользователем (например, Visual FoxPro ранних версий).

Наиболее используемыми в настоящее время объектно-ориентированными языками являются Паскаль с объектами, Delphi и Си++, причем наиболее развитые средства для работы с объектами содержатся в Си++.

Практически все объектно-ориентированные языки программирования являются развивающимися языками, их стандарты регулярно уточняются и расширяются. Следствием этого развития являются неизбежные различия во входных языках компиляторов разных систем программирования. Наиболее распространенны в настоящее время это системы программирования Microsoft C++ , Microsoft Visual C++ и системы программирования фирмы Borland International.

1.3. Объектно-ориентированное мышление и принципы объектно-ориентированного подхода в программировании

1.3.1. Объектно-ориентированное мышление

Объектно-ориентированное программирование (ООП) - основная методология программирования 1990-х годов. Она является продуктом 25 летней практики и включает ряд языков: Simula 67, Smalltalk, Lisp, Clu, Actor, Eiffel, Objective C, Delphi, C++. Это стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты.

Почему ООП так популярно:

- надежда, что ООП может просто и быстро привести к росту продуктивности и улучшению надежности программ, помогая, тем самым, разрешить кризис в программном обеспечении;
- желание перейти от существующих языков программирования к новой технологии;
- вдохновляющее сходство с идеями, родившимися в других областях.

ООП является лишь последним звеном в длиной цепи решений, которые были предложены для разрешения "кризиса программного обеспечения". Кризис программного обеспечения означает, что те задачи, которые мы хотим решить, опережают наши возможности.

Несмотря на то, что ООП действительно помогает в проектировании сложных и больших программ, ООП не панацея. Чтобы стать профессионалом в программировании, необходимы талант, способность к творчеству, интеллект, знания, логика, умение строить и использовать абстракции и опыт, даже если используются лучшие средства разработки.

ООП является новым пониманием того, что называется вычислениями. Чтобы стать профессионалом в ООП, недостаточно просто добавить новые знания, необходима полная переоценка привычных методов разработки программ.[50,52]

Язык и мышление.

Язык, на котором мы говорим, непосредственно влияет на способ восприятия мира. Это касается не только естественных языков, но и для искусственных языков - языков программирования. Чтобы эффективно

использовать ООП, требуется глядеть на мир иным способом, не как с точки зрения структурных языков. Само по себе использование C++ или Delphi не делает программу ООП. "Программа фортрановского типа " может быть написана на любом языке".

Гипотеза Сапиро-Ворфа утверждает, что индивидуум, использующий некоторый язык, в состоянии вообразить или придумать нечто, что не может быть переведенным или даже понятым индивидуумом из другой языковой среды. Существует и прямо противоположная концепция: **принцип Черча:** *Любое вычисление, для которого существует эффективная процедура, может быть реализовано на машине Тьюринга.*

Это утверждение недоказуемо, поскольку не имеется строго определения "эффективная процедура". Но не найдено опровержения этого принципа.

В 1960-х годах было показано, что машина Тьюринга может быть смоделирована на любом языке, где есть условный оператор и оператор циклов. (Отсюда создалось утверждение, что оператор goto не нужен). Принцип Черча утверждает, что по своей сути все языки программирования идентичны, но объектно-ориентированный подход решает проблему проще, приближая решение к естественному восприятию.

Новая парадигма

ООП часто называют новой *парадигмой* программирования. Другие парадигмы: директивная (структурное программирование - Pascal, C) и логическая - Prolog, функциональное - Lisp, Effel. Парадигмы в программировании определяют как проводить вычисления, как работа, выполняемая компьютером, должна быть структурирована и организована.

Новички в информатике часто могут освоить парадигму лучше, чем опытные профессионалы, так как этот способ решения задач ближе к естественному восприятию.[26]

Способ видения мира.

Рассмотрим ситуацию из обыденной жизни. Например, Вам надо сообщить поздравить своего родственника, живущего в другом городе с днем рождения. Для это Вы идете на почту и посылаете телеграмму. Вы сообщаете оператору, что хотите переслать данный текст по некоторому адресу. И Вы можете быть уверены, что ваше поздравление попадет по нужному адресу.

Итак, для решения своей проблемы Вы нашли *агента* (почту) и передали ему *сообщение*, содержащее запрос. Обязанностью почты (или работников почты) будет удовлетворить Ваш запрос любым известным только им способом. Вас совершенно не интересует каким именно. Есть некий *метод* - т.е. алгоритм или последовательность операций, которые используют почтовые работники для выполнения запроса. Вам не надо знать какой конкретно метод используется.

Итак, **первым принципом объектно-ориентированного подхода** к решению задачи является способ задания действий.[27,52]

Действие в ООП инициируется посредством передачи сообщений агенту (объекту), ответственному за действия. Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией

(аргументами), необходимой для его выполнения. Получатель - это агент, посылается сообщение. Если он принимает сообщение, то на него автоматически возлагается ответственность за выполнение указанного действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.

Сокрытие информации является важным принципом и в традиционных языках программирования. Чем же пересылка сообщения отличается в ООП и в традиционных подходах?

Первое отличие состоит в том, что имеется вполне определенный *получатель* - агент, которому послано сообщение. При вызове процедуры нет столь явно выделенного получателя. [53]

Второе отличие состоит в том, что *интерпретация* сообщения (а именно метод, вызываемый после приема сообщения) зависит от получателя и является различной для разных получателей. Вы можете попросить своего товарища, летящего в город, где живут ваши родственники, поздравить их, и метод, который он изберет для решения этого запроса будет отличаться от того, который использовали на почте. Хотя родственники будут поздравлены. Если же Вы попросите коменданта общежития поздравить Ваших родственников, то у нее, вероятно, вообще не найдется метода для решения этой задачи, а если она и примет сообщение, то выдаст диагностическое сообщение об ошибке.[27]

Таким образом, различие между вызовом процедуры и пересылкой сообщения заключается в том, что в последнем случае существуют определенные получатель и интерпретация (т.е. выбор подходящего метода, который запускается в ответ на сообщение) разные для разных получателей. Обычно конкретный получатель неизвестен вплоть до выполнения программы, и, следовательно, неизвестен и метод какой будет вызван. В таком случае говорят, что имеет место позднее связывание между сообщением и методом, который вызывается в ответ на данное сообщение. Эта ситуация противопоставляется раннему связыванию на стадиях компоновки, присущих традиционному программированию.

Фундаментальной концепцией в ООП является понятие обязанности или *ответственности* за выполнение действий. Ваш запрос на поздравление родственников выражает лишь желаемый результат. Почта вольна сама выбирать средство для удовлетворения этого запроса. Полный набор обязанностей, связанных с определенным объектом называют протоколом.

Различие между взглядом структурного подхода и ООП можно выразить как *Задавайтесь вопросом не о том, что Вы можете сделать для своих структур, а о том, что структуры данных могут сделать для Вас.*

Мы можем рассматривать почту как некоторое средство связи. Объединим в средства связи все, что позволяет связываться. Эта операция выступает как **второй принцип ООП:**

Все объекты являются представителями, или экземплярами, классов. Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты

одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Рис.1. Весь мир - это объекты, которые передают друг другу сообщения.

О почте Вы знаете больше, чем то, что нужно, чтобы сделать запрос. Вы знаете, что у Вас попросят деньги, что Вам выдадут квитанцию. Все это справедливо и для магазинов, ресторанов. Поскольку категория Post более узкая, чем Service, то любое знание, которым Вы обладаете для категории Service, будет справедливо и для Post.[26,27]

Работников почты можно представить в виде, например, такой иерархии категорий: работник почты - это продавец услуг, продавец услуг - это просто продавец, продавец - это человек, человек - это млекопитающее, млекопитающее - это животное, животное - это материальные объекты.

Принцип, в соответствии с которым знание о более общей категории разрешается использовать для более узкой категории, называется *наследованием.*

Классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс (или подкласс) наследует атрибуты родительского класса (или надкласса), расположенного выше в иерархическом дереве.

Утконос представляет проблему для структуры, изображенной на рис. 2. Утконос млекопитающее, но откладывает яйца, следовательно, важно переопределить способ его рождения. Таким образом, необходимо разрешать переопределять информацию, наследуемую из родительских классов. Для поиска метода, подходящего для обработки сообщения, используется следующее правило. [27,51]

Поиск метода, который вызывается в ответ на определенное сообщение, начинается с методов, принадлежащих классу получателя. Если подходящий метод не найден, то поиск продолжается для родительского класса. Поиск продвигается вверх по цепочке родительских классов до тех пор, пока не будет найден нужный метод или пока не будет исчерпана последовательность родительских классов. В первом случае выполняется найденный метод, во втором - выдается сообщение об ошибке. Если выше в иерархии классов

существуют методы с тем же именем, что и текущий, то говорят, что данный метод переопределяет наследуемое поведение.

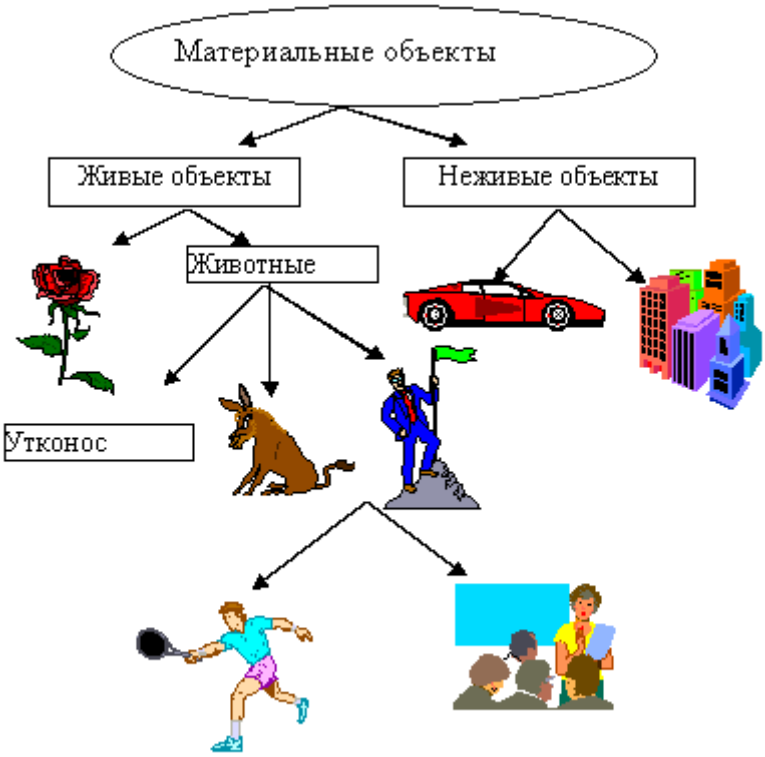


Рис. 2. Иерархическое дерево.

Тот факт, что Ваш друг-студент и оператор почты по-разному будут реагировать на просьбу поздравить родственников, является признаком *полиморфизма*.

Характеристики ООП

1. Все является объектом.

2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщения - это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

3. Каждый объект имеет независимую память, которая состоит из других объектов.

4. Каждый объект является представителем класса, который выражает общие свойства объектов.

5. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанные с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Вычисление и моделирование.

Традиционная модель, описывающая выполнение программы на компьютере, базируется на дуализме - *процесс-состояние*. С этой точки зрения компьютер является администратором данных, следующим некоторому набору инструкций. В рамках ООП мы никогда не поминаем адреса ячеек памяти, переменные, присваивания.

Вместо процессора, который перемалывает биты и шарит по карманам структур данных, мы получаем вселенную благовоспитанных объектов, которые любезно просят друг друга о выполнении тех или иных своих желаний. [50]

При ООП мы считаем, что *вычисление есть моделирование*.

Конечно, объекты не могут во всех случаях реагировать на сообщение только тем, что вежливо обращаются друг к другу с просьбой выполнить некоторое действие. Это приведет к бесконечному циклу запросов, аналогично тому, как два джентльмена так и не вошли в дверь, уступая друг другу дорогу. На определенной стадии объекты должны выполнять некоторые действия перед пересылкой запросов другим объектам.

Сложность

На заре информатики большинство программ писалось на ассемблере, затем, с увеличением сложности задач, стало невозможным помнить обо всех состояниях регистров, стеков и т.д. Появились такие языки, как Fortran и Cobol. По мере дальнейшего развития методов программирования было замечено, что ту задачу, которую программист решает за один месяц, два программиста не решают за один месяц.

При традиционном подходе сложность программы усложняется большим числом перекрестных ссылок, которые ссылки обозначают зависимость одного фрагмента кода от другого. Даже самый независимый фрагмент кода часто невозможно понять в изоляции от других.

Главный механизм борьбы со сложностью - это *абстрагирование*, т.е. способность отделить логический смысл фрагмента программы от проблемы его реализации: от процедур - к модулям, далее к абстрактным данным и, наконец, к объектам.

Процедуры и функции были первыми механизмами абстрагирования, использованными в языках программирования. Процедуры позволяют сконцентрировать в одном месте работу, которая многократно выполняемую, а затем многократно использовать этот код. Кроме того, данные процедуры обеспечили возможность *маскировки информации*. Программист мог написать процедуру, которуе используют другие программисты, не вдаваясь в детали. Но они не решали всех проблем: нет эффективного механизма маскировки деталей организации данных.

Рассмотрим пример организации стека.

Листинг 1.1.

```
Void init(){datatop=0;}
int top()
{ if(datatop > 0)
return datastack[datatop-1];
return 0;}
```

```
int datastack[100];
int datatop=0;void push(int val)
{ if (datatop < 100)
datastack[datatop++]=val; }
int pop()
{ if(datatop > 0)
return datastack[--datatop];
return 0; }
```

Отсюда явствует, что данные не могут быть локальными, потому что являются общими для всех процедур. Данные должны содержаться в глобальных переменных. Однако, если переменные глобальные, то нет способа ограничить доступ к ним или их видимость. Если имя переменной datastack, то об этом должны знать все программисты, чтобы не создать переменные с таким именем, поскольку онизапрещены, даже если переменные с таким именем больше нигде не используются.

Модули можно рассматривать как улучшенный метод создания и управления совокупности имен и связанными с ними значениями. Если рассматривать модуль как абстрактную концепцию, то ее суть состоит в разбиении пространства имен на две части. Так, открытая (public) часть является доступной извне модуля, закрытая (private) - доступна только внутри модуля. Типы, данные и процедуры могут быть применимы к любой из этих двух частей.

1. Пользователя, который намерен использовать модуль, следует снабдить всей полнотой информации, необходимой для того, чтобы делать это корректно, и не *более того*.

2. Разработчика следует снабдить всей полнотой информацией, необходимой для создания модуля, и не *более того*.

Модули решают некоторые, но не все проблемы разработки программного обеспечения, если например, захочется иметь два экземпляра стека. Механизм модуля, позволяя маскировать данные, не дает возможность размножать экземпляры.[26,27,30,51]

Абстрактный тип данных задается программистом, С которыми можно манипулировать так же, как и с данными типов, встроенных в систему. Пользователю разрешается создавать переменные, которые принимают значения из допустимого множества, и манипулировать ими, используя имеющиеся операции. Чтобы построить абстрактный тип данных, мы должны:

1. Экспортировать определение типа данных;

2. Делать доступным набор операций, использующихся для манипулирования экземплярами типа данных;

3. Защищать данные, связанные с типом данных, чтобы с ними можно было работать только через указанные подпрограммы;

4. Создавать несколько экземпляров абстрактного типа данных.

Объекты - это почти абстрактный тип данных, но дополненный некоторыми новшествами.

ООП добавляет идеи к абстрактному типу данных. Главная из них - пересылка сообщений. Действие инициируется по *запросу,* обращенному к конкретному объекту, а не через вызов функции.

К пересылке сообщения добавляется переопределение имен и совместного многократного использования кода.

Интерпретация сообщений меняется для разных объектов. Так, добавляются механизмы *наследования* и *полиморфизма.* Наследование позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности. Полиморфизм обеспечивает, чтобы общий код удовлетворял конкретным особенностям отдельных типов данных.

Люди строят дома, машины, самолеты, собирая их из отдельных деталей, и не изготовляя заново для каждого отдельного случая. Можно ли сконструировать ПО таким же образом?

Многократное использование - это цель, к которой все стремятся, но редко достигают. ООП обеспечивает механизм для отделения существенной информации от специализированной (например, конкретный формат данных).[27]

Подводя итог выше сказанному, следует отметить, что ООП - это не просто набор некоторых свойств, добавленных в уже существующие языки. Это новый шаг в *осмыслении* задач и разработки ПО.

Программы - это совокупность взаимодействующих объектов. Каждый объект отвечает за конкретную задачу. Вычисление осуществляется посредством взаимодействия объектов. Объект получается в результате инкапсуляции состояния (данных) и поведения (операций), во многом аналогично абстрактному типу данных - АТД. Поведение объекта диктуется классом. Данные и поведение представлены в виде классов, экземпляры которых - объекты. Все экземпляры одного класса будут вести себя одинаковым образом в ответ на одинаковые запросы.

Объект проявляет свое поведение путем вызова метода в ответ на сообщение. Интерпретация сообщения зависит от объекта и может быть различной для разных классов объектов.

Для удобства создания нового типа из уже существующих типов, определенных пользователем используется механизм *наследования*. Классы могут быть организованы в виде иерархического дерева наследования.

С помощью уменьшения взаимозависимости ООП позволяет разрабатывать системы, пригодные для многократного использования.

ООП - это взгляд на программирование, сосредоточенный на данных; в котором данные и поведение жестко связаны. Для этого необходимо, чтобы объекты определялись вместе с сообщениями, на которые они могут реагировать.

Объектно-ориентированная парадигма предлагает новый подход к разработке программного обеспечения. Фундаментальная концепция объектно-

ориентированной парадигмы состоит в *передаче сообщений объектам*. [26,27,50,52]

Таким образом, объектно-ориентированный язык должен обладать свойствами *абстракции, инкапсуляции, наследования и полиморфизма.*

1. Инкапсуляция с сокрытием данных - способность отличать внутреннее состояние объекта и поведение от его внешнего состояния и поведения;
2. Абстракция - расширяемость типов - способность добавляять типы, определяемые пользователем для того, чтобы дополнить ими встроенные типы. Один из принципов ООП заключается в том, чтобы типы, определяемые пользователем, должны обладать теми же привилегиями, что и встроенные типы;
3. Наследование - способность создавать новые типы, повторно используя, описание существующих типов;
4. Полиморфизм с динамическим (поздним) связыванием - способность объектов быть ответственными за интерпретацию вызова функции.

1.3.2. Принципы объектно-ориентированного подхода

1. Действие в объектно-ориентированном программировании инициируется посредством передачи сообщений объекту. Сообщение содержит запрос на осуществление действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.
2. Все объекты являются экземплярами, классов. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.
3. Принцип наследования. Классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс наследует атрибуты родительского класса.

Иерархия – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем.

В ООП используется два вида иерархии: иерархия *«целое/часть»* и иерархия *«общее/частное».*

Иерархия «целое/часть» показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне – при декомпозиции предметной области на объекты, на физическом уровне – при декомпозиции системы на модули и при выделении отдельных процессов в мультипроцессной системе).

Иерархия «общее/частное» показывает, что некоторая абстракция является частным случаем другой абстракции, например, «письменный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Один из важнейших механизмов ООП – наследование свойств в иерархии *общее/частное*. *Наследование* – такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).[30]

4. Принцип полиморфизма. Объекты реагируют на одно и тоже сообщение строго специфичным для них образом.

1.3.3. Языковые средства объектно-ориентированного программирования

На сегодняшний день существует множество сред программирования, предусматривающих объектную модель. Самая простая объектная модель использована при разработке Borland Pascal 7.0. Она специально создавалась для облегчения перехода программистов на использование технологии ООП и не поддерживает абстракции метаклассов, почти не содержит специальных средств сокрытия реализации объектов, но даже в таком варианте позволяет создавать достаточно сложные системы. Объектные модели остальных языков являются практически полными.

В таб. 1 представлены сравнительные характеристики моделей ООП в некоторых средах программирования.

Таблица 1.

Сравнительные характеристики моделей ООП в некоторых средах программирования

Характеристики	**Среды программирования**			
	Borland Pascal 7.0.	Borland C++ 3.1	Delphi	C++ Builder
Абстракции: объекты классы	 да да	 да да	 да да	 да да
Ограничение доступа: механизм сокрытия деталей реализации	внутри модуля	внутри класса и /или потомков класса	внутри модуля и потомков класса	внутри класса и /или потомков класса
обеспечение интерфейса к	нет	нет	да	да

	Среды программирования			
полям объекта				
Модульность: интерфейс реализация	да да	да (файл – заголовок) да (файл)	да да	да (файл-заголовок) да (файл)
Иерархичность: «целое – часть» «общее – частное» наследование	да да простое	да да множественное	да да простое	да да множест-венное
Типизация: степень позднее связывание шаблоны метаклассы	строгое да нет нет	среднее да да нет	строгое да нет да	среднее да да нет
Параллелизм разделение времени	моделируется	моделируется	Обеспечива-ется Windows	Обеспечива-ется Windows
Устойчивость объектов	определяется временем жизни переменной			

Как видно из таблицы особое место занимают объектные модели Delphi и C++Builder. Эти модели обобщают опыт ООП для MS DOS и включают некоторые новые средства, обеспечивающие эффективное создание более сложных систем. [30] На базе этих моделей созданы визуальные среды для разработки приложений Windows. Сложность программирования под Windows удалось существенно снизить за счет создания специальных библиотек объектов, «спрятавших» многие элементы техники программирования.

Краткие выводы

- Объект - это осязаемая реальность, характеризующаяся четко определяемым поведением.

- Объект - особый опознаваемый предмет, блок или сущность (реальная или абстрактная), имеющий важное функциональное назначение в данной предметной области.

Объект может быть охарактеризован структурой, состоянием объекта, его поведением и индивидуальностью.

В объектно-ориентированных языках программирования операции называют методами. Можно выделить пять типов операций:

- конструктор, создание и инициализация объекта;

- деструктор, разрушающий объект;

- модификатор, изменяющий состояние объекта;

- селектор для доступа к переменным объекта без их изменения;

- итератор для доступа к содержанию объекта по частям в определенной последовательности.

Известна и другая классификация методов объекта, когда выделяют функции управления, реализации, доступа и вспомогательные функции.

Технология программирования - это совокупность методов и средств разработки (написания) программ и порядок применения этих методов и средств.

Опыт программирования показывает, что любой методический подход в технологии программирования не должен применяться слепо с игнорированием других подходов.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

- программа представляет собой модель некоторого реального процесса, части реального мира;

- модель реального мира или его части может быть описана как совокупность взаимодействующих между собой объектов;

- объект описывается набором параметров, значения которых определяют состояние объекта, и набором операций (действий), которые может выполнять объект;

- взаимодействие между объектами осуществляется посылкой специальных сообщений от одного объекта к другому. Сообщение, полученное объектом, может потребовать выполнения определенных действий, например, изменения состояния объекта;

– Объекты, описанные одним и тем же набором параметров и способные выполнять один и тот же набор действий, представляют собой класс однотипных объектов.

Мышление **–** высшая ступень человеческого познания, процесса отображения объективной действительности. Позволяет получать знания о таких объектах, свойствах и отношениях реального мира, которые не могут быть непосредственно восприняты на чувственной ступени познания.

ООП часто называют **новой *парадигмой* программирования.** Другие парадигмы: директивная (структурное программирование - Pascal, C++), логическая - Prolog, функциональные - Lisp, Effel. Парадигмы в

программировании определяют, как проводить вычисления, как работа, выполняемая компьютером, должна быть структурирована и организована.

Новички в информатике часто могут освоить парадигму лучше, чем опытные профессионалы, так как этот способ решения задач ближе к естественному восприятию.

Характеристики ООП

1. Все является объектом.

2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщения - это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.

3. Каждый объект имеет независимую память, которая состоит из других объектов.

4. Каждый объект является представителем класса, который выражает общие свойства объектов.

5. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанные с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Объектно-ориентированная парадигма предлагает новый подход к разработке программного обеспечения. Фундаментальная концепция объектно-ориентированной парадигмы состоит в *передаче сообщений объектам*.

Таким образом, объектно-ориентированный язык должен обладать свойствами *абстракции, инкапсуляции, наследования и полиморфизма.*

1. Инкапсуляция с сокрытием данных - способность отличать внутреннее состояние объекта и поведение от его внешнего состояния и поведения

2. Абстракция - расширяемость типов - способность добавлять типы, определяемые пользователем для того, чтобы дополнить ими встроенные типы. Один из принципов ООП заключается в том, чтобы типы, определяемые пользователем, должны обладать теми же привилегиями, что и встроенные типы.

3. Наследование - способность создавать новые типы, повторно используя описание существующих типов.

4. Полиморфизм с динамическим (поздним) связыванием - способность объектов быть ответственными за интерпретацию вызова функции

Ключевые слова

Объект, структура, состояние и поведение объекта, пять типов операций (методов): конструктор, деструктор, модификатор, селектор, итератор, отношения между классами, объектно-ориентированный анализ, имя класса,

множественность экземпляров класса, иерархия класса, структура и интерфейс класса, шаги объектно-ориентированного проектирования: выявления классов и объектов на установление уровне абстракции, определение семантики классов, определение связей между классами и объектами, реализация классов.

Технология программирования, структурное программирование, процедурное программирование, надежность программного обеспечения, канонические структуры языков программирования: операторы присваивания, циклы и разветвления, определение классов в программе, их описание экземпляры объектов, иерархическая структура объектов.

Принцип Черча, новая парадигма, принципы объектно-ориентированного подхода к решению задач, иерархическое дерево, характеристики ООП, объектно-ориентированный язык, свойства: абстракции, наследования и полиморфизма, принципы объектно-ориентированного подхода.

Вопросы для самоконтроля

1. Как Вы можете охарактеризовать понятие «объект» в объектно-ориентированном проектировании?
2. Назовите пять типов операций, именуемых в объектно-ориентированных языках методами.
3. Что предполагает объектно-ориентированный подход к проектированию программных продуктов?
4. В чем суть объектно-ориентированного анализа?
5. Укажите основные этапы объектно-ориентированного проектирования.
6. Что подразумевается под понятием «технология программирования»?
7. Какие этапы прошла технология программирования и как это отразилось на языках программирования?
8. В чем заключается сущность объектно-ориентированного подхода?
9. Назовите наиболее используемые в настоящее время объектно-ориентированные языки программирования.
10. Каковы основные характеристики продукта Delphi?
11. Что такое масштабируемые средства для построения баз данных?
12. Для кого предназначен Delphi?
13. Что такое библиотека визуальных компонентов?

Рекомендуемая литература

1. Архангельский А. Программирование в Delphi 7. - М.: ООО «Бином – Пресс», 2004. -1152с.
2. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Бином · 1998.
3. Джон Влиссидес, Эрих Гамма, Ричард Хелм, Ральф Джонсон. Приемы объектно-ориентированного проектирования. Паттерны проектирования. - Питер. 2003. - 256 с.

4. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. - М.: Изд. МГТУ имени Н.Э. Баумана. 2003.- 320с.

5. Кьоу Дж., Джеанини М.Объектно-ориентированное программирование. Просто и понятно. − М.: Питер, 2005. -403с.

6. Лесневский А.С. Объектно-ориентированное программирование для начинающих. - М.: Бином. Лаборатория знаний. 2005. - 382 с.

7. Синтес Антони. Освой самостоятельно объектно-ориентированное программирование за 21 день. Просто и понятно− М.: Вильямс. 2002. -284с.

8. http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».

9. http://.borland.com/delphi_net/ - Официальный сайт Borland Delphi

10. http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

Глава 2. Средства объектно-ориентированного программирования в Delphi

2.1. Основные характеристики программного продукта Delphi

Delphi - это греческий город, где жил дельфийский оракул. Этим же именем был назван новый программный продукт с феноменальными характеристиками.

Следует отметить, что к моменту выхода продукта обстановка вокруг компании Borland складывалась не лучшим для нее образом. Поговаривали о возможной перепродаже компании, курс акций который неудержимо катился вниз. Сейчас уже можно без всяких сомнений утверждать, что период трудностей позади. Неверно, конечно, было бы говорить, что только **Delphi** явился причиной восстановления компании; кроме **Delphi**, у Borland появились и другие замечательные продукты, так же, как и **Delphi**, основывающиеся на новых, появившихся недавно у компании Borland, технологиях. Имеются в виду новые BDE 2.0, BC++ 4.5, Paradox for Windows 5.0, dBase for Windows 5.0, BC++ 2.0 for OS/2.

Тем не менее, именно **Delphi** стал тем продуктом, на примере которого стало ясно, что у Borland еще есть порох в пороховницах, и что один единственный продукт может столь удачно сочетать несколько передовых технологий.[22]

Delphi - это комбинация нескольких важнейших технологий:

- высокопроизводительный компилятор в машинный код;
- объектно-ориентированная модель компонент;
- визуальное (а, следовательно, и скоростное) построение приложений из программных прототипов;
- масштабируемые средства для построения баз данных.

Компилятор в машинный код

Компилятор, встроенный в **Delphi**, обеспечивает высокую производительность, необходимую для построения приложений в архитектуре “клиент-сервер”. Этот компилятор в настоящее время является самым быстрым в мире, скорость компиляции который составляет свыше 120 тыс. строк в минуту на компьютере 486DX33. Он предлагает легкость разработки и быстрое время проверки готового программного блока, характерного для языков четвертого поколения (4GL) и, в то же время обеспечивает качество кода, характерного для компилятора 3GL. Кроме того, **Delphi** обеспечивает быструю разработку без необходимости писать вставки на Си или ручного написания кода (хотя это возможно).

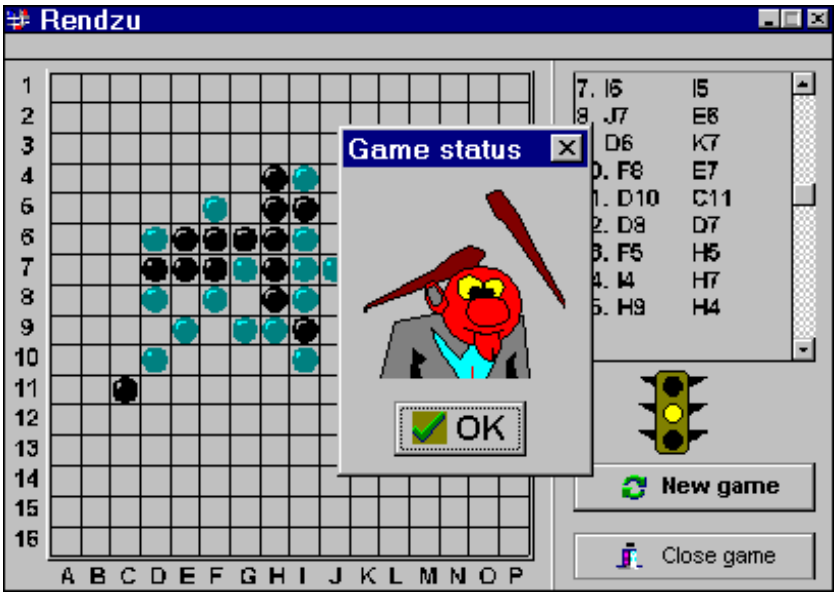
В процессе построения приложения разработчик выбирает из палитры компонентов готовые как художник, делающий крупные мазки кистью. Еще до компиляции он видит результаты своей работы - после подключения к источнику данных их уже можно видеть отображенными на форме, перемещаться по данным, представлять их в том или ином виде. В этом смысле проектирование в **Delphi** мало чем отличается от проектирования в интерпретирующей среде, однако после выполнения компиляции мы получаем код, который исполняется в 10-20 раз быстрее, чем то же самое, осуществленное с помощью интерпретатора. Помимо этого, компилятор компилятору рознь, в **Delphi** компиляция производится непосредственно в родной машинный код, в то время как существуют компиляторы, превращающие программу в так называемый p-код, который затем интерпретируется виртуальной p-машиной. Это не может не сказаться на фактическом быстродействии готового приложения.

Объектно-ориентированная модель программных компонент

Основной упор этой модели в **Delphi** делается на максимальном реиспользовании кода, что позволяет разработчикам строить приложения весьма быстро из заранее подготовленных объектов, а также дает им возможность создавать свои собственные объекты для среды **Delphi**. Никаких ограничений по типам объектов, которые могут создавать разработчики, не существует. Действительно, все в **Delphi** написано на нем же, поэтому разработчики имеют доступ к тем же объектам и инструментам, которые использовались для создания среды разработки. В результате нет никакой разницы между объектами, поставляемыми Borland или третьими фирмами, и объектами, которые вы можете создать.[22,25,29]

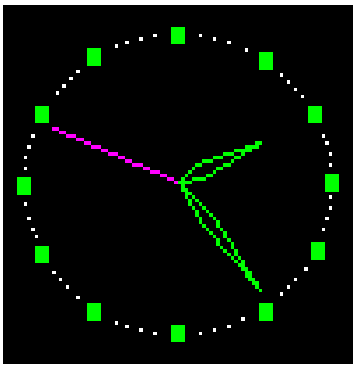
В стандартную поставку **Delphi** входят основные объекты, которые образуют удачно подобранную иерархию из 270 базовых классов. На **Delphi** можно одинаково хорошо писать как приложения к корпоративным базам данных, так и, к примеру, игровые программы. Во многом это объясняется тем, что традиционно в среде Windows было достаточно сложно реализовывать пользовательский интерфейс. Событийная модель в Windows всегда была сложна для понимания и отладки. Но именно разработка интерфейса в **Delphi** является самой простой задачей для программиста.

Быстрая разработка работающего приложения из прототипов



Например, игровая программа Rendzu была собрана из готовых кусков за один рабочий день, причем большая часть времени была посвящена прихорашиванию и приукрашиванию. Screen Saver в виде прыгающих часиков был также изготовлен на **Delphi** за весьма незначительное время. Теперь эти часики украшают почти каждую IBM-совместимую машину.

Конечно, на разработку серьезной информационно-поисковой системы в архитектуре клиент-сервер может уйти гораздо большее время, чем на разработку программы-игрушки. Тем не менее многие специалисты, программировавшие на других языках, утверждают, что на **Delphi** скорость изготовления сложного проекта выше раз в десять.



Среда **Delphi** включает в себя полный набор визуальных инструментов для скоростной разработки приложений (RAD - rapid application development), поддерживающих разработку пользовательского интерфейса и подключение к корпоративным базам данных. VCL - библиотека визуальных компонентов, включает в себя стандартные объекты построения пользовательского интерфейса, объекты управления данными, графические объекты, объекты мультимедиа, диалоги и объекты управления файлами, управление DDE и OLE. Единственное, что можно поставить в вину **Delphi**, это то, что готовых компонентов, поставляемых Borland, могло бы быть и больше. Однако, разработки других фирм, а также свободно распространяемые программистами freeware-компоненты уже восполнили этот недостаток.[25]

Соответствующий стандарт компонент назывался VBX. И этот стандарт так же поддерживается в **Delphi**. Однако, визуальные компоненты в **Delphi** обладают большей гибкостью. Вспомним, в чем заключалось проблема в *VB*. Прикладной программист программировал, вообще говоря, в среде языка бэйсик. Компоненты же в стандарте VBX ему готовили его коллеги-профессионалы на C++. VBX приходили, “как есть”, и ни исправить, ни добавить ничего было нельзя.

Для изготовления же VBX следовало освоить “кухню” языка C++. В **Delphi** визуальные компоненты пишутся на объектном паскале, т.е. на том же Паскале, на котором пишется алгоритмическая часть приложения. И

визуальные компоненты **Delphi** получаются открытыми для надстройки и переписывания.

Масштабируемые средства для построения баз данных

Объекты БД в **Delphi** основаны на SQL и включают в себя полную мощь Borland Database Engine. В состав **Delphi** также включен Borland SQL Link, поэтому доступ к СУБД Oracle, Sybase, Informix и InterBase происходит с высокой эффективностью. Кроме того, **Delphi** включает в себя локальный сервер Interbase для того, чтобы можно было разработать расширяемые на любые внешние SQL-сервера приложения в офлайновом режиме. Разработчик в среде **Delphi**, проектирующий информационную систему для локальной машины (к примеру, небольшую систему учета медицинских карточек для одного компьютера), может использовать для хранения информации файлы формата **.dbf** (как в dBase или Clipper) или **.db** (Paradox). Если же он будет использовать локальный InterBase for Windows 4.0 (это локальный SQL-сервер, входящий в поставку), то его приложение безо всяких изменений будет работать и в составе большой системы с архитектурой клиент-сервер.[27,29]

Вот она - масштабируемость на практике - одно и то же приложение можно использовать как для локального, так и для более серьезного клиент-серверного вариантов.

Delphi - два варианта поставки.

Что лежит внутри в коробке, и чем может воспользоваться программист при разработке прикладной системы? Выпущены две версии Delphi - одна (Delphi Client-Server) адресована для разработчиков приложений в архитектуре “клиент-сервер”, а другая (Delphi for Windows) для остальных программистов. Приложения, разработанные с помощью Delphi, можно использовать без выплаты royalty-процентов и без оплаты runtime- лицензий.

Клиент-серверная версия Delphi

Она адресована корпоративным разработчикам, желающим разрабатывать высокопроизводительные приложения для рабочих групп и корпоративного применения. Клиент-серверная версия включает в себя следующие особенности:

SQL Links: специально написанные драйверы для доступа к Oracle, Sybase, Informix, InterBase

Локальный сервер InterBase: SQL-сервер для Windows 3.1 и версий выше. СУБД для разработки в корпоративных приложений на компьютере, не подключенном к локальной сети. ReportSmith Client/server Edition: генератор отчетов для SQL-серверов. Team Development Support: предоставляет версионный контроль с помощью PVCS компании Intersolve (приобретается отдельно) или посредством других программных продуктов версионного контроля.Visual Query Builder - это средство визуального построения SQL-запросов, лицензия на право распространения приложений в архитектуре клиент-сервер, изготовленные при помощью Delphi исходные тексты всех визуальных компонентов

Delphi for Windows

Delphi for Windows представляет собой подмножество **Delphi** Client-Server и предназначен для разработчиков высокопроизводительных персональных приложений, работающих с локальными СУБД типа dBase и Paradox.**Delphi** Desktop Edition предлагает такую же среду для быстрой разработки и первоклассный компилятор как и клиент-серверная версия (Client/Server Edition), Которая позволяет разработчику быстро изготавливать персональные приложения, работающие с персональными СУБД типа dBase и Paradox. **Delphi** также позволяет создавать разработчику DLL, которая может быть вызвана из Paradox, dBase, C++ или каких-нибудь других готовых программ.

В Delphi for Windows, как и в Delphi Client-Server, входят компилятор Object Pascal (этот язык является расширением языка Borland Pascal 7.0) генератор отчетов ReportSmith 2.5 (у которого, правда, отсутствует возможность работы с SQL-серверами),среда визуального построителя приложений,библиотека визуальных компонентов, локальный сервер InterBase, RAD Pack for Delphi

В этом обзоре следует упомянуть еще один продукт, выпущенный компанией Borland для **Delphi**. В RAD Pack for **Delphi** входит набор полезных дополнений, которые помогут разработчику при освоении и использовании **Delphi**. Это учебник по объектному паскалю, интерактивный отладчик самой последней версии, Borland Visual Solutions Pack (набор VBX для реализации редакторов, электронных таблиц, коммуникационные VBX, VBX с деловой графикой и т.п.), Resource WorkShop для работы с ресурсами Borland Pascal 7.0, а также дельфийский эксперт для преобразования ресурсов BP 7.0 в формы **Delphi**.

Для кого предназначен Delphi

В первую очередь **Delphi** предназначен для профессионалов-разработчиков корпоративных информационных систем. Может быть, здесь следует пояснить, что конкретно имеется в виду. Не секрет, что некоторые удачные продукты, предназначенные для скоростной разработки приложений (*RAD - rapid application development*) прекрасно работают при изготовлении достаточно простых приложений, однако, разработчик сталкивается с непредвиденными сложностями, когда пытается сделать что-то действительно сложное. Бывает, что в продукте вскрываются присущие ему ограничения только по прошествии некоторого времени.

Delphi такие ограничения не присущи. Хорошее доказательство тому - это тот факт, что сам **Delphi** разработан на **Delphi**. Можете делать выводы. Однако **Delphi** предназначен не только для программистов-профессионалов. В электронной конференции встречаются совершенно неожиданные письма, где учителя, врачи, преподаватели вузов, бизнесмены, все те, кто использует компьютер с сугубо прикладной целью, рассказывали о том, что приобрели **Delphi for Windows** для того, чтобы быстро решить какие-то свои задачи, не привлекая для этого программистов со стороны. В большинстве случаев им это удается. Поразительный факт - журнал Visual Basic Magazine присудил свою премию **Delphi for Windows**.

Некоторые особенности Delphi

Локальный сервер InterBase - следует заметить, что данный инструмент предназначен только для автономной отладки приложений. В действительности он представляет собой сокращенный вариант обработчика SQL-запросов InterBase, в который не включены некоторые возможности настоящего сервера InterBase. Отсутствие этих возможностей с лихвой компенсируется преимуществом автономной отладки программ.

Team Development Support - средство поддержки разработки проекта в группе. Позволяет существенно облегчить управление крупными проектами. Это сделано в виде возможности подключения такого продукта как Intersolve PVCS 5.1 непосредственно к среде **Delphi.**

Высокопроизводительный компилятор в машинный код - в отличие от большинства Паскаль-компиляторов, транслирующих в р-код, в **Delphi** программный текст компилируется непосредственно в машинный код, в результате чего **Delphi**- приложения исполняются в 10-20 раз быстрее (особенно приложения, использующие математические функции). Готовое приложение может быть изготовлено либо в виде исполняемого модуля, либо в виде динамической библиотеки, которую можно использовать в приложениях, написанных на других языках программирования.

Открытая компонентная архитектура

Благодаря такой архитектуре, приложения, изготовленные с помощью **Delphi**, работают надежно и устойчиво. **Delphi** поддерживает использование уже существующих объектов, включая DLL, написанные на C и C++, OLE сервера, VBX, объекты, созданные с помощью **Delphi**. Из готовых компонентов работающие приложения собираются весьма быстро. Кроме того, поскольку **Delphi** имеет полностью объектную ориентацию, разработчики могут создавать свои, повторно используемые объекты, для того, чтобы уменьшить затараты на разработку.

Delphi предлагает разработчикам - как в составе команды, так и индивидуальным - открытую архитектуру, позволяющую добавлять компоненты, где бы они ни были изготовлены, и оперировать этими вновь введенными компонентами в визуальном построителе. Разработчики могут добавлять CASE-инструменты, кодовые генераторы, а также авторские help’ы, доступные через меню **Delphi.**[29]

Two-way tools - однозначное соответствие между визуальным проектированием и классическим написанием текста программы. Это означает, что разработчик всегда может видеть код, соответствующий тому, что он построил с помощью визуальных инструментов и наоборот.

Визуальный построитель интерфейсов (Visual User-interface builder) дает возможность быстро создавать клиент-серверные приложения визуально, просто выбирая компоненты из соответствующей палитры.

Библиотека визуальных компонент

Эта библиотека объектов включает в себя стандартные объекты построения пользовательского интерфейса, объекты управления данными, графические объекты, объекты мультимедиа, диалоги и объекты управления файлами, управление DDE и OLE.

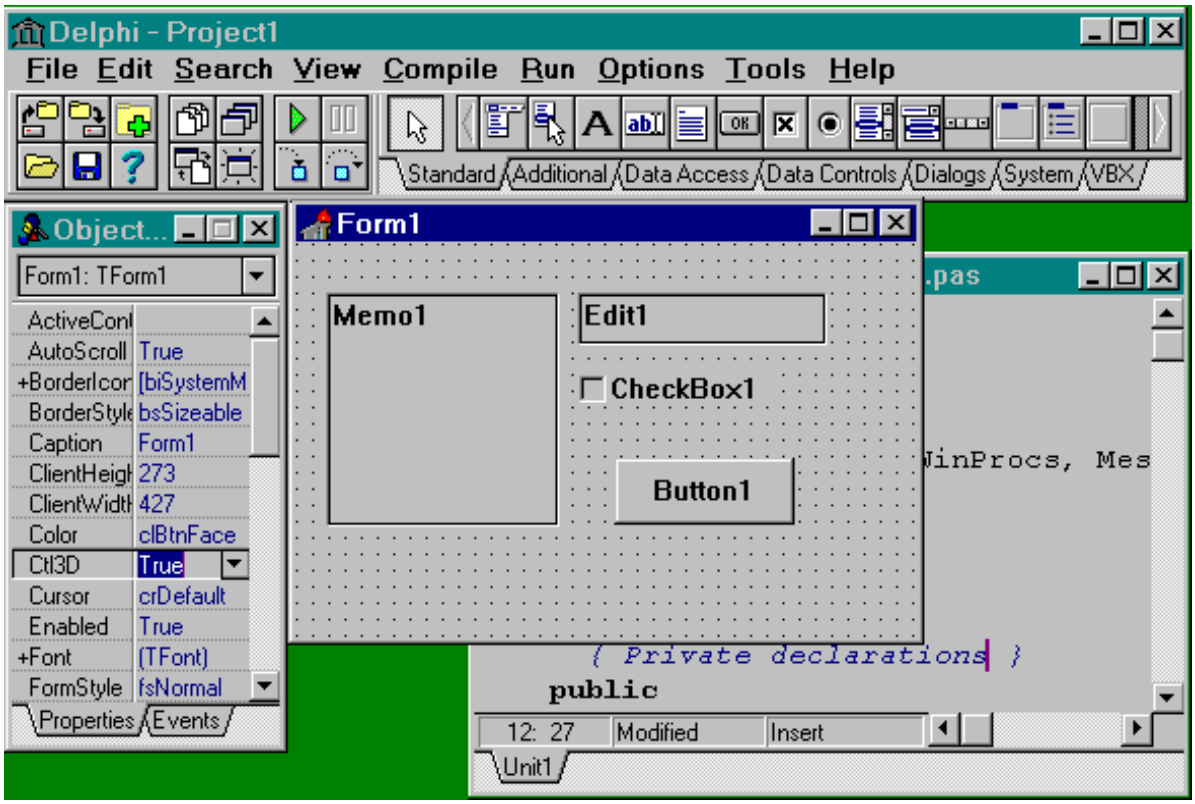
Структурное объектно-ориентированное программирование

Delphi использует структурный объектно-ориентированный язык (Object Pascal), который сочетает, с одной стороны, выразительную мощь и простоту программирования, характерную для языков 4GL,с другой - эффективность языка 3GL. Программисты немедленно могут начать производить работающие приложения, и для этого им не придется изучать особенности программирования событий в Windows. **Delphi** полностью поддерживает передовые программные концепции, включая инкапсуляцию, наследование, полиморфизм и управление событиями.

Поддержка OLE 2.0, DDE и VBX

Это весьма важная особенность для разработчиков в среде Windows, поскольку в уже существующие Windows-приложения программист может интегрировать то, что разработает с помощью **Delphi**.

Delphi: настраиваемая среда разработчика



После запуска **Delphi** в верхнем окне горизонтально располагаются иконки палитры компонент. Если курсор задерживается на одной из иконок, под ней в желтом прямоугольнике появляется подсказка.

Из этой палитры компонентов можно выбирать те, из которых можно строить приложения. Компоненты включают в себя как визуальные, так и логические. Такие вещи, как кнопки, поля редактирования - это визуальные компоненты; а таблицы, отчеты - это логические.

Понятно, что поскольку в **Delphi** вы визуальным образом строите свою программу, все эти компоненты имеют свое графическое представление в поле форм для того, чтобы можно было бы ими соответствующим образом оперировать. Но для работающей программы видимыми остаются только

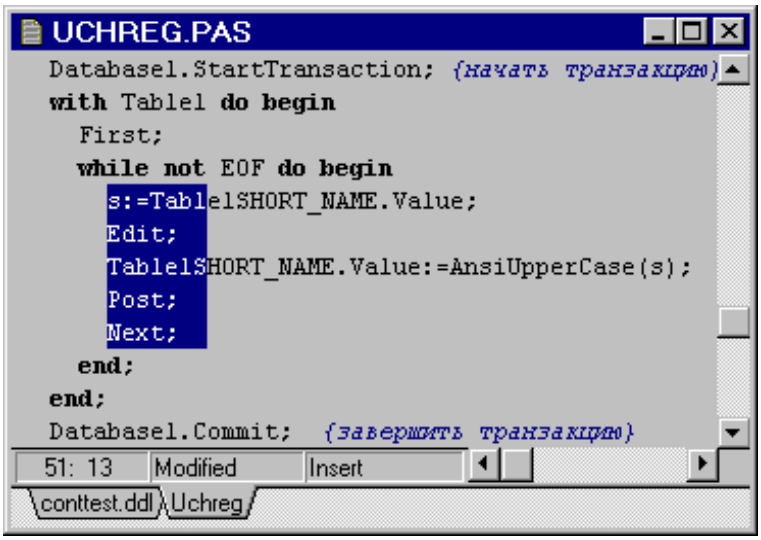
визуальные компоненты. Компоненты сгруппированы на страницах палитры по своим функциям. К примеру, компоненты, представляющие Windows “common dialogs”, все размещены на странице палитры с названием “Dialogs”.

Delphi позволяет разработчикам настроить среду для максимального удобства. Вы можете легко изменить палитру компонент, инструментальную линейку, а также настраивать выделение синтаксиса цветом.

Заметим, что в **Delphi** вы можете определить свою группу компонент и разместить ее на странице палитры, а если возникнет необходимость, перегруппировать компоненты или удалить неиспользуемые.

Интеллектуальный редактор

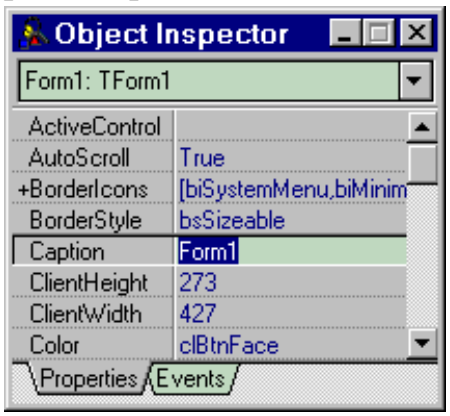
Редактирование программ можно осуществлять, используя запись и исполнение макросов, работу с текстовыми блоками, настраиваемые комбинации клавиш и цветовое выделение строк.



Графический отладчик

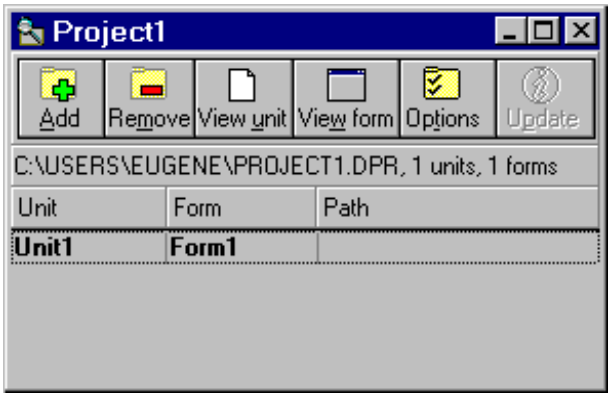
Delphi обладает мощнейшим, встроенным в редактор графическим отладчиком, позволяющим находить и устранять ошибки в коде. Вы можете установить точки останова, проверить и изменить переменные, с помощью пошагового выполнения в точности понять поведение программы. Если же требуются возможности более тонкой отладки, Вы можете использовать отдельно доступный Turbo Debugger, проверив ассемблерные инструкции и регистры процессора.

Инспектор объектов представляет собой отдельное окно, где вы можете в период проектирования программы устанавливать значения свойств и событий объектов (Properties & Events).



Менеджер проектов.

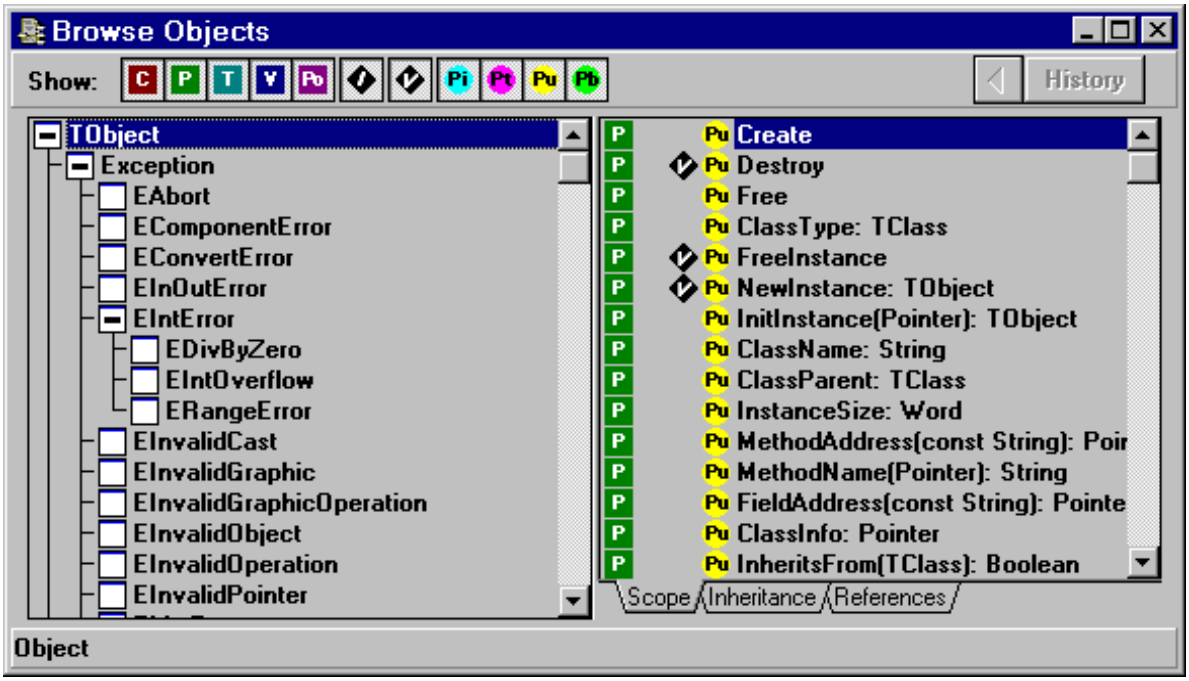
Дает возможность разработчику просмотреть все модули в соответствующем проекте и снабжает удобным механизмом для управления проектами.



Менеджер проектов показывает имена файлов, время/дату выбранных форм и пр. Можно немедленно попасть в текст или форму, просто щелкнув

мышкой на соответствующее имя.

Навигатор объектов



Показывает библиотеку доступных объектов и осуществляет навигацию по вашему приложению. Можно посмотреть иерархию объектов, рекомпилированные модули в библиотеке, список глобальных имен вашего кода.

Дизайнер меню помогает создавать меню, сохранить созданные данные в виде шаблонов и затем использовать в их в любом приложении**.**

Эксперты

Это набор инструментальных программ, облегчающих проектирование и настройку Ваших приложений.

Есть возможность подключать самостоятельно разработанные эксперты. Потенциально это та возможность, с помощью которой третьи фирмы могут расширять **Delphi** CASE-инструментами, разработанными специально для **Delphi**.[50,51,53]

Включает в себя эксперт:

форм, работающих с базами данных;

стилей и шаблонов приложений;

шаблонов форм.

В состав RAD Pack входит эксперт для преобразования ресурсов, изготовленных в Borland Pascal 7.0, в формы **Delphi**. Уже появились эксперты, облегчающие построение DLL и даже написание собственных экспертов.

Интерактивная обучающая система позволяет более полно освоить Delphi. Она является не просто системой подсказок, а указывает на возможности Delphi в самой среде разработчика.

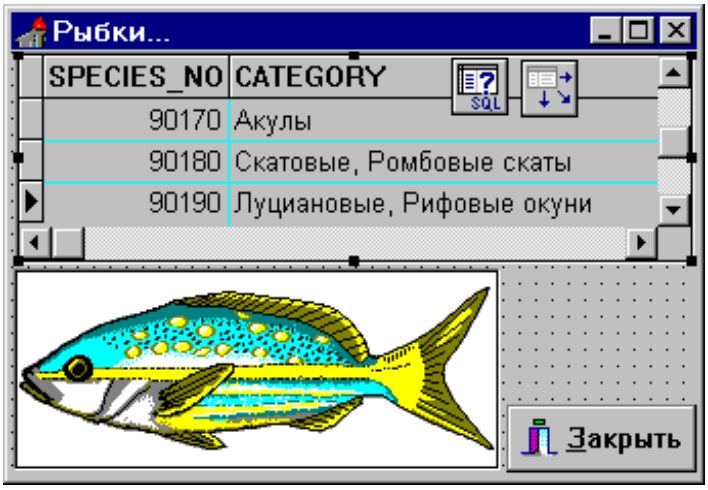
Компоненты доступа к базам данных и визуализации данных.

Библиотека объектов содержит набор визуальных компонент, значительно упрощающих разработку приложений для СУБД с архитектурой клиент-сервер. Объекты инкапсулируют в себя нижний уровень - Borland Database Engine.

Предусмотрены специальные наборы компонент, отвечающих за доступ к данным, и компонент, отображающих данные. Компоненты доступа к данным позволяют осуществлять соединения с БД, производить выборку, копирование данных, и т.п.

Компоненты визуализации данных позволяют отображать данные виде таблиц, полей, списков. Отображаемые данные могут быть текстового, графического или произвольного формата.

Разработка приложений БД



Delphi позволяет использовать библиотеку визуальных компонент для быстрого создания надежных приложений, которые легко расширяются до приложений с архитектурой клиент-сервер. Другими словами, можно создать приложение, работающее с локальным сервером InterBase, а затем использовать созданное приложение, соединяясь с удаленным SQL-сервером через SQL-Links.

Библиотека объектных визуальных компонент

Компоненты, используемые при разработке в **Delphi** (и, также, собственно самим **Delphi**), встроены в среду разработки приложений и представляя собой набор типов объектов, используемых в качестве фундамента при строительстве приложения.[25]

Этот костяк называется **Visual Component Library** (VCL). В VCL есть такие стандартные элементы управления, как строки редактирования, статические элементы управления, строки редактирования со списками, списки объектов.

Еще имеются такие компоненты, которые ранее были доступны только в библиотеках третьих фирм: табличные элементы управления, закладки, многостраничные записные книжки.

VCL содержит специальный объект, предоставляющий интерфейс графических устройств Windows и позволяющий разработчикам рисовать, не заботясь об обычных для программирования в среде Windows деталях.

Ключевой особенностью **Delphi** является возможность не только использовать визуальные компоненты для строительства приложений, но и создать новые компоненты. Такая возможность позволяет разработчикам не переходить в другую среду разработки, а, наоборот, встраивать новые инструменты в существующую среду. Кроме того, можно улучшить или полностью заменить существующие по умолчанию в **Delphi** компоненты.

Здесь следует отметить, что обычных ограничений, присущих средам визуальной разработки, в **Delphi** нет. Сам **Delphi** написан с помощью **Delphi**, что свидетельствует об отсутствии таких ограничений.

Классы объектов построены в виде иерархии, состоящей из абстрактных, промежуточных и готовых компонент. Разработчик может пользоваться готовыми компонентами, создавать собственные на основе абстрактных или промежуточных, а также создавать собственные объекты.

Язык программирования **Delphi** базируется на Borland Object Pascal.

Помимо этого, **Delphi** поддерживает такие низкоуровневые особенности, как подклассы элементов управления Windows, перекрытие цикла обработки сообщений Windows, использование встроенного ассемблера.

Формы, модули и метод разработки “Two-Way Tools”.

Формы - это объекты, в которые вы помещаете другие объекты для создания пользовательского интерфейса вашего приложения. Модули состоят из кода, который реализует функционирование вашего приложения, обработчики событий для форм и их компонент.

Информация о формах хранится в двух типах файлов - *.dfm* и *.pas*, причем первый тип файла - двоичный - хранит образ формы и ее свойства, второй тип описывает функционирование обработчиков событий и поведение компонент. Оба файла автоматически синхронизируются **Delphi**, так что если добавить новую форму в ваш проект, связанный с ним файл *.pas* автоматически будет создан, и его имя будет добавлено в проект.

Такая синхронизация и делает **Delphi** two-way-инструментом, обеспечивая полное соответствие между кодом и визуальным представлением. Как только вы добавите новый объект или код, **Delphi** устанавливает так называемую “кодовую синхронизацию” между визуальными элементами и соответствующими им кодовыми представлениями.

Например, предположим, вы добавили описание поведения формы (соотв. обработчик событий), чтобы показать окно сообщения по нажатию кнопки. Такое описание появляется, если дважды щелкнуть мышкой непосредственно на оъект Button в форме или дважды щелкнуть мышью на строчку OnClick на странице Events в Инспекторе объектов. В любом случае **Delphi** создаст процедуру или заголовок метода, куда вы можете добавить код.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

Создавая этот код, **Delphi** автоматически формирует декларацию объекта TForm1, которая содержит процедуру ButtonClick, представляющую собой собственно обработчик события.

```
TForm1 = class (TForm)
 Button1: Tbutton;
 procedure Button1Click(Sender: TObject);
private
{ Private declarations }
```

```
public
{ Public declarations }
end;
```

Конечно, вы запросто можете решить после получения этого кода, что автоматически созданные имена Вас не устраивают, и заменить их. Например, Button1 на Warning. Это можно сделать, изменив свойство Name для Button1 с помощью Инспектора объектов. Как только вы нажмете Enter, **Delphi** автоматически произведет соответствующую синхронизацию в коде. Так как объект TForm1 существует в коде, вы свободно можете добавлять любые другие поля, процедуры, функции или object definition. К примеру, вы можете дописать в коде свою собственную процедуру, обрабатывающую событие, а не делать это визуальным методом.

Следующий пример показывает, как это можно сделать. Обработчик принимает аргумент типа TObject, который позволяет нам определить, если необходимо, кто инициировал событие. Это полезно в случае, когда несколько кнопок вызывают общую процедуру для обработки.

```
TForm1 = class(TForm)
 Warning: TButton;
 Button1: TButton;
 procedure WarningClick(Sender: TObject);
 procedure NewHandler(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
```

Здесь мы имеем дело уже со второй стороной синхронизации. Визуальная среда в данном случае распознает, что новая процедура добавлена к объекту и соответствующие имена появляются в Инспекторе объектов.

Добавление новых объектов

Delphi - это прежде всего среда разработки, базирующаяся на использовании компонент. Поэтому можно добавлять совершенно новые компоненты в палитру компонент. Можно создавать компоненты внутри **Delphi**, или вводить компоненты, созданные как управляющие элементы VBX или OLE 2.0, или же вы можете использовать компоненты, написанные на C или C++ в виде dll.

Последовательность введения новой компоненты состоит из трех шагов:

- наследование из уже существующего типа компоненты;
- определение новых полей, свойств и методов;
- регистрация компоненты.

Это все делается через меню *Install Components*.

Добавление управляющих элементов VBX

Delphi генерирует объектное расширение VBX, которое инсталлируется в качестве компонент.

Например, если вы инсталлируете SaxComm VBX из Visual Solutions Pack компании Borland в **Delphi**, автоматически генерится тип объекта TComm, который наследуется из стандартного TVBXControl. Когда вы инсталлируете компоненты, **Delphi** будет компилировать и подлинковывать их к библиотеке компонент.

Делегирование: события программируются проще

Под делегированием понимается то, что некий объект может предоставить другому объекту отвечать на некоторые события.

Такая модель в некоторых случаях значительно упрощает программирование. Например, вместо того, чтобы создавать подкласс для Windows controls при добавлении нового поведения, можно просто привязать процедуру обработки события, которая автоматически будет вызываться на каждый щелчок мышью пользователем или нажатием им клавиши. Аналогично вы можете написать процедуру определения допустимости таблицы, которая будет выполняться обработчиком события, когда транзакция начинается или завершается, записи вставляются, удаляются или изменяются.

К примеру, когда вы добавляете кнопку в форму и прикрепляете код, обрабатывающий нажатие, вы фактически используете делегирование кода для ассоциирования кода с событием OnClick. Такая ассоциация происходит для вас автоматически. Если проверить страницу Events в Инспекторе объектов для вашего приложения, можно увидеть ассоциированные с событиями процедуры.[51]

Ссылки на классы

Ссылки на классы придают дополнительный уровень гибкости. Так, когда вы хотите динамически создавать объекты, чьи типы могут быть известны только во время выполнения кода. К примеру, ссылки на классы используются при формировании пользователем документа из разного типа объектов, где пользователь набирает нужные объекты из меню или палитры. Собственно, эта технология использовалась и при построении **Delphi.**

Обработка исключительных ситуаций

Серьезные приложения должны надежным образом обрабатывать исключительные ситуации, сохранять, если возможно, выполнение программы или, если это невозможно, аккуратно ее завершать. Написание кода, обрабатывающего исключительные ситуации, всегда было непростой задачей и являлось источником дополнительных ошибок.

В **Delphi** это устроено в стиле C++. Исключения представлены в виде объектов, содержащих специфическую информацию о соответствующей ошибке (тип и место-нахождение ошибки). Разработчик может оставить обработку ошибки, существовавшую по умолчанию, или написать свой собственный обработчик.

Обработка исключений реализована в виде *exception-handling blocks* (также еще называется *protected blocks*), которые устанавливаются ключевыми словами **try** и **end**. Существуют два типа таких блоков: ***try...except*** и ***try...finally***.

Общая конструкция выглядит примерно так:

try

```
 { выполняемые операторы }
except
 on exception1 do statement1; { реакция на ситуации }
 on exception2 do statement2;
else
 { операторы по умолчанию }
end;
```

Конструкция ***try....finally*** предназначена для того, чтобы разработчик мог быть полностью уверен в том, что, что бы ни случилось, перед обработкой исключительной ситуации всегда будет выполнен некоторый код (например, освобождение ресурсов).

```
try
 { выполняемые операторы }
finally
 { операторы, выполняемые безусловно }
end;
```

Немного о составе продукта

Документация.

Руководство пользователя

Руководство по написанию компонент

Построение приложений, работающих с базами данных

Руководство по генератору отчетов ReportSmith

Руководство по SQL Links

В состав Delphi входят пять интерактивных обучающих систем, документация в электронном виде и около 10 Мб справочной информации.

Требования к аппаратным и программным средствам

Windows 3.1 и выше

27 Mb дискового пространства для минимальной конфигурации

50 Mb дискового пространства для нормальной конфигурации

процессор 80386, а лучше 80486

6-8 Mb RAM

2.2. Среда программирования Delphi

2.2.1.Общая структура среды программирования

Внешний вид среды программирования Delphi отличается от многих других из тех, что можно увидеть в Windows. К примеру, Borland Pascal for Windows 7.0, Borland C++ 4.0, Word for Windows, Program Manager - это все MDI приложения и выглядят по-другому, чем Delphi. MDI (Multiple Document Interface) - определяет особый способ управления нескольких дочерних окон внутри одного большого окна.

Среда Delphi же следует другой спецификации, называемой Single Document Interface (SDI), и состоит из нескольких отдельно расположенных

окон. Это было сделано из-за того, что SDI близок к той модели приложений, что используется в Windows 95 и выше.[25]

Если Вы используете SDI приложение типа Delphi, то уже знаете, что перед началом работы лучше минимизировать другие приложения, чтобы их окна не загромождали рабочее пространство. Если нужно переключиться на другое приложение, то просто щелкните мышкой на системную кнопку минимизации Delphi. Вместе с главным окном свернутся все остальные окна среды программирования, освободив место для работы других программ.

2.2.2. Основные элементы

1. Дизайнер Форм (Form Designer)
2. Окно Редактора Исходного Текста (Editor Window)
3. Палитра Компонент (Component Palette)
4. Инспектор Объектов (Object Inspector)
5. Справочник (On-line help)

Есть, конечно, и другие важные составляющие Delphi, вроде линейки инструментов, системного меню и многие другие, нужные для точной настройки программы и среды программирования.

Программисты на Delphi проводят большинство времени переключаясь между Дизайнером Форм и Окном Редактора Исходного Текста (которое для краткости называют Редактор). Прежде чем начать, убедитесь, что можете распознать эти два важных элемента. Дизайнер Форм показан на рис.1, окно Редактора - на рис.2.

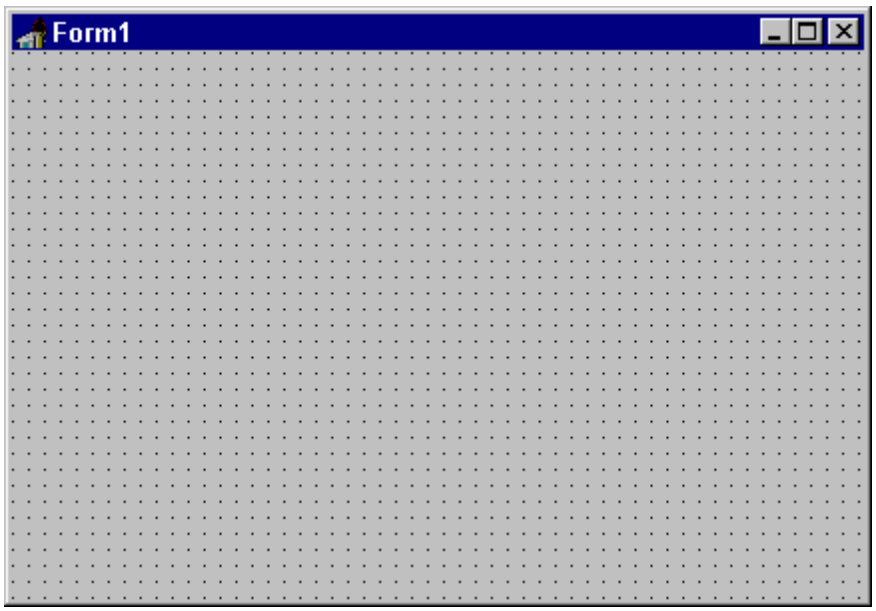


Рис.1. Дизайнер Форм - то место, где создается визуальный интерфейс программы.

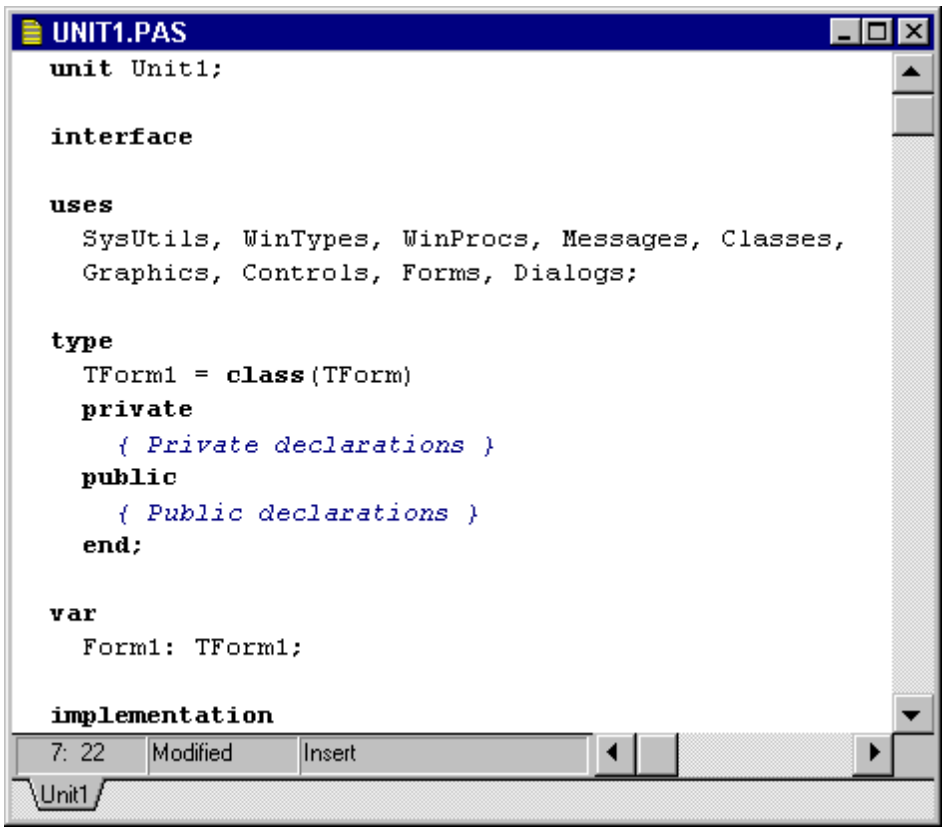


Рис.2. В окне Редактора создается логика управления программой.

Дизайнер Форм в Delphi столь интуитивно понятен и прост в использовании, что создание визуального интерфейса превращается в детскую игру. Дизайнер Форм первоначально состоит из одного пустого окна, которое заполняется всевозможными объектами, выбранными на Палитре Компонент.

Несмотря на всю важность Дизайнера Форм, местом, где программисты проводят основное время является Редактор. Логика - это движущая силой программы и Редактор - то место, где пользователь ее “кодирует”.

Палитра Компонент (см. рис.3) позволяет выбрать нужные объекты для их размещения на Дизайнере Форм. Для использования Палитры Компонент просто первый раз щелкните мышкой на один из объектов и потом второй раз - на Дизайнере Форм. Выбранный Вами объект появится на проектируемом окне и им можно манипулировать с помощью мыши. [22]

Палитра Компонент использует постраничную группировку объектов. Внизу Палитры находится набор закладок - Standard, Additional, Dialogs и т.д. Если Вы щелкнете мышью на одну из закладок, то можете перейти на следующую страницу Палитры Компонент. Принцип разбиения на страницы широко используется в среде программирования Delphi и его легко можно использовать в своей программе. (На странице Additional есть компоненты для организации страниц с закладками сверху и снизу).

Рис.3. Палитра Компонент - место, где выбираются объекты, которые будут помещены на форму.

Предположим, Вы помещаете компонент TEdit на форму; Вы можете двигать его с места на место. Вы также можете использовать границу, прорисованную вокруг объекта для изменения его размеров. Большинством других компонент можно манипулировать тем же образом. Однако невидимые во время выполнения программы компоненты (типа TMenu или TDataBase) не меняют своей формы.

Слева от Дизайнера Форм Вы можете видеть инспектор объектов (рис.4). Заметьте, что информация в инспекторе объектов меняется в зависимости от объекта, выбранного на форме. Важно понять, что каждый компонент является настоящим объектом и Вы можете менять его вид и поведение с помощью инспектора объектов.

Инспектор Объектов состоит из двух страниц, каждую из которых можно использовать для определения поведения данного компонента. Первая страница - это список свойств, вторая - список событий. Если нужно изменить что-нибудь, связанное с определенным компонентом, то Вы, обычно, делаете это в Инспекторе Объектов. К примеру, Вы можете изменить имя и размер компонента TLabel изменяя свойства Caption, Left, Top, Height, и Width.

Вы можете использовать закладки внизу Инспектора Объектов для переключения между страницами свойств и событий. Страница событий связана с Редактором; если Вы дважды щелкнете мышкой на правую сторону какого-нибудь пункта, то соответствующий данному событию код автоматически запишется в Редактор, сам Редактор немедленно получит фокус, и Вы сразу же имеете возможность добавить код обработчика данного события.

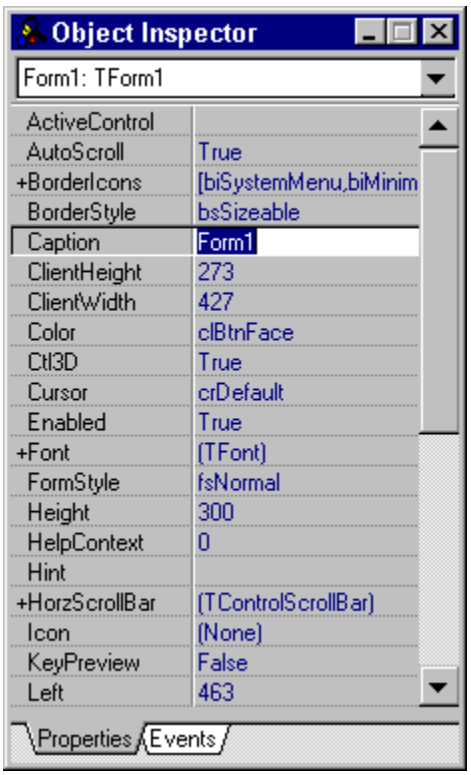


Рис.4. Инспектор Объектов позволяет определять свойства и поведение объектов, помещенных на форму.

Последняя важная часть среды Delphi - Справочник (on-line help). Для доступа к этому инструменту нужно просто выбрать в системном меню пункт Help и затем Contents. На экране появится Справочник, показанный на рис.5.

Справочник является контекстно-зависимым; при нажатии клавиши F1, Вы получите подсказку, соответствующую текущей ситуации. Например, находясь в Инспекторе Объектов, выберите какое-нибудь свойство и нажмите F1 - Вы получите справку о назначении данного свойства. Если в любой момент работы в среде Delphi возникают неясность или затруднение - жмите F1 и необходимая информация появится на экране.

2.2.3. Дополнительные элементы

В данном разделе внимание фокусируется на трех инструментах, которые можно воспринимать как вспомогательные для среды программирования:

- Меню (Menu System)
- Панель с кнопками для быстрого доступа (SpeedBar)
- Редактор картинок (Image Editor)

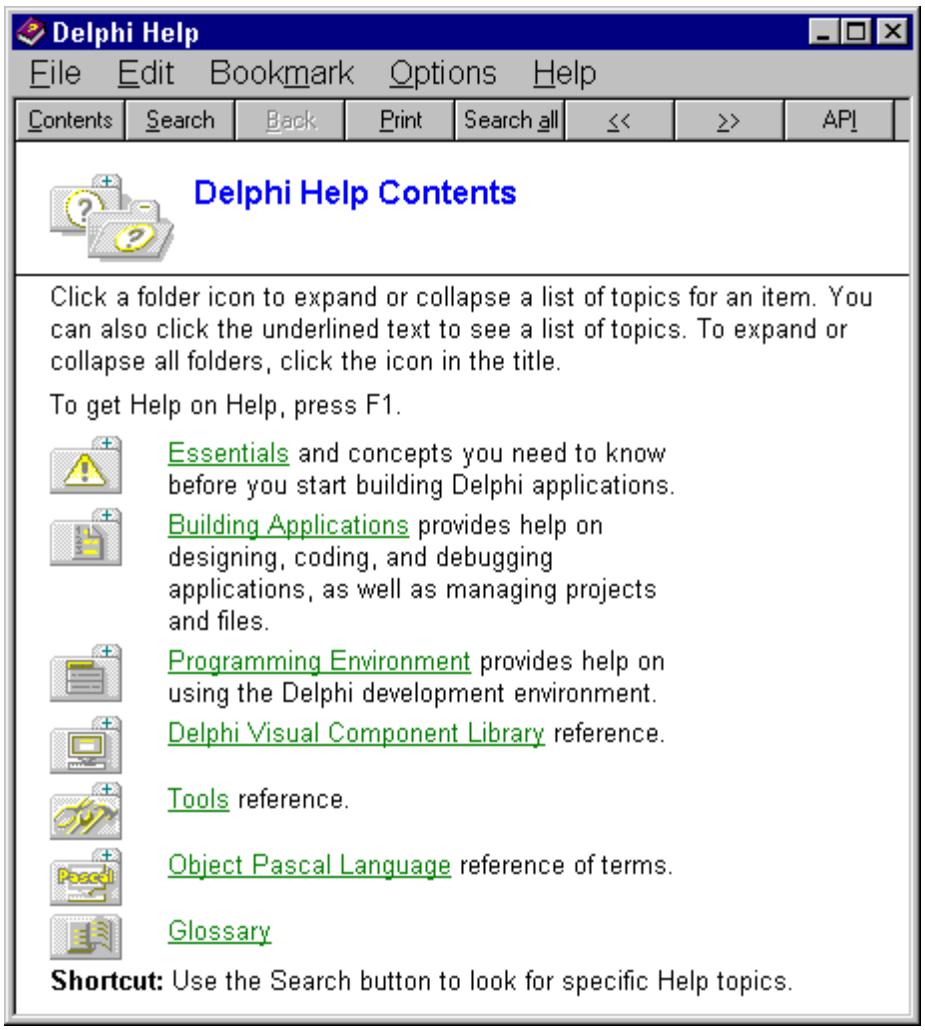


Рис.5. Справочник - быстрый поиск любой информации.

Меню предоставляет быстрый и гибкий интерфейс к среде Delphi, потому что может управляться по набору “горячих клавиш”.[22,25,29] Это удобно еще и потому, что здесь используются слова или короткие фразы, более точные и понятные, нежели иконки или пиктограммы. Вы можете использовать меню для выполнения широкого круга задач; скорее всего, для наиболее общих задач вроде открытия и закрытия файлов, управления отладчиком или настройкой среды программирования.

SpeedBar находится непосредственно под меню, слева от Палитры Компонент (рис.6). SpeedBar выполняет много из того, что можно сделать через меню. Если задержать мышь над любой из иконок на SpeedBar, то Вы увидите, что появится подсказка, объясняющая назначение данной иконки.

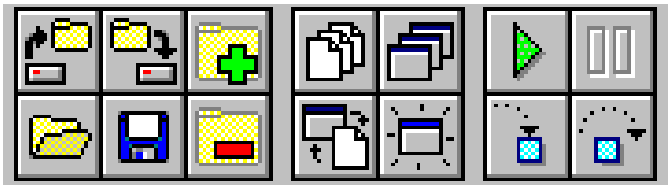


Рис.6. SpeedBar находится слева от Палитры Компонент.

Редактор Картинок, показанный на рис.7, работает аналогично программе Paintbrush из Windows. Вы можете получить доступ к этому модулю, выбрав пункт меню Tools | Image Editor.

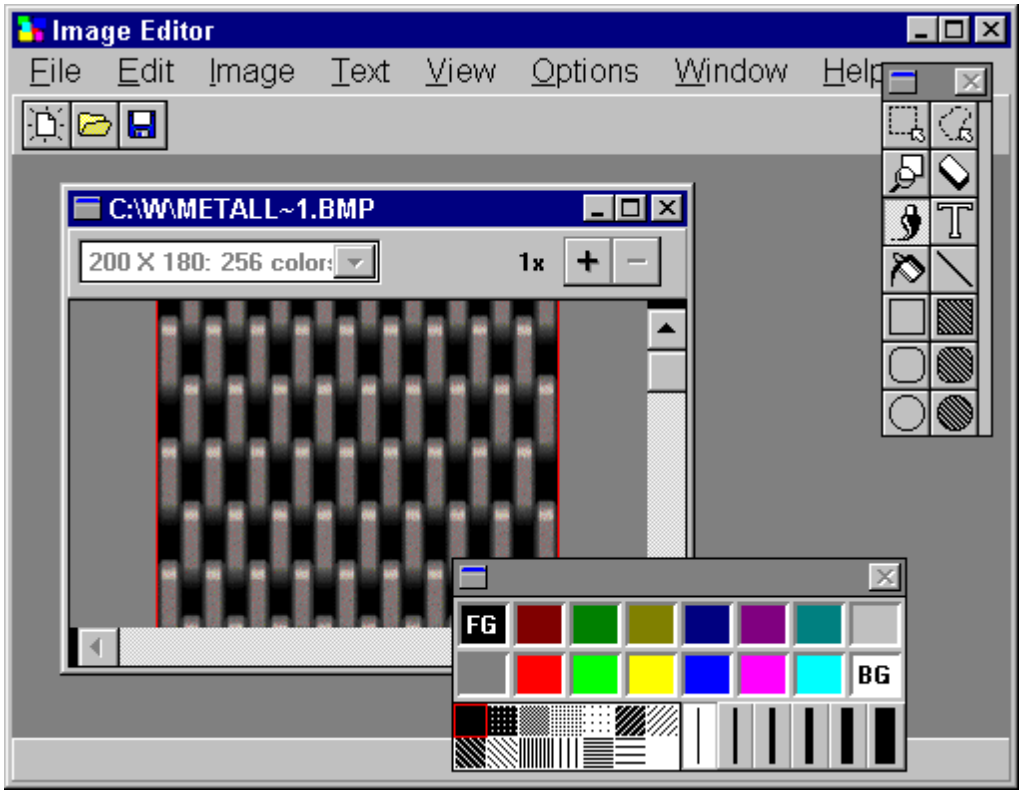


Рис.7. Редактор Картинок можно использовать в качестве создания картинок для кнопок, иконок и других визуальных частей для программы.

Теперь рассмотрим те элементы, которые программист на Delphi использует в повседневной жизни.

2.2.4. Инструментальные средства

В дополнение к обсуждавшимся выше, инструментам, существуют пять средств, поставляемых вместе с Delphi. Эт следующие инструментальные средства:

- Встроенный отладчик
- Внешний отладчик (поставляется отдельно)
- Компилятор командной строки
- WinSight
- WinSpector

Данные инструменты собраны в отдельную категорию не потому, что они менее важны, чем другие, а потому, что играют достаточно абстрактную техническую роль в программировании.

Чтобы стать сильным программистом на Delphi, необходимо понять, как использовать отладчик Delphi. Отладчик позволяет пройти пошагово по исходному тексту программы, выполняя по одной строке за раз, и открыть

просмотровое окно (Watch), в котором будут отражаться текущие значения переменных программы.

Встроенный отладчик, который наиболее важен из пяти вышеперечисленных инструментов, работает в том же окне, что и Редактор. Внешний отладчик делает все, что делает встроенный и кое-что еще. Он более быстр и мощен, чем встроенный. Однако он не так удобен в использовании, главным образом из-за необходимости покидать среду Delphi.

Теперь давайте поговорим о компиляторах. Внешний компилятор, называется DCC.EXE и, в основном, полезен, в том случае если Вы хотите скомпилировать приложение перед отладкой его во внешнем отладчике. Большинство программистов, наверняка, посчитают, то гораздо проще компилировать в среде Delphi, нежели пытаться создать программу из командной строки. Однако, всегда найдется несколько оригиналов, которые будут чувствовать себя счастливее, используя компилятор командной строки. Но это факт - возможно создать и откомпилировать программу на Delphi, используя только DCC.EXE и еще одну программу CONVERT.EXE, которая поможет создать формы. Однако, данный подход неудобен для большинства программистов.[22,25,53]

WinSight и WinSpector интересны преимущественно для опытных программистов в Windows. Это не оззначает, что начинающий не должен их запускать и экспериментировать с ними по своему усмотрению. Однако данные инструменты вторичны и используются для узких технических целей.

Из этих двух инструментов WinSight определенно более полезен. Основная его функция - позволить Вам наблюдать за системой сообщений Windows. Хотя Delphi и делает многое для того, чтобы спрятать сложные детали данной системы сообщений от неопытных пользователей, тем не менее Windows является операционной системой, управляемой событиями. Почти все главные и второстепенные события в среде Windows принимают форму сообщений, которые рассылаются с большой интенсивностью среди различных окон на экране. Delphi дает Вам полный доступ к сообщениям Windows и позволяет отвечать на них, как только будет нужно. В результате, опытным пользователям WinSight становится просто необходим.

WinSpector сохраняет запись о текущем состоянии машины в текстовый файл; Вы можете просмотреть этот файл для того, чтобы узнать, что именно неправильно идет в программе. Данный инструмент полезен, того когда программа находится в опытной эксплуатации - можно получить важную информацию при крушении системы.

2.2.5. Стандартные компоненты

Для дальнейшего знакомства со средой программирования Delphi потребуется рассказать о составе первой страницы Палитры Компонент.

Так на первой странице Палитры Компонент размещены 14 объектов (рис.8), определенно важных для использования. Мало кто длительное время

обойдется без кнопок, списков, окон ввода и т.д. Все эти объекты такая же часть Windows, как мышь или окно.

Набор и порядок компонент на каждой странице являются конфигурируемыми. Так, Вы можете добавить к имеющимся компонентам новые, изменить их количество и порядок.

Рис.8. Компоненты, расположенные на первой странице Палитры.

Стандартные компоненты Delphi перечислены ниже с некоторыми комментариями по их применению.

- **TMainMenu** позволяет поместить главное меню в программу. При помещении TMainMenu на форму это выглядит, как просто иконка. Иконки данного типа называют "невидимыми компонентом", поскольку они невидимы во время выполнения программы. Создание меню включает три шага: (1) помещение TMainMenu на форму, (2) вызов Дизайнера Меню через свойство Items в Инспекторе Объектов, (3) определение пунктов меню в Дизайнере Меню.
- **TPopupMenu** позволяет создавать всплывающие меню. Этот тип меню появляется по щелчку правой кнопки мыши.
- **TLabel** служит для отображения текста на экране. Вы можете изменить шрифт и цвет метки, если дважды щелкнете на свойство Font в Инспекторе Объектов. Вы увидите, что это легко сделать и во время выполнения программы, написав всего одну строчку кода.
- **TEdit** - стандартный управляющий элемент Windows для ввода. Он может быть использован для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы.
- **TMemo** - иная форма TEdit. Подразумевает работу с большими текстами. TMemo может переносить слова, сохранять в Clipboard фрагменты текста и восстанавливать их, и другие основные функции редактора. TMemo имеет ограничения на объем текста в 32Кб, это составляет 10-20 страниц. (Есть VBX и “родные” компоненты Delphi, где этот предел снят).
- **TButton** позволяет осуществить какие-либо действия при нажатии кнопки во время выполнения программы. В Delphi все делается весьма просто. Поместив TButton на форму, по двойному щелчку можно создать заготовку обработчика события нажатия кнопки. Далее нужно заполнить заготовку кодом (подчеркнуто то, что нужно написать вручную):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
MessageDlg('Areyouthere?',mtConfirmation,mbYesNoCancel,0);
```

```
end;
```

- **TCheckBox** отображает строку текста с маленьким окошком рядом. В окошке можно поставить отметку, которая означает, что что-то выбрано.

Например, если посмотреть окно диалога настроек компилятора (пункт меню Options | Project, страница Compiler), то можно увидеть, что оно состоит преимущественно из CheckBox'ов.

- **TRadioButton** позволяет выбрать только одну опцию из нескольких. Если Вы вновь откроете диалог Options | Project и выберете страницу Linker Options, то можете видеть, что секции Map file и Link buffer file состоят из наборов RadioButton.
- **TListBox** нужен для показа прокручиваемого списка. Классический пример ListBox'а в среде Windows - выбор файла из списка в пункте меню File | Open многих приложений. Названия файлов или директорий находятся в ListBox'е.
- **TComboBox** во многом напоминает ListBox, за исключением того, что позволяет водить информацию в маленьком поле ввода сверху ListBox. Есть несколько типов ComboBox, но наиболее популярен выпадающий вниз (drop-down combo box), который можно видеть внизу окна диалога выбора файла.
- **TScrollbar** - полоса прокрутки появляется автоматически в объектах редактирования, ListBox'ах при необходимости прокрутки текста для просмотра.
- **TGroupBox** используется для визуальных целей и для указания Windows, т.е. каков порядок перемещения по компонентам на форме (при нажатии клавиши TAB).
- **TPanel** - управляющий элемент, похожий на TGroupBox, используется в декоративных целях. Чтобы использовать TPanel, просто поместите его на форму и затем на него положите другие компоненты. Теперь при перемещении TPanel будут передвигаться и эти компоненты. TPanel также используется для создания линейки инструментов и окна статуса.
- **TScrollBox** представляет место на форме, которое можно скроллировать в вертикальном и горизонтальном направлениях. Пока Вы в явном виде не отключите эту возможность, форма сама по себе действует так же. Однако, могут быть случаи, когда понадобится прокручивать только часть формы. В таких случаях используется TScrollBox.

Это полный список объектов на первой странице Палитры Компонент. Если Вам нужна дополнительная информация, то выберите на Палитре объект и нажмите клавишу F1 - появится Справочник с полным описанием данного объекта.

2.2.6. Инспектор объектов

Ранее мы вкратце рассмотрели Инспектор объектов (Object Inspector).. Основное для понимания Инспектора объектов состоит в том, что он

используется для изменения характеристик любого объекта, помещенного на форма, кроме того, и для изменения свойств самой формы.

Лучший путь для изучения Инспектора объектов - поработать с ним. Для начала откроем новый проект, выбрав пункт меню File | New Project. Затем положите на форму объекты TMemo, TButton, и TListBox, как показано на рис.9.

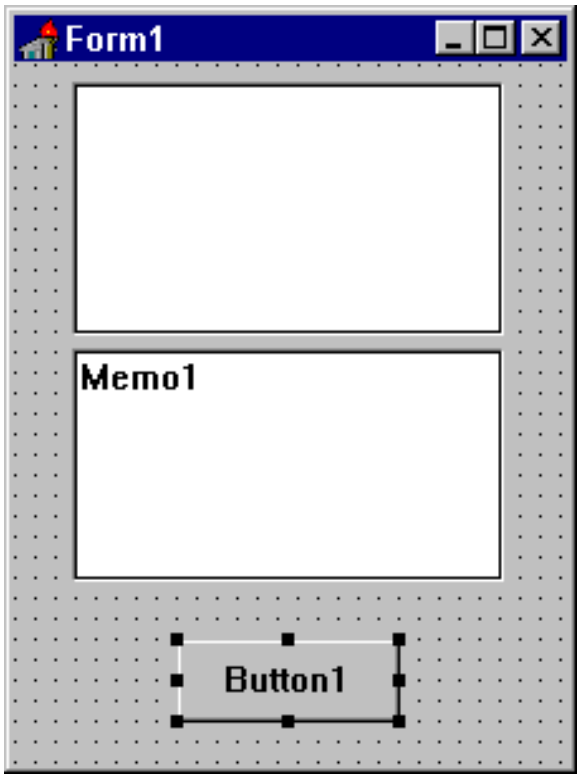


Рис.9. Простой объект TForm с компонентами TMemo, TButton, и TListBox.

Сначала рассмотрим работу со свойствами на примере свойства Ctl3D (по умолчанию включено). Выберем форму, щелкнув на ней мышкой, перейдем в Инспектор объектов и несколько раз с помощью двойных щелчков мышью переключим значение свойства Ctl3D. Можно заметить, что это действие радикально меняет внешний вид формы. Изменение свойства Ctl3D формы автоматически изменяет свойство Ctl3D каждого дочернего окна, помещенного на форму.

Вернемся на форму и поставим значение Ctl3D в True. Теперь нажмем клавишу <Shift> и щелкнем на TMemo и затем на TListBox. Теперь оба объекта имеют по краям маленькие квадратики, показывающие, что объекты выбраны.

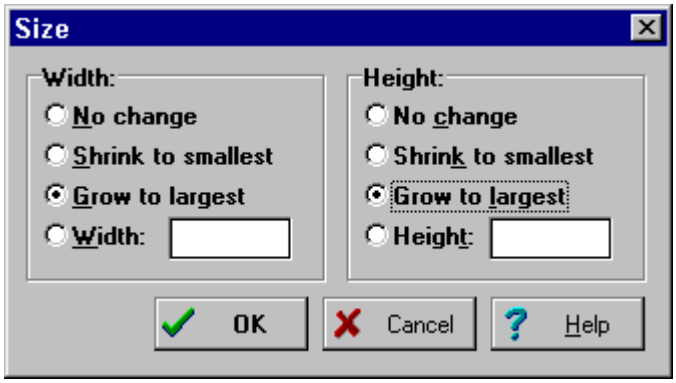


Рис.10. Пункт меню Edit дает Вам доступ к двум диалогам для выравнивания выбранного набора компонент. Первый диалог - управление размерами объектов в наборе.

Выбрав два или более объектов одновременно, Вы можете выполнить большое число операций над ними. Например, передвигать по форме. Затем попробуйте выбрать пункт меню Edit | Size и установить оба поля Ширину(Width) и Высоту(Height) в Grow to Largest**,** как показано на рис.10. Теперь оба объекта стали одинакового размера. Затем выберите пункт меню Edit | Align и поставьте в выравнивании по горизонтали значение Center (см. рис.11).

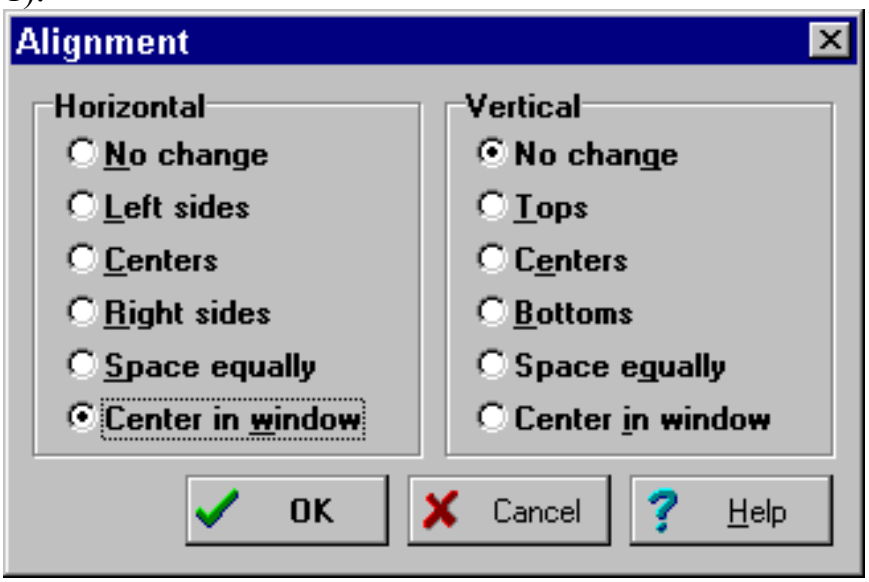


Рис.11. Диалог Alignment помогает выровнять компоненты на форме.

Поскольку Вы выбрали сразу два компонента, то содержимое Инспектора объектов изменится - он будет показывать только те поля, которые являются общими для объектов. Это означает то, что изменения в свойствах, произведенные Вами, повлияют не на один, а на все выбранные объекты. [50,51]

Рассмотрим изменение свойств объектов на примере свойства Color. Есть три способа изменить его значение в Инспекторе Объектов. Первый - просто напечатать имя цвета (clRed) или его номер. Второй путь - нажать на маленькую стрелку справа и выбрать цвет из списка. Третий путь - дважды

щелкнуть на поле ввода свойства Color. При этом появится диалог выбора цвета.

Свойство Font работает на манер свойства Color. Чтобы это посмотреть, сначала выберите свойство Font для объекта TMemo и дважды щелкните мышкой на поле ввода. Появится диалог настройки шрифта, как показано на рис.12. Выберите, например, шрифт New Times Roman и установите какой-нибудь большой размер, например 72. Затем измените цвет фонта с помощью ComboBox'а в нижнем правом углу окна диалога. Когда нажмете кнопку OK, то увидите, что вид текста в объекте TMemo радикально изменился.

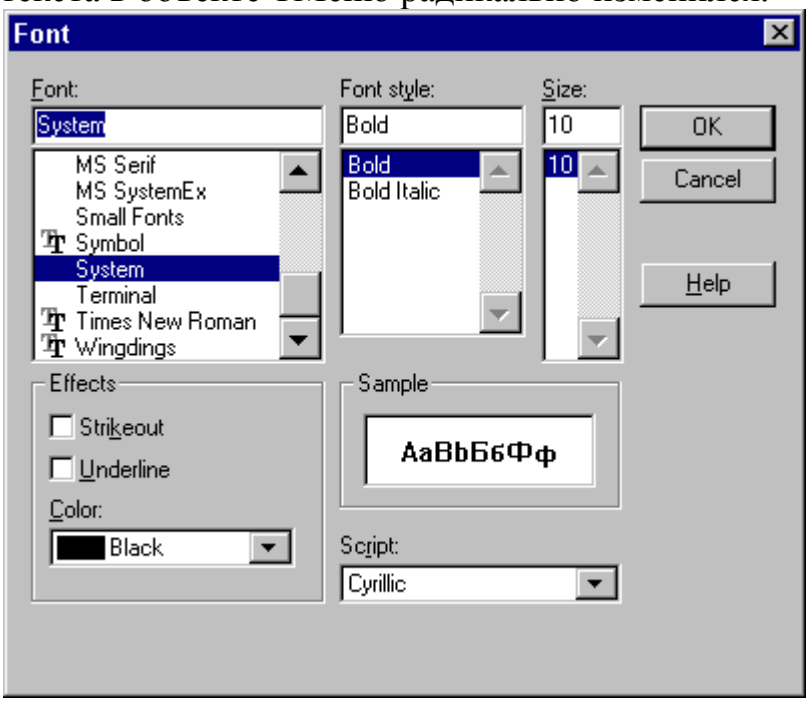


Рис.12: Диалог выбора шрифта позволяет задать тип шрифта, размер и цвет.

В завершение краткого экскурса по Инспектору Объектов дважды щелкните на свойство Items объекта ListBox. Появится диалог, в котором Вы можете ввести строки для отображения в ListBox. Напечатайте несколько слов, по одному на каждой строке и нажмите кнопку OK. Текст отобразится в ListBox'е.

2.2.7. Сохранение программы

Вы приложили некоторые усилия по созданию программы и можете захотеть ее сохранить. Это позволит загрузить программу позже и вновь с ней поработать.

Первый шаг - создать поддиректорию для программы. Лучше всего создать директорию, где будут храниться все Ваши программы и в ней - создать поддиректорию для данной конкретной программы. Например, Вы можете создать директорию MYCODE и внутри нее - вторую директорию TIPS1, которая содержала бы программу, над которой Вы только что работали.

После создания поддиректории для хранения Вашей программы нужно выбрать пункт меню File | Save Project. Сохранить нужно будет два файла. Первый - модуль (unit), над которым Вы работали, второй - главный файл проекта, который "владеет" Вашей программой. Сохраните модуль под именем MAIN.PAS и проект под именем TIPS1.DPR. (Любой файл с расширением PAS и словом “unit” в начале является *модулем*.)

2.2.8.TButton, исходный текст, заголовки и *Z*-упорядочивание. Тьюторы.

Еще несколько возможностей Инспектора Объектов и Дизайнера Форм.

Создайте новый проект. Поместите на форму объект TMemo, а затем TEdit так, чтобы он наполовину перекрывал TMemo, как показано на рис.13. Теперь выберите пункт меню Edit | Send to Back, что приведет к перемещению TEdit вглубь формы, за объект TMemo. Это называется изменением *Z*-порядка компонент. Буква *Z* используется потому, что обычно математики обозначают третье измерение буквой *Z*. Так, *X* и *Y* используются для обозначения ширины и высоты, а *Z* - для обозначения глубины.

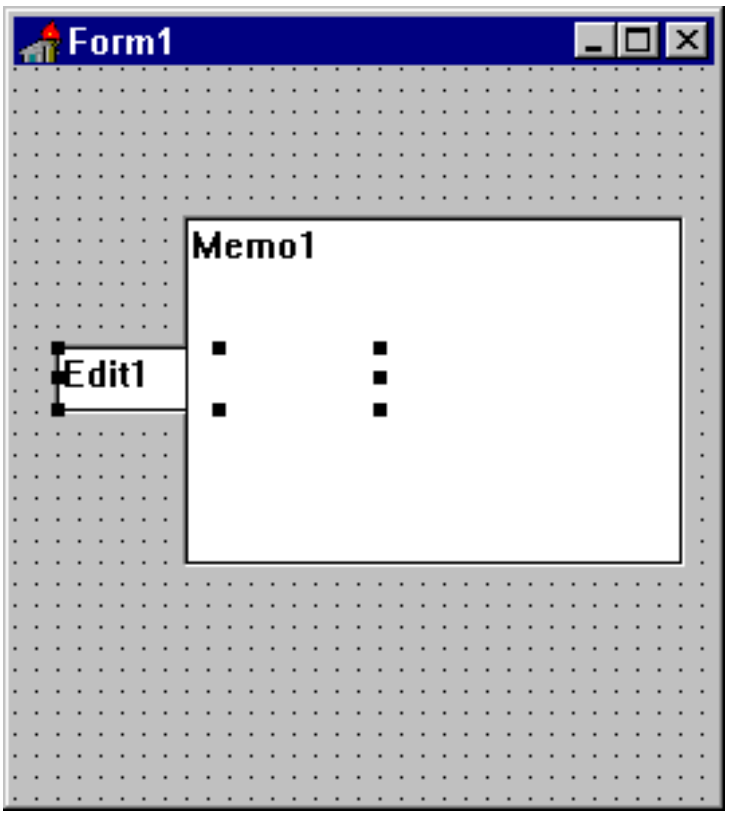


Рис.13. Объект TEdit перекрывается наполовину объектом TMemo.

Если Вы “потеряли” на форме какой-то объект, то найти его можно в списке Combobox’а, который находится в верхней части Инспектора Объектов.

Поместите кнопку TButton в нижнюю часть формы. Теперь растяните Инспектор Объектов так, чтобы свойства Name и Caption были видны одновременно на экране. Теперь измените имя кнопки на Terminate. Заметьте, что заголовок (Caption) поменялся в тот же момент. Такое двойное изменение наблюдается только если ранее не изменялось свойство Caption.

Текст, который Вы видите на поверхности кнопки - это содержимое свойства Caption, свойство Name служит для внутренних ссылок, Вы будете использовать его при написании кода программы. Если Вы откроете сейчас окно Редактора, то увидите следующий фрагмент кода:

```
TForm1 = class(TForm)
Edit1: TEdit;
Memo1: TMemo;
Terminate: TButton;
private
{ Private declarations }
public
{ Public declarations }
end;
```

В этом фрагменте кнопка TButton называется Terminate из-за того, что Вы присвоили это название свойству Name. Заметьте, что TMemo имеет имя, которое присваивается по умолчанию.

Перейдите на форму и дважды щелкните мышкой на объект TButton. Вы сразу попадете в окно Редактора, в котором увидите фрагмент кода вроде этого:

```
procedure TForm1.TerminateClick(Sender: TObject);
begin

end;
```

Данный код был создан автоматически и будет выполняться всякий раз, когда во время работы программы пользователь нажмет кнопку Terminate. Вдобавок, Вы можете видеть, что определение класса в начале файла теперь включает ссылку на метод TerminateClick:[51]

```
 TForm1 = class(TForm)
Edit1: TEdit;
Memo1: TMemo;
Terminate: TButton;
procedure TerminateClick(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
```

Изначально Вы смотрите на кнопку на форме. Вы делаете двойной щелчок на эту кнопку, и соответствующий фрагмент кода автоматически заносится в Редактор.

Теперь самое время написать строчку кода. Это весьма простой код, состоящий из одного слова Close:

```
procedure TForm1.TerminateClick(Sender: TObject);
begin
 Close;
```

Когда этот код исполняется, то главная форма (значит и все приложение) закрывается. Для проверки кода запустите программу и нажмите кнопку Terminate. Если все сделано правильно, программа закроется и Вы вернетесь в режим дизайна.

Прежде, чем перейти к следующему разделу, перейдите в Инспектор Объектов и измените значение свойства Name для кнопки на любое другое, например OK. Нажмите Enter для внесения изменений. Посмотрите в Редактор, Вы увидите, что код, написанный Вами изменился:

```
procedure TForm1.OkClick(Sender: TObject);
begin
 Close;
end;
```

Заметьте, что аналогичные изменения произошли и в определении класса:

```
 TForm1 = class(TForm)
Edit1: TEdit;
Memo1: TMemo;
Ok: TButton;
procedure OkClick(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;
```

Тьюторы (интерактивные обучающие программы)

Delphi предоставляет тьютор, содержащий несколько тем и который можно запустить из пункта меню Help | Interactive Tutors. Тьютор запускается только если среда Delphi имеет все установки по умолчанию. Если конфигурация была изменена, то проще всего сохранить файл DELPHI.INI под другим именем и скопировать файл DELPHI.CBT в DELPHI.INI.

2.3. Управление проектом Delphi

Любой проект имеет, покрайней мере, шесть связанных с ним файлов. Три из них относятся к управлению проектом из среды и напрямую программистом не меняются. [22,25,52] Вот эти файлы :

- Главный файл проекта, изначально называется PROJECT1.DPR.
- Первый модуль программы /unit/, который автоматически появляется в начале работы. Файл называется UNIT1.PAS по умолчанию, но его можно назвать любым другим именем, вроде MAIN.PAS.
- Файл главной формы, который по умолчанию называется UNIT1.DFM, используется для сохранения информации о внешнем виде главной формы.
- Файл PROJECT1.RES содержит иконку для проекта, создается автоматически.
- Файл, который называется PROJECT1.OPT по умолчанию, является текстовым файлом для сохранения установок, связанных с данным проектом. Например, установленные Вами директивы компилятора сохраняются здесь.
- Файл PROJECT1.DSK содержит информацию о состоянии рабочего пространства.

Разумеется, если сохранить проект под другим именем, то изменят название и файлы с расширением RES, OPT и DSK.

После компиляции программы получаются файлы с расширениями:

DCU - скомпилированные модули

EXE - исполняемый файл

DSM - служебный файл для запуска программы в среде, весьма большой, его рекомендуется стирать при окончании работы.

~PA, ~DP - backup файлы Редактора.

2.3.1.Пункт меню “File”

Если нужно сохранить проект, то необходимо выбирать пункт главного меню “File” (с помощью мышки или по клавише Alt+F). Пункт меню “File” выглядит следующим образом:

New Project
Open Project
Save Project
Save Project As
Close Project

New Form
New Unit
New Component
Open File
Save File
Save File As

Close File

Add File
Remove File

Print

Exit
---
1 PREV1.DPR
2 PREV2.DPR

Здесь есть шесть секций, вот их назначение:[50,52]

- Первая секция позволяет возможность управлять проектом в целом.
- Вторая секция дает контроль над формами, модулями и компонентами проекта.
- Третья позволяет добавлять и удалять файлы из проекта.
- Четвертая управляет печатью.
- Пятая секция - выход из Delphi
- Шестая секция предоставляет список ранее редактировавшихся проектов; Вы можете быстро открыть нужный.

Как Вы увидите позже, большинство операций из пункта меню “File” можно выполнить с помощью Менеджера Проекта (Project Manager), который можно вызвать из пункта меню View. Некоторые операции доступны и через SpeedBar. Данная стратегия типична для Delphi: она предоставляет несколько путей для решения одной и той же задачи, Вы сами можете решить, какой из них более эффективен в данной ситуации.

Каждая строка пункта меню “File” объяснена в Справочнике. Выберите меню “File” и нажмите F1, появится экран справочника, как на рис.1.

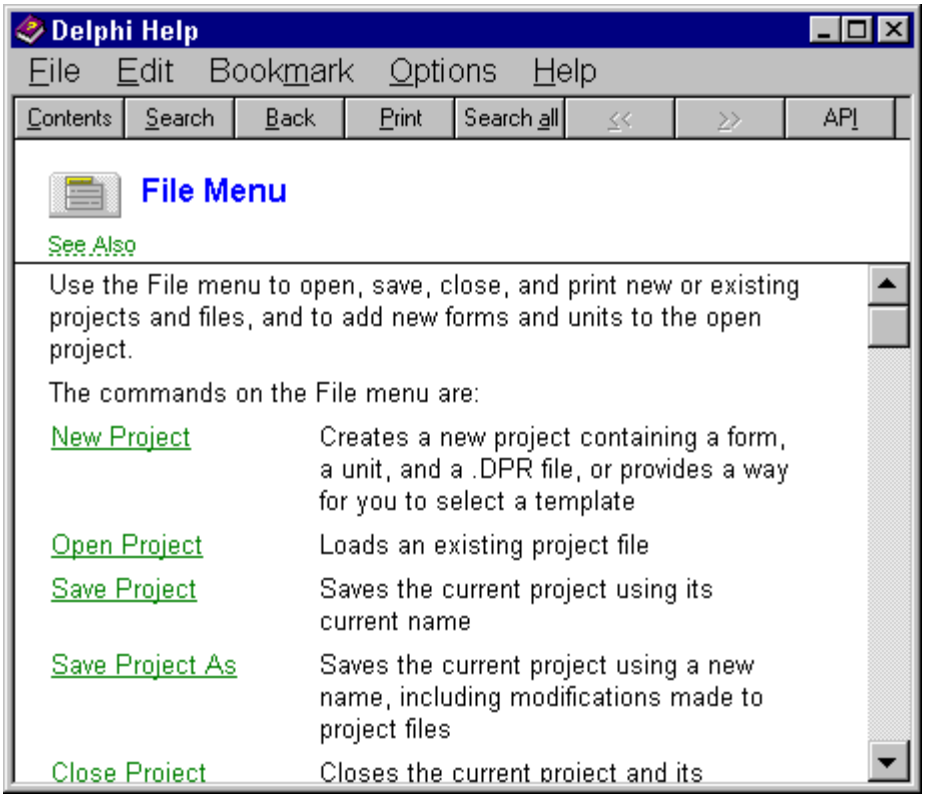


Рис.1. Delphi включает подсказку о том, как использовать пункт меню “File”.

Большинство из пунктов первой секции очевидны. “New Project” начинает новый проект, “Open Project” открывает существующий проект и т.д.

Первые два пункта второй секции позволяют Вам создать новую форму или новый модуль. Выбирая “New Form”, Вы создаете новую форму и модуль, связанный с ней; выбирая “New Unit”, Вы создаете один модуль.

“New Component” вызывает диалог для построения заготовки нового визуального компонента. В результате создается модуль, который можно скомпилировать и включить в Палитру Компонент.

“Open File” открывает, при необходимости, любой модуль или просто текстовый файл. Если модуль описывает форму, то эта форма тоже появится на экране.

При создании нового модуля Delphi дает ему имя поумолчанию. Вы можете изменить это имя на что-нибудь более осмысленное (например, MAIN.PAS) с помощью пункта “Save File As“.[22]

“Save File” сохраняет только редактируемый файл, но не весь проект.

“Close File” удаляет файл из окна Редактора.

Нужно обратить внимание: Вы должны регулярно сохранять проект через File | Save Project либо через нажатие Ctrl+S.

2.3.2. Управление проектом

Менеджер Проектов, на рис.3, разделен на две части: верхняя - панель с управляющими кнопками: нижняя - список модулей, входящих в проект.

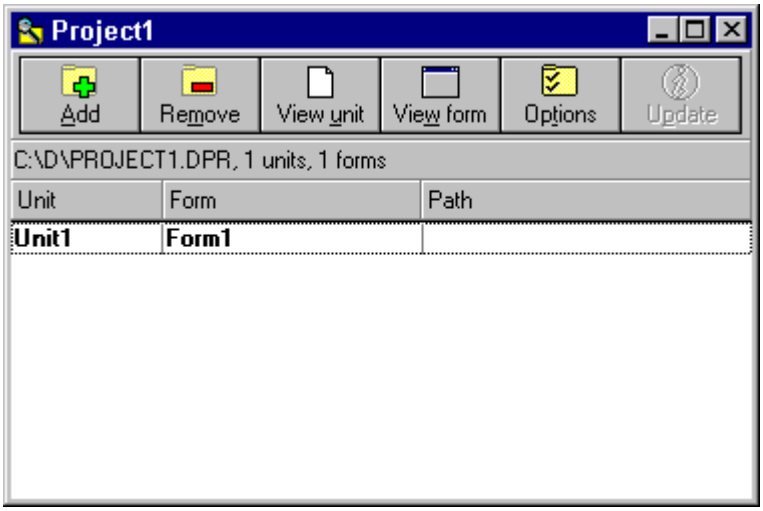


Рис.3. Кнопки сверху используются для удаления и добавления модулей в проект.

Вы можете использовать кнопки с плюсом и минусом для добавления и удаления файлов в проекте. Эти изменения влияют на файлы с исходным текстом, т.е. если добавить в проект модуль, то ссылка на него появится в файле с расширением DPR.

Краткое описание других кнопок :

- Третья слева кнопка - просмотр текста модуля, на котором стоит курсор.
- Четвертая - просмотр формы, если таковая имеется для данного модуля
- Пятая - вызов диалога настройки проекта, сам диалог будет рассмотрен позже.
- Последняя - сохранение изменений на диске.

Обзор других пунктов меню

Пункт меню “Edit”

“Edit” содержит команды “Undo” и “Redo”, которые могут быть весьма полезны при работе в редакторе для устранения последствий при неправильных действии, например, если случайно удален нужный фрагмент текста.

Отметьте для себя, что Справочник (on-line help) объясняет как нужно использовать пункт меню Options | Environment для настройки команды “Undo”. Возможность ограничить возможное количество команд “Undo” может пригодиться, если Вы работаете на машине с ограниченными ресурсами.

Команды “Cut”, “Copy”, “Paste” и “Delete” – как и во всех остальных приложениях Windows, но их можно применять не только к тексту, но и к визуальным компонентам.

Пункт меню “Menu”

В “Search” есть команда “Find Error” (поиск ошибки), которая поможет отследить ошибку периода выполнения программы. Когда в сообщении об ошибке указан ее адрес, Вы можете выбрать пункт меню Search | Find Error и ввести этот адрес. Если это представится возможным, то среда переместит Вас в то место программы, где произошла ошибка.

Пункт меню “View”

Составляющие пункта меню “View”:

- Project Manager (Менеджер Проекта).
- Project Source - загружает главный файл проекта (DPR) в Редактор
- Установка показывать или нет Object Inspector на экране.
- Установка показывать или нет Alignment Palette. То же самое доступно из пункта меню Edit | Align.
- Browser - вызов средства для просмотра иерархии объектов программы, поиска идентификатора в исходных текстах и т.п.
- Watch, Breakpoint и Call Stack - связаны с процедурой отладки программы и будут обсуждаться позднее.
- Component List - список компонент, альтернатива Палитре Компонент. Используется для поиска компонента по имени или при отсутствии мыши.
- Window List - список окон, открытых в среде Delphi.
- Toggle Form/Unit, Units, Forms - переключение между формой и соответствующим модулем, выбор модуля или формы из списка.
- New Edit Window - открывает дополнительное окно Редактора. Полезно, если нужно, например, просмотреть две разные версии одного файла.
- SpeedBar и Component Palette - установки, нужно ли их отображать.

Пункт меню “Compile”

В пункте меню “Compile” проект можно скомпилировать (compile) или перестроить (build). Если выбрать Compile или Run, то Delphi перекомпилирует только те модули, которые изменились со времени последней компиляции. Build all, с другой стороны, перекомпилирует все модули, исходные тексты которых доступны. Команда Syntax Check только проверяет правильность кода программы, но не обновляет DCU файлы.

В самом низу - пункт Information, который выдает информацию о программе: размеры сегментов кода, данных и стека, размер локальной динамической памяти и количество скомпилированных строк.

Пункт меню “Run”

Можно использовать “Run” для компиляции и запуска программы а также для указания параметров командной строки для передачи в программу. Здесь же имеются опции для режима отладки.

Пункт меню Options | Project

“Options” наиболее сложная часть системного меню. Это центр управления, из которого вы можете изменять установки для проекта и всей рабочей среды Delphi. В “Options” имеются семь пунктов:

Project
Environment
Tools
Gallery
--
Open Library
Install Components

Rebuild Library

Первые четыре пункта вызывают диалоговые окна. Ниже приведено общее описание пункта меню “Options”:

- Project - выбор установок, которые напрямую влияют на текущий проект, это могут быть, к примеру, директивы компилятора проверки стека (stack checking) или диапазона (range checking).
- Environment - конфигурация самой среды программирования (IDE). Например, здесь можно изменить цвета, используемые в Редакторе.
- Tools - позволяет добавить или удалить вызов внешних программ в пункт главного меню “Tools”. Например, если Вы часто пользуетесь каким-нибудь редактором или отладчиком, то здесь его вызов можно добавить в меню.
- Gallery - позволяет определить специфические установки для Эксперта Форм и Эксперта Проектов и их “заготовок”. Эксперты и “заготовки” предоставляют путь для ускорения конструирования интерфейса программы.
- Последние три пункта позволяют сконфигурировать Палитру Компонент.

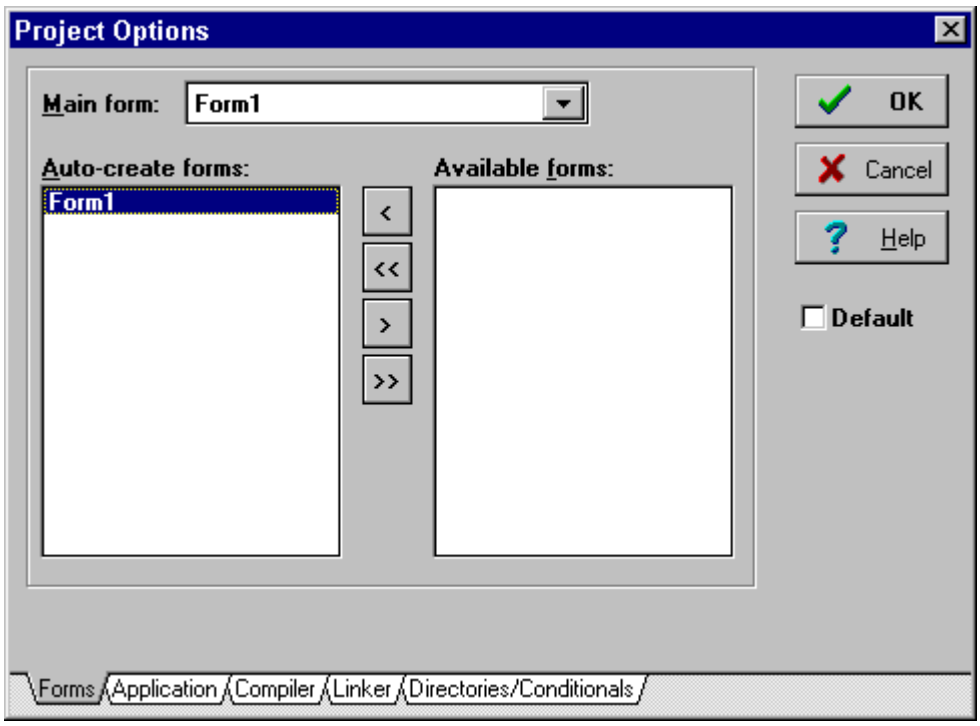
Диалог из пункта Options | Project включает пять страниц:

- На странице Forms перечислены все формы, включенные в проект; Вы можете указать, нужно ли автоматически создавать форму при старте программы или Вы ее создадите сами.
- На странице Application Вы определяете элементы программы, такие, как заголовок, файл помощи и иконка.
- Страница Compiler включает установки для генерации кода, управления обработкой ошибок времени выполнения, синтаксиса, отладки и др.
- На странице Linker можно определить условия для процесса линковки приложения
- Страница Directories/Conditionals - здесь указываются директории, специфичные для данного проекта.

После предыдущего абзаца с общим описанием каждая страница описана детально в отдельной главе.

Все установки для проекта сохраняются в текстовом файле с расширением OPT и Вы можете вручную их исправить.

Страница Forms



На странице Forms можно выбрать главную форму проекта. Изменения, которые Вы сделаете, отобразятся в соответствующем файле DPR. Например, в нижеследующем проекте, Form1 является главной, поскольку появляется первой в главном блоке программы:

```
program Project1;
uses
 Forms,
 Unit1 in 'UNIT1.PAS' {Form1},
 Unit2 in 'UNIT2.PAS' {Form2};
{$R *.RES}
begin
 Application.CreateForm(TForm1, Form1);
 Application.CreateForm(TForm2, Form2);
 Application.Run;
end.
```

Если изменить код так, чтобы он читался

```
begin
 Application.CreateForm(TForm2, Form2);
 Application.CreateForm(TForm1, Form1);
 Application.Run;
end.
```

то теперь Form2 станет главной формой проекта.

Вы также можете использовать эту страницу для определения, будет ли данная форма создаваться автоматически при старте программы. Если форма создается не автоматически, а по ходу выполнения программы, то для этого следует использовать процедуру Create.

Кстати, в секции **Uses** имя формы в фигурных скобках является существенным для Менеджера Проектов и удалять его не стоит. Не нужно вообще ничего изменять вручную в файле проекта, если только Вы не захотели создать DLL, но об этом позднее.

Страница Applications

На странице Applications, (см. рис.5), вы можете задать заголовок (Title), файл помощи (Help file) и пиктограмму (Icon) для проекта.

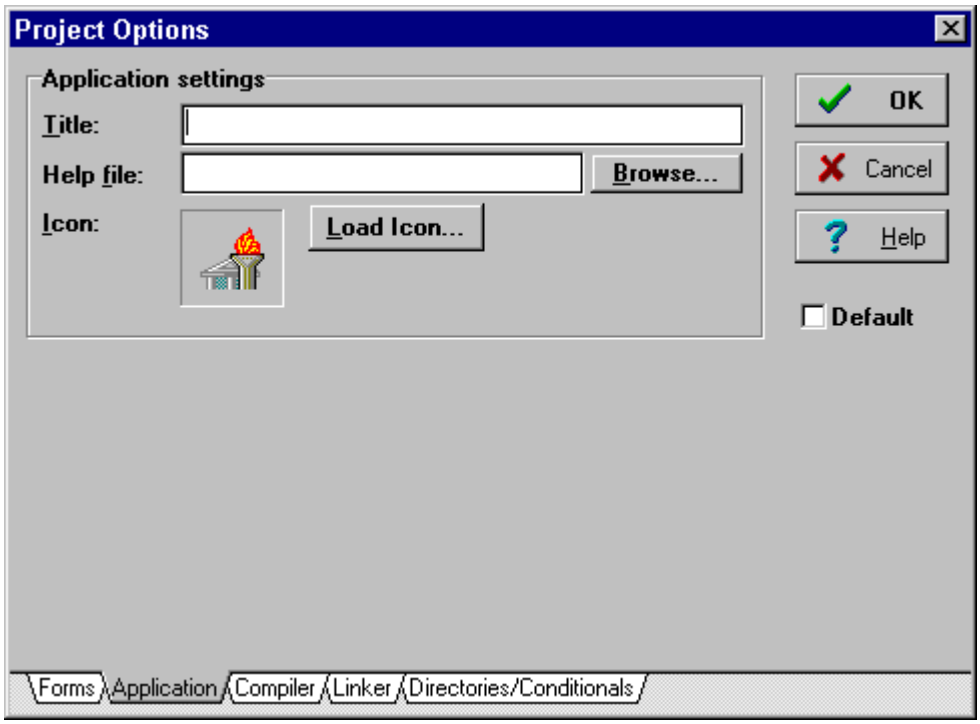


Рис.5. Страница общих установок для приложения.
Страница Compiler.

Ранее уже отмечалось, что установки из пункта меню “Options | Project” сохраняются в соответствующем файле с расширением OPT. Давайте рассмотрим директивы компилятора на странице Compiler (рис.6).

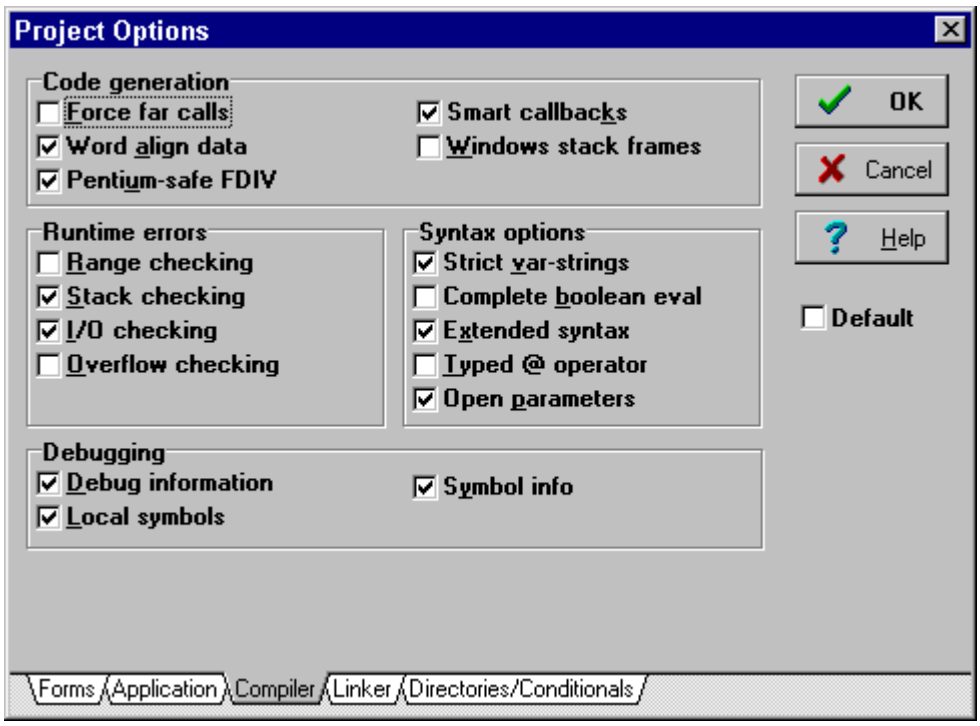


Рис.6. Страница для определения директив компилятора.

Следующая таблица показывает, как различные директивы отображаются в OPT файле, на странице Compiler и внутри кода программы:

OPT File	Options Page	Editor Symbol
F	Force Far Calls	{$F+}
A	Word Align Date	{$A+}
U	Pentium-Safe FDIV	{$U+}
K	Smart Callbacks	{$K+}
W	Windows (3.0) Stack Frame	{$W+}
R	Range Checking	{$R+}
S	Stack Checking	{$S+}
I	IO Checking	{$I+}
Q	Overflow Checking	{$Q+}
V	Strict Var Strings	{$V+}
B	Complete Boolean Evaluation	{$B+}
X	Extended Syntax	{$X+}
T	Typed @ Operator	{$T+}
P	Open Parameters	{$P+}
D	Debug Information	{$D+}
L	Local Symbols	{$L+}
Y	Symbol Information	{$Y+}
N	Numeric Processing	{$N+}

Страница Linker

Теперь давайте перейдем к странице Linker, проиллюстрированной на рис.7.

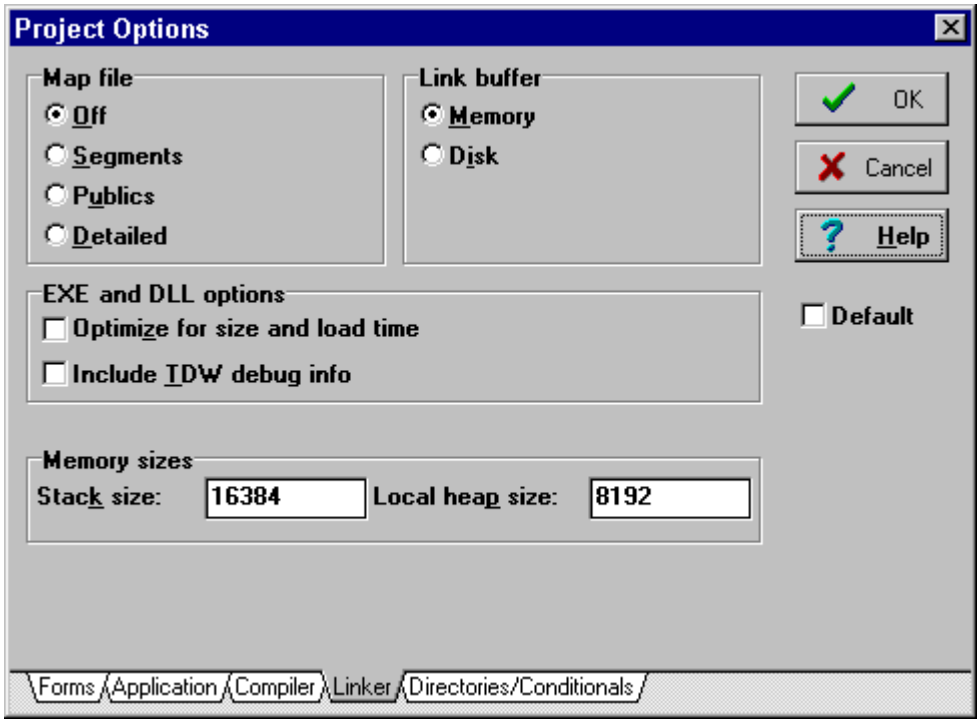


Рис.7. Страница линковщика.

Установки отладчика рассматриваются ниже. Если буфер линковщика расположен в памяти, то линковка происходит быстрее.

Размер стека (Stack Size) и локальной динамической памяти (Heap Size) весьма важны. Delphi устанавливает по умолчанию и Stack Size, и Heap Size в 8192 байт каждый. Вам может понадобиться изменить размер стека в программе, но обычно это не более 32Кб. В сумме эти два размера не должны превышать 64Кб, иначе будет выдаваться ошибка при компиляции программы.

Страница Directories/Conditionals

Страница Directories/Conditionals, (рис.8), дает возможность расширить число директорий, в которых компилятор и линковщик ищут DCU файлы.

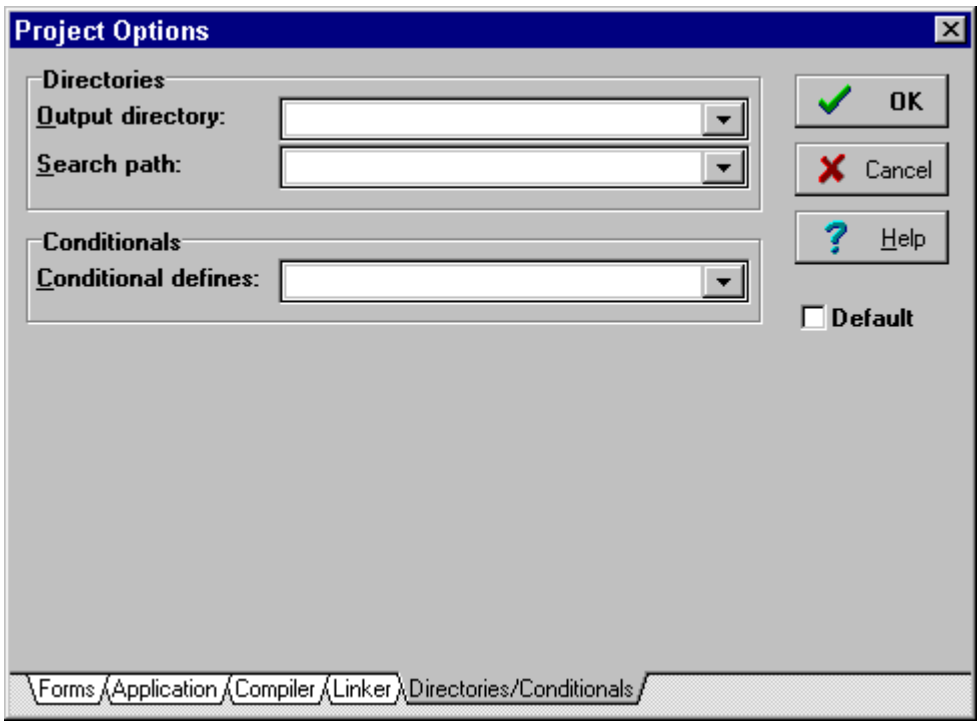


Рис.8. Страница Directories/Conditionals.

В файле DELPHI.INI содержится еще один список директорий. Запомните, что в OPT файле - список директорий для конкретного проекта, а в файле DELPHI.INI - список относится к любому проекту.

Output directory - выходная директория, куда складываются EXE и DCU файлы, получающиеся при компиляции.

Search path - список директорий для поиска DCU файлов при линковке. Директории перечисляются через точку с запятой ;

Conditional defines - для опытного программиста и на первом этапе создания проекта не требуется. Для информации можно вызвать Справочник (on-line help).[50,51,52]

Конфигурация среды программирования (IDE)

Пункт меню “Options | Environment” предоставляет Вам большой набор страниц и управляющих элементов, которые определяют внешний вид и работу IDE. Delphi позволяет сделать следующие важные настройки:

1. Определить, что из проекта будет сохраняться автоматически.
2. Можно менять цвета IDE.
3. Можно менять подсветку синтаксиса в Редакторе.
4. Можно изменить состав Палитры Компонент.
5. Указать “горячие клавиши” IDE.

Первая страница пункта меню “Options | Environment” показана на рис.9.

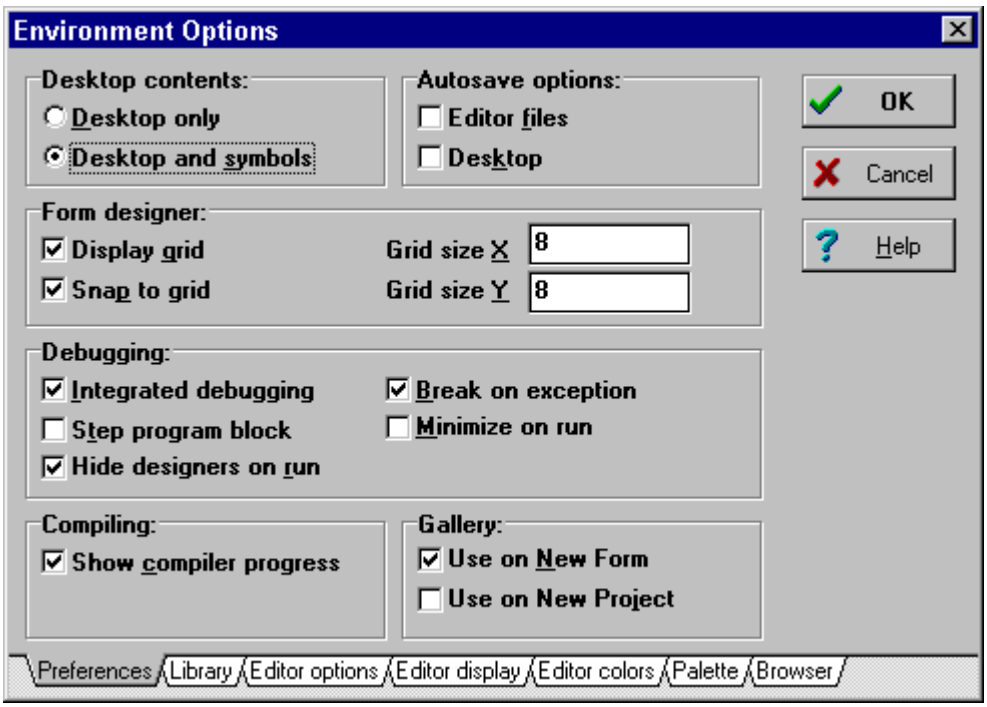


Рис.9. Страница Preferences.

В группе “Desktop Contents” определяется, что будет сохраняться при выходе из Delphi. Если выбрать Desktop Only - это сохранит информацию о директориях и открытых окнах, если выбрать Desktop And Symbols - это сохранит то же самое плюс информацию для броузера (browser).

В группе “Autosave” указывается, что нужно сохранять при запуске программы. Если выбрана позиция Editor Files, то сохраняются все модифицированные файлы из Редактора при выполнении команд Run|Run, Run|Trace Into, Run|Step Over, Run|Run To Cursor или при выходе из Delphi. Если выбрана позиция Desktop - сохраняется рабочая среда при закрытии проекта или при выходе из Delphi. Если Вы позже откроете проект, то он будет иметь тот же вид, что и при его закрытии.

В группе “Form Designer” можно установить, показывать ли сетку (grid) на экране и выравнивать ли объекты по ней, и размер ячеек сетки.

В группе “Debugging”: опция Integrated Debugging - использовать ли встроенный отладчик; Step Program Block - отладчик остановится на первой строке модуля, в котором есть отладочная информация; Break On Exception - останавливать ли программу при возникновении исключительной ситуации; Minimize On Run - свертывать ли Delphi при запуске программы. После закрытия программы среда Delphi восстанавливается. Hide Designers On Run - прячет окна Дизайнера (Инспектор Объектов, формы) при запуске приложения.

Show Compiler Progress - показывать ли окно, в котором отражается процесс компиляции программы.

“Gallery” - указывает, в каких случаях нужно предоставлять “галерею” (коллекцию заготовок и экспертов).

Страницы Editor Options, Editor Display и Editor Colors позволяют Вам изменить цвета и “горячие” клавиши, используемые IDE. Страница Editor Display показана на рис.10, а Editor Colors - на рис.11.

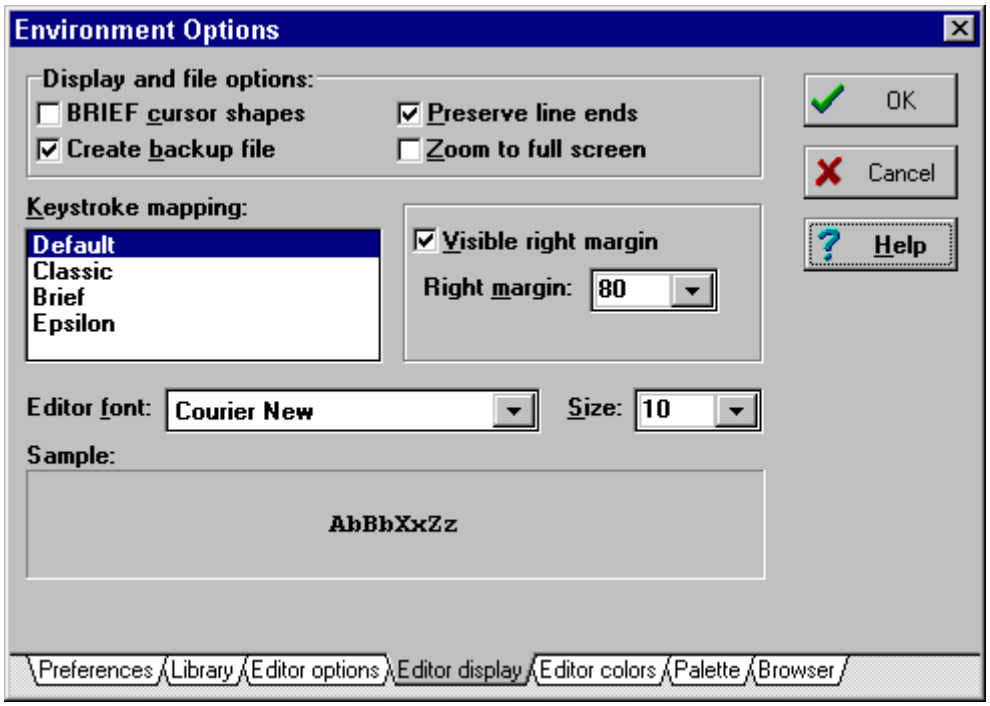


Рис.10. Страница Editor Display.

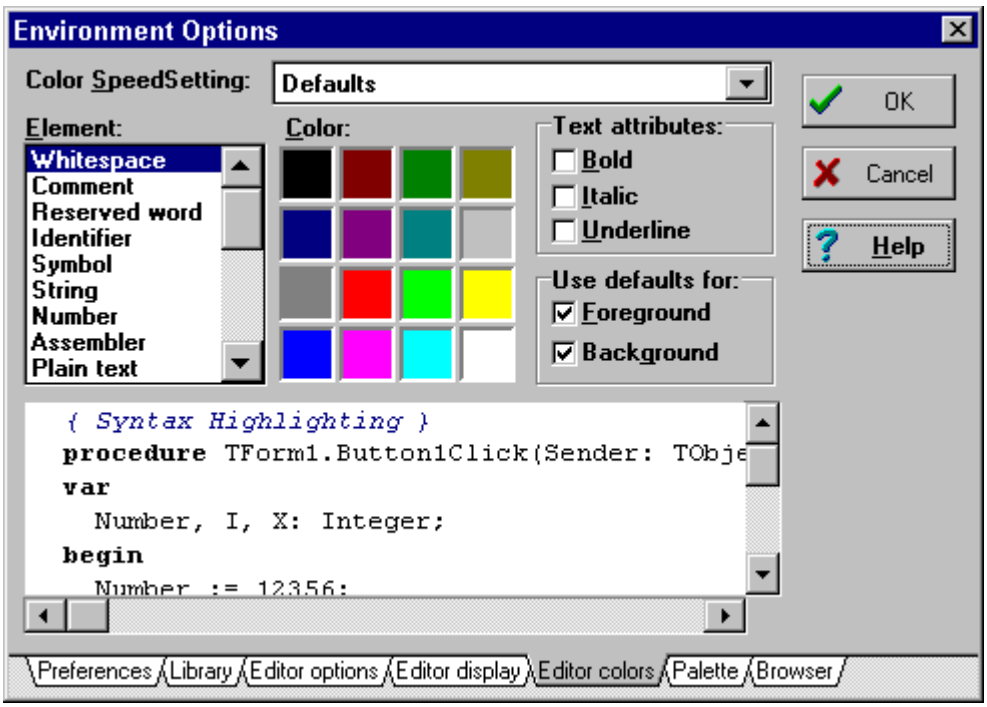


Рис.11. Страница Editor Colors.

Существует несколько способов изменить назначение “горячих” клавиш, используемых Редактором. Например, многие пользователи привыкли, что по клавише F5 максимизируется окно Редактора. Для этого им надо использовать расположение клавиш, называемое “Classic” (Keystroke mapping : Classic). Всего есть четыре вида конфигурации клавиш:

- “Default” - характерно для Microsoft. Если Вы новичок в Windows или уже привыкли к этому расположению клавиш, то это подойдет.

- “Classic” - более известно ветеранам Borland C++ и Borland Pascal. Поддерживает многие комбинации клавиш WordStar и отладчик управляется старым добрым способом.
- Остальные два вида - имитируют редакторы Epsilon и BRIEF. Подойдут, если вы с ними знакомы.

Точное описание назначения клавиш можно найти в Справочнике (в Help | Topic Search набрать “key mapping”).

Цвета IDE можно изменить на странице Editor Colors.

И, наконец, Editor Options (рис.12).

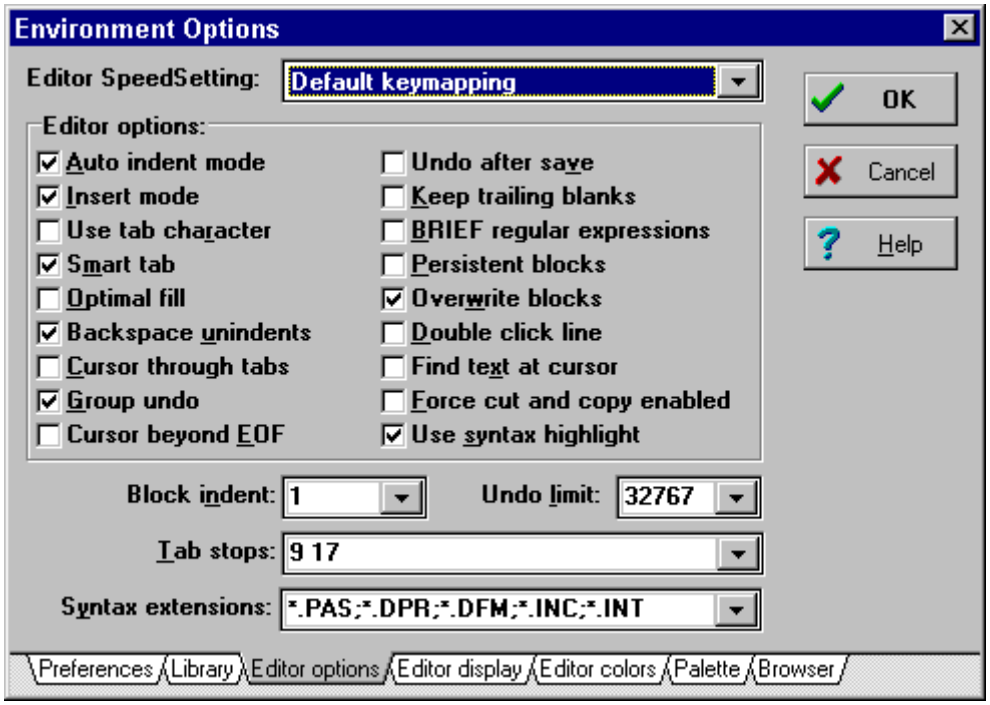


Рис.12. На странице Editor Options можно настроить тонкие детали работы Редактора.

Многие из установок на данной странице не весьма важны для большинства пользователей, поэтому остановимся лишь на некоторых.

“Use syntax highlight” - выделять ли цветом синтаксические конструкции в Редакторе Исходного текста.

“Find text at cursor” - если включено, то при поиске (Ctrl+F) в качестве подстроки для поиска будет браться то слово, на котором стоит курсор.

Обо всех опциях можно подробнее узнать в Справочнике (F1).

Установки сохраняются в файле DELPHI.INI, который находится в директории Windows.

2.4. Обзор Палитры Компонентов

На первой странице Палитры Компонент размещены 14 объектов (рис.1) определенно важных для использования. Мало кто обойдется длительное время без кнопок, списков, окон ввода и т.д. Все эти объекты такая же часть Windows, как мышь или окно.

Набор и порядок компонент на каждой странице являются конфигурируемыми. Так, к имеющимся компонентам Вы можете добавить новые, изменить их количество и порядок. Это можно сделать, вызвав всплывающее меню (нажать правую кнопку мыши, когда указатель над Палитрой).

Рис.1. Компоненты, расположенные на первой странице Палитры.

Стандартные компоненты Delphi перечислены ниже с некоторыми комментариями по их применению.

Курсор - не компонент, просто пиктограмма для быстрой отмены выбора какого-либо объекта.

TMainMenu позволяет Вам поместить главное меню в программу. При помещении TMainMenu на форму это выглядит, как просто иконка. Иконки данного типа называют "невидимыми компонентом", поскольку они невидимы во время выполнения программы. Создание меню включает три шага: (1) помещение TMainMenu на форму, (2) вызов Дизайнера Меню через свойство Items в Инспекторе Объектов, (3) определение пунктов меню в Дизайнере Меню.

TPopupMenu позволяет создавать всплывающие меню. Этот тип меню появляется по щелчку правой кнопки мыши на объекте, к которому привязано данное меню. У всех видимых объектов имеется свойство PopupMenu, где и указывается нужное меню. Создается PopupMenu аналогично главному меню.

TLabel служит для отображения текста на экране. Вы можете изменить шрифт и цвет метки, если дважды щелкнете на свойство Font в Инспекторе объектов. Вы увидите, что это легко сделать и во время выполнения программы, написав всего одну строчку кода.

TEdit - стандартный управляющий элемент Windows для ввода. Он может быть использован для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы.

TMemo - иная форма TEdit. Подразумевает работу с большими текстами. TMemo может переносить слова, сохранять в ClipBoard

фрагменты текста и восстанавливать их, и другие основные функции редактора. TMemo имеет ограничения на объем текста в 32Кб, это составляет 10-20 страниц. (Есть VBX и “родные” компоненты Delphi, где этот предел снят).

TButton позволяет выполнить какие-либо действия при нажатии кнопки во время выполнения программы. В Delphi все делается весьма просто. Поместив TButton на форму, Вы по двойному щелчку можете создать заготовку обработчика события нажатием кнопки. Далее нужно заполнить заготовку кодом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
MessageDlg('Are you there?',mtConfirmation,mbYesNoCancel,0);
end;
```

TCheckBox отображает строку текста с маленьким окошком рядом. В окошке можно поставить отметку, которая означает, что что-то выбрано. Например, если посмотреть окно диалога настроек компилятора (пункт меню Options | Project, страница Compiler), то можно увидеть, что оно преимущественно состоит из CheckBox’ов.

TRadioButton позволяет выбрать только одну опцию из нескольких. Если Вы вновь откроете диалог Options | Project и выберете страницу Linker Options, то можете увидеть, что секции Map file и Link buffer file состоят из наборов RadioButton.

TListBox нужен для показа прокручиваемого списка. Классический пример ListBox’а в среде Windows - выбор файла из списка в пункте меню File | Open многих приложений. Названия файлов или директорий и находятся в ListBox’е.

TComboBox во многом напоминает ListBox, за исключением того, что позволяет вводить информацию в маленьком поле ввода сверху ListBox. Есть несколько типов ComboBox, но наиболее популярен спадающий вниз (drop-down combo box), который можно видеть внизу окна диалога выбора файла.

TScrollbar - полоса прокрутки появляется автоматически в объектах редактирования, ListBox’ах при необходимости прокрутки текста для просмотра.

TGroupBox используется для визуальных целей и указания Windows, каков порядок перемещения по компонентам на форме (при нажатии клавиши TAB).

TRadioGroup используется аналогично TGroupBox, для группировки объектов TRadioButton.

TPanel - управляющий элемент, похожий на TGroupBox, используется в декоративных целях. Чтобы использовать TPanel, просто поместите его на форму, а затем на него положите другие компоненты. Теперь при перемещении TPanel будут передвигаться и эти компоненты. TPanel используется также для создания линейки инструментов и окна статуса.

Это полный список объектов на первой странице Палитры Компонент. Если Вам нужна дополнительная информация, то выберите на Палитре объект и нажмите клавишу F1 - появится Справочник с полным описанием данного объекта.

2.4.1. Страница Additional

На странице Standard представлены управляющие элементы, появившиеся в Windows 3.0. На странице Additional размещены объекты, позволяющие создать более красивый пользовательский интерфейс программы. [50,52]

Список компонент:

TBitBtn - кнопка вроде TButton, однако на ней можно разместить картинку (glyph). TBitBtn имеет несколько предопределенных типов (bkClose, bkOK и др.), при выборе которых кнопка принимает соответствующий вид. Кроме того, нажатие кнопки на модальном окне (Form2.ShowModal) приводит к закрытию окна с соответствующим модальным результатом (Form2.ModalResult).

TSpeedButton - кнопка для создания панели быстрого доступа к командам (SpeedBar). Пример - SpeedBar слева от Палитры Компонент в среде Delphi. Обычно на данную кнопку помещается только картинка (glyph).

TTabSet - горизонтальные закладки. Обычно используются вместе с TNoteBook для создания многостраничных окон. Название страниц можно задать в свойстве Tabs, но проще это сделать в программе при создании формы (OnCreate) :

```
TabSet1.Tabs := Notebook1.Pages;
```

Для того, чтобы при выборе закладки страницы перелистывались, нужно в обработчике события OnClick для TTabSet написать:

```
Notebook1.PageIndex := TabSet1.TabIndex;
```

TNoteBook - используется для создания многостраничного диалога, на каждой странице располагается свой набор объектов. Используется совместно с TTabSet.

TTabbedNotebook - многостраничный диалог со встроенными закладками, в данном случае - закладки сверху.

TMaskEdit - аналог TEdit, но с возможностью форматированного ввода. Формат определяется в свойстве EditMask. В редакторе свойств для EditMask есть заготовки некоторых форматов: даты, валюты и т.п. Специальные символы для маски можно посмотреть в Справочнике.

TOutline - используется для представления иерархических отношений связанных данных. Например - дерево директорий.

TStringGrid - служит для представления текстовых данных в виде таблицы. Доступ к каждому элементу таблицы происходит через свойство Cell.

TDrawGrid - служит для представления данных любого типа в виде таблицы. Доступ к каждому элементу таблицы происходит через свойство CellRect.

TImage - отображает графическое изображение на форме. Воспринимает форматы BMP, ICO, WMF. Если картинку подключить во время дизайна программы, то она прикомпилируется к EXE файлу.

TShape - служит для отображения простейших графических объектов на форме: окружность, квадрат и т.п.

TBevel - элемент для рельефного оформления интерфейса.

THeader - элемент оформления для создания заголовков с изменяемыми размерами для таблиц.

TScrollBox - позволяет создать на форме прокручиваемую область с размерами большими, нежели экран. На этой области можно разместить свои объекты.

2.4.2. Страница Dialogs

На странице Dialogs представлены компоненты для вызова стандартных диалогов Windows. Внешний вид диалогов зависит от используемой версии Windows. Объекты, представленные на данной странице, невидимы во время выполнения и вызов диалогов происходит программно, например:

```
if OpenDialog1.Execute then
Image1.Picture.LoadFromFile(OpenDialog1.FileName);
```

Диалоги Windows в порядке появления на странице Dialogs:

OpenDialog – выбрать файл

SaveDialog; сохранить файл

FontDialog; настроить шрифт

ColorDialog; выбор цвета
PrintDialog; печать
PrinterSetupDialog; настройка принтера
FindDialog; поиск строки
ReplaceDialog; поиск с заменой

2.4.3. Страница System

Страница представляет набор компонент для доступа к некоторым системным сервисам типа таймер, DDE, OLE и т.п.

TTimer - таймер, событие OnTimer периодически вызывается через промежуток времени, указанный в свойстве Interval. Период времени может составлять от 1 до 65535 мс.

TPaintBox - место для рисования. В обработчики событий, связанных с мышкой, передаются относительные координаты мышки в TPaintBox, а не абсолютные в форме.

TFileListBox - специализированный ListBox, в котором отображаются файлы из указанной директории (свойство Directory). На названия файлов можно наложить маску, для чего служит свойство Mask. Кроме того, в свойстве FileEdit можно указать объект TEdit для редактирования маски.

TDirectoryListBox - специализированный ListBox, в котором отображается структура директорий текущего диска. В свойстве FileList можно указать TFileListBox, который будет автоматически отслеживать переход в другую директорию.

TDriveComboBox - специализированный ComboBox для выбора текущего диска. Имеет свойство DirList, в котором можно указать TDirectoryListBox, который будет отслеживать переход на другой диск.

TFilterComboBox - специализированный ComboBox для выбора маски имени файлов. Список масок определяется в свойстве Filter. В свойстве FileList указывается TFileListBox, на который устанавливается маска.

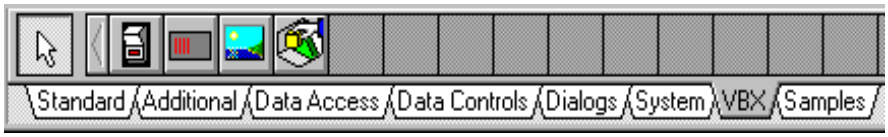
С помощью последних четырех компонент (TFileListBox, TDirectoryListBox, TDriveComboBox, TFilterComboBox) можно построить свой собственный диалог выбора файла, причем для этого не потребуется написать ни одной строчки кода.

TMediaPlayer - служит для управления мультимедийными устройствами (типа CD-ROM, MIDI и т.п.). Выполнен в виде панели управления с кнопками Play, Stop, Record и др. Для воспроизведения может понадобиться как соответствующее оборудование, так и программное обеспечение. Подключение устройств и установка ПО производится в среде Windows. Например, для воспроизведения видео, записанного в формате AVI, в потребуется установить ПО MicroSoft Video (в Windows 3.0, 3.1, WFW 3.11).

TOLEContainer - контейнер, содержащий OLE объекты. Поддерживается OLE 2.02

TDDEClientConv,TDDEClientItem, TDDEServerConv, TDDEServerItem - 4 объекта для организации DDE. С их помощью можно построить приложение как DDE-сервер, так и DDE-клиент. Подробнее - на следующих уроках.

2.4.4. Страница VBX



Поскольку формат объектов из MicroSoft Visual Basic (VBX) является своего рода стандартом и существует большое количество библиотек таких объектов, то в Delphi была предусмотрена совместимость с этим форматом. VBX версии 1.0 можно включить в Палитру Компонент Delphi и использовать их как "родные" компоненты (в том числе, выбирать их в качестве предков и наследовать свойства и методы).

TBiSwitch - двухпозиционный переключатель.

TBiGauge - прогресс-индикатор.

TBiPict - аналог TImage.

TChartFX - деловая графика.

2.5. Рисование и закраска

2.5.1. Графические компоненты

В стандартную библиотеку визуальных компонент Delphi входят несколько объектов, с помощью которых можно придать своей программе совершенно оригинальный вид. Это - TImage (TDBImage), TShape, TBevel.

TImage позволяет поместить графическое изображение в любое место на форме. Данный объект весьма прост в использовании - выберите его на странице Additional и поместите в нужное место формы. Собственно, картинку можно загрузить во время дизайна в редакторе свойства Picture (Инспектор Объектов). Картинка должна храниться в файле в формате BMP (*bitmap*), WMF (*Windows Meta File*) или ICO (*icon*). (TDBImage отображает картинку, хранящуюся в таблице в поле типа BLOB. При этом доступен только формат BMP.)

Как известно, форматов хранения изображений гораздо больше трех вышеназванных (например, наиболее известны PCX, GIF, TIFF, JPEG). Для включения в программу изображений в этих форматах нужно либо перевести их в формат BMP, либо найти библиотеки третьих фирм, в которых есть аналог

TImage, “понимающий” данные форматы (есть как VBX объекты, так и “родные” объекты для Delphi).

При проектировании следует помнить, что изображение, помещенное на форму во время дизайна, включается в файл .DPR и затем прикомпилируется к EXE файлу. Поэтому такой EXE файл может получиться достаточно большой. Как альтернативу можно рассмотреть загрузку картинки во время выполнения программы, для этого у свойства Picture (которое является объектом со своим набором свойств и методов) есть специальный метод LoadFromFile. Это делается, например, так:

```
if OpenDialog1.Execute then
 Image1.Picture.LoadFromFile(OpenDialog1.FileName);
```

Важными являются свойства объекта Center и Stretch - оба имеют булевский тип. Если Center установлено в True, то центр изображения будет совмещаться с центром объекта TImage. Если Stretch установлено в True, то изображение будет сжиматься или растягиваться таким образом, чтобы заполнить весь объект TImage.

TShape - простейшие графические объекты на форме типа круг, квадрат и т.п. Вид объекта указывается в свойстве Shape. Свойство Pen определяет цвет и вид границы объекта. Brush задает цвет и вид заполнения объекта. Эти свойства можно менять как во время дизайна, так и во время выполнения программы.

TBevel - объект для украшения программы, может принимать вид рамки или линии. Объект предоставляет меньше возможностей по сравнению с TPanel, но не занимает ресурсов. Внешний вид указывается с помощью свойств Shape и Style.

2.5.2. Свойство объектов Canvas

У ряда объектов из библиотеки визуальных компонент есть свойство Canvas (канва), которое предоставляет простой путь для рисования на них. Эти объекты - TBitmap, TComboBox, TDBComboBox, TDBGrid, TDBListBox, TDirectoryListBox, TDrawGrid, TFileListBox, TForm, TImage, TListBox, TOutline, TPaintBox, TPrinter, TStringGrid. Canvas в свою очередь является объектом, объединяющим в себе поле для рисования, карандаш (Pen), кисть (Brush) и шрифт (Font). [50,51] Canvas обладает также рядом графических методов : Draw, TextOut, Arc, Rectangle и др. Используя Canvas, Вы можете воспроизводить на форме любые графические объекты - картинки, многоугольники, текст и т.п. без использования компонент TImage,TShape и TLabel (т.е. без использования дополнительных ресурсов), однако при этом Вы должны обрабатывать событие OnPaint того объекта, на канве которого рисуете. Рассмотрим подробнее свойства и методы объекта Canvas.

Свойства Canvas :

Brush -кисть, является объектом со своим набором свойств:

Bitmap - картинка размером строго 8x8, используется для заполнения (заливки) области на экране.

Color - цвет заливки.

Style - предопределенный стиль заливки; это свойство конкурирует со свойством Bitmap - какое свойство Вы определили последним, то и будет определять вид заливки.

Handle - данное свойство дает возможность использовать кисть в прямых вызовах процедур Windows API .

ClipRect - (только чтение) прямоугольник, на котором происходит графический вывод.

CopyMode - свойство определяет, каким образом будет происходить копирование (метод CopyRect) на данную канву изображения из другого места: один к одному, с инверсией изображения и др.

Font - шрифт, которым выводится текст (метод TextOut).

Handle - данное свойство используется для прямых вызовов Windows API.

Pen - карандаш, определяет вид линий; как и кисть (Brush) является объектом с набором свойств:

Color - цвет линии

Handle - для прямых вызовов Windows API

Mode - режим вывода: простая линия, с инвертированием, с выполнением исключающего или и др.

Style - стиль вывода: линия, пунктир и др.

Width - ширина линии в точках

PenPos - текущая позиция карандаша, карандаш рекомендуется перемещать с помощью метода MoveTo, а не прямой установкой данного свойства.

Pixels - двухмерный массив элементов изображения (pixel), с его помощью Вы получаете доступ к каждой отдельной точке изображения

Методы Canvas:

Методы для рисования простейшей графики - *Arc*, *Chord*, *LineTo*, *Pie*, *Polygon*, *PolyLine*, *Rectangle*, *RoundRect*. При прорисовке линий в этих методах используются карандаш (Pen) канвы, а для заполнения внутренних областей - кисть (Brush).

Методы для вывода картинок на канву - *Draw* и *StretchDraw*. В качестве параметров указываются прямоугольник и графический объект для вывода (это может быть TBitmap, TIcon или TMetafile). *StretchDraw* отличается тем, что растягивает или сжимает картинку так, чтобы она заполнила весь указанный прямоугольник (см. пример к данному уроку).

Методы для вывода текста - *TextOut* и *TextRect.* При выводе текста используется шрифт (Font) канвы. При использовании *TextRect* текст выводится только внутри указанного прямоугольника. Длину и высоту текста можно узнать с помощью функций *TextWidth* и *TextHeight.*

Объект TPaintBox

На странице System Палитры Компонент есть объект TPaintBox, который можно использовать для построения приложений типа графического редактора или, например, в качестве места построения графиков (если, конечно, у Вас нет для этого специальных компонент третьих фирм). Никаких ключевых свойств, кроме Canvas, TPaintBox не имеет, собственно, этот объект является просто

канвой для рисования. Важно, что координаты указателя мыши, передаваемые в обработчики соответствующих событий (OnMouseMove и др.), являются относительными, т.е. это смещение мыши относительно левого верхнего угла объекта TPaintBox, а не относительно левого верхнего угла формы.

Примеры

В первом примере (проект SHAPE.DPR, рис.1) показано, как во время выполнения программы можно изменять свойства объекта TShape. Изменение цвета объекта (событие OnChange для ColorGrid1):

```
procedure TForm1.ColorGrid1Change(Sender: TObject);
begin
 Shape1.Brush.Color:=ColorGrid1.ForeGroundColor;
end;
```

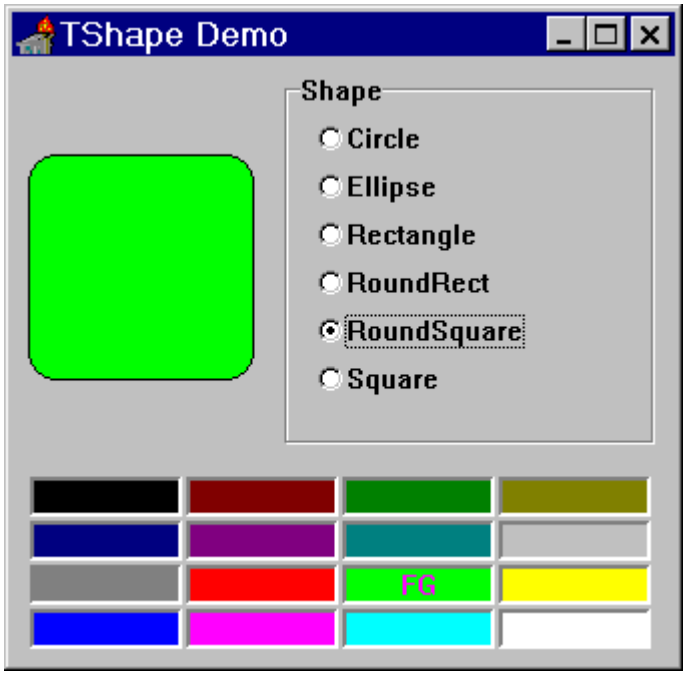


Рис.А. Пример с TShape.

Во втором примере (проект PIXELS.DPR, рис.2) показано, как осуществить доступ к отдельной точке на изображении (на канве). По нажатию кнопки “Fill” всем точкам изображения присваивается свой цвет:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i, j : Longint;
begin
 Button1.Enabled:=False;
 with Canvas do
 for i:=1 to Width do begin
 Application.ProcessMessages;
 for j:=1 to Height do
```

```
 Pixels[i,j]:=i*j;
 end;
 Button1.Enabled:=True;
end;
```

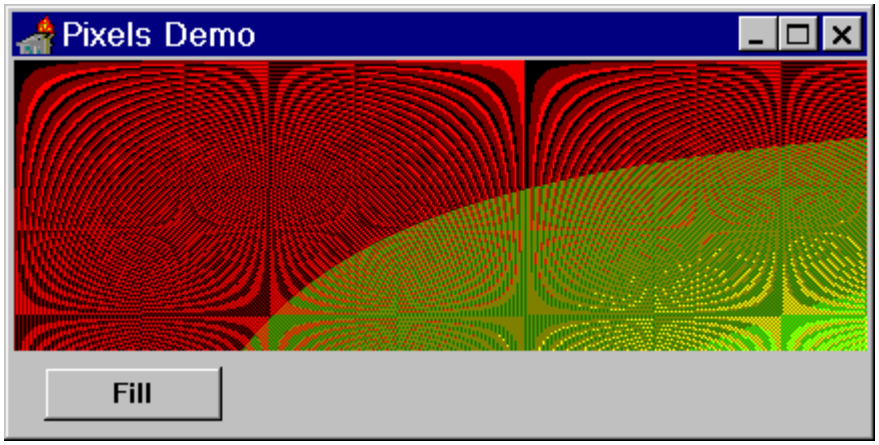


Рис.B. Работа с точками на канве.

В третьей программе (проект DRAW.DPR, рис.3) приведен пример использования методов, выводящих изображение - *Draw* и *StretchDraw:*

Рис.C. Вывод изображений на канву.

Прорисовка изображений происходит в обработчике события OnPaint для формы:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 with Canvas do begin
 Draw(0,0, Image1.Picture.BitMap);
 StretchDraw(Rect(250,0,350,50),Image1.Picture.BitMap)
 end;
```

end;

2.6. Печать текстовая и графическая

2.6.1. Печать в текстовом режиме

Если возникает необходимость напечатать на принтере документ в текстовом режиме, то это делается следующим образом. С принтером возможно работать, как с обычным текстовым файлом, за исключением того, что вместо процедуры AssignFile нужно вызывать процедуру AssignPrn. В примере на принтер выводится одна строка текста:

```
procedure TForm1.Button1Click(Sender: TObject);

var
 To_Prn : TextFile;
begin
 AssignPrn(To_Prn);
 Rewrite(To_Prn);
 Writeln(To_Prn, 'Printer in Text Mode');
 CloseFile(To_Prn);
end;
```

Здесь необходимо пояснить, что по сравнению с BP 7.0, в Delphi изменены названия некоторых функций и переменных в модуле System :

- AssignFile вместо Assign
- CloseFile вместо Close
- TextFile вместо Text

Иногда в программе требуется просто получить твердую копию экранной формы. В Delphi это делается более, чем просто - у объекта TForm есть метод Print [52], который и нужно вызвать в нужный момент.

2.6.2. Графическая печать (объект TPrinter)

И все же более интересно, как из программы, созданной в Delphi, можно вывести на печать графическую информацию. Для этого есть специальный объект Printer (класса TPrinter). Он становится доступен, если к программе подключить модуль Printers (т.е. добавить имя модуля в разделе **uses**). С помощью данного объекта печать на принтере графической информации становится не сложнее вывода этой информации на экран. Основным является то, что Printer предоставляет разработчику свойство Canvas (работа с канвой описана в предыдущей лекции) и методы, выводящие содержание канвы на принтер. Рассмотрим подробнее свойства и методы объекта Printer.

Свойства Printer:

Aborted - тип булевский; показывает, прервал ли пользователь работу принтера методом *Abort.*

Canvas - канва, место для вывода графики; работа с Canvas описана в соответствующей лекции.

Fonts - список доступных шрифтов.

Handle - используется при прямых вызовах Windows API.

Orientation - ориентация страницы, вертикально или горизонтально.

PageWidth, *PageHeight*, *PageNumber* - соответственно ширина, высота и номер страницы.

Printers перечисляет все установленные в системе принтеры, а

PrinterIndex указывает, какой из них является текущим. Чтобы печатать на принтере по умолчанию, здесь должно быть значение -1.

Printing - тип булевский; показывает, начата ли печать (методом *BeginDoc*).

Title - заголовок для Print Manager и для заголовка перед выводом на сетевом принтере.

Методы Printer:

Abort - прерывает печать, начатую методом *BeginDoc*

BeginDoc - вызывается перед тем, как начать рисовать на канве.

EndDoc – вызывается, когда все необходимое уже нарисовано на канве, принтер начинает печатать именно после этого метода.

NewPage - переход на новую страницу.

Остальными методами объекта в обычных случаях пользоваться не нужно.[22,51]

Итак, порядок вывода на печать графической информации выглядит следующим образом:

- выполняется метод *BeginDoc*
- на канве (Canvas) рисуем все, что нужно
- при необходимости разместить информацию на нескольких листах

вызываем метод *NewPage*

- посылаем нарисованное на принтер, выполняя метод *EndDoc*

Пример

На данном примере (проект PRINTS.DPR, рис.1) реализованы все три вышеописанные ситуации.

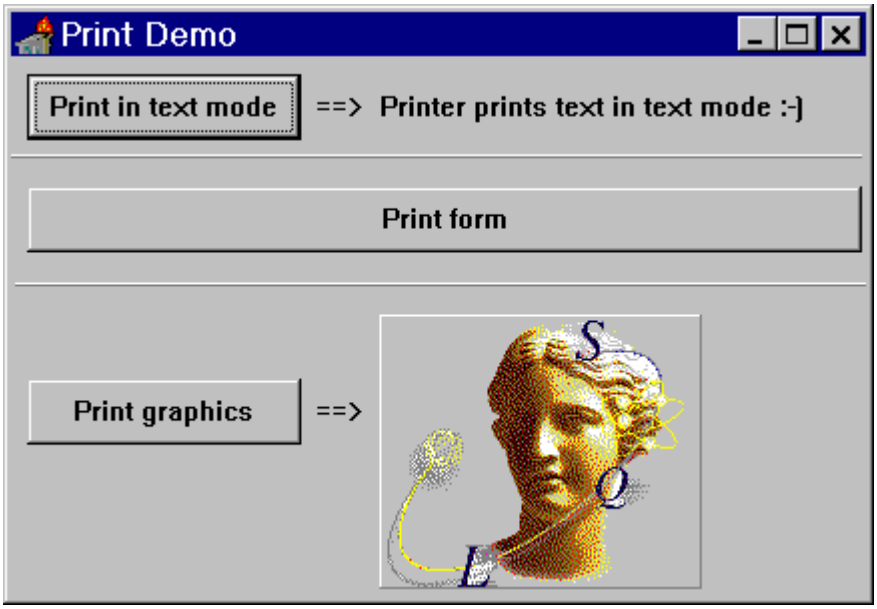


Рис.4. Демонстрационная программа

Краткие выводы

Delphi - это комбинация нескольких важнейших технологий:

- Высокопроизводительный компилятор в машинный код
- Объектно-ориентированная модель компонент
- Визуальное (а, следовательно, и скоростное) построение приложений из программных прототипов
- Масштабируемые средства для построения баз данных

Главные составные части среды программирования

1.Дизайнер Форм (Form Designer)
2.Окно Редактора Исходного Текста (Editor Window)
3.Палитра Компонент (Component Palette)
4.Инспектор Объектов (Object Inspector)
5.Справочник (On-line help)

Инструментальные средства DELPHI:

- Встроенный отладчик
- Внешний отладчик (поставляется отдельно)
- Компилятор командной строки
- WinSight
- WinSpector

Любой **проект** имеет, по-крайней мере, шесть связанных с ним файлов. Три из них относятся к управлению проектом из среды и напрямую программистом не меняются. Вот эти файлы :

- Главный файл проекта, изначально называется PROJECT1.DPR.
- Первый модуль программы /unit/, который автоматически появляется в начале работы. Файл называется UNIT1.PAS по умолчанию, но его можно назвать любым другим именем, вроде MAIN.PAS.

- Файл главной формы, который по умолчанию называется UNIT1.DFM, используется для сохранения информации о внешнем виде главной формы.
- Файл PROJECT1.RES содержит иконку для проекта, создается автоматически.
- Файл, который называется PROJECT1.OPT по умолчанию, является текстовым файлом для сохранения установок, связанных с данным проектом. Например, установленные Вами директивы компилятора сохраняются здесь.
- Файл PROJECT1.DSK содержит информацию о состоянии рабочего пространства.

Ключевые слова

Дизайнер форм, Окно редактора Исходного Текста, Палитра компонент, Инспектор объектов, Справочник, меню, Панель с кнопками для быстрого доступа, Редактор картинок, инструментальные средства: встроенный отладчик, внешний отладчик, компилятор командной строки, WinSight, WinSpector, сохранение программы, тьюторы.

Проект Delphi, управление проектом, пункт меню «File», «Edit», «Search», «View», «Compile».

Страница Standard, страницы Additional, Dialogs, System, VBX

Вопросы для самоконтроля

1. Каковы основные характеристики продукта Delphi?
2. В чем сущность объектно-ориентированной модели программных компонент?
3. Что такое масштабируемые средства для построения баз данных?
4. Для кого предназначен Delphi?
5. Что такое Библиотека Визуальных компонент?
6. Какова общая структура среды Delphi?
7. Что подразумевается под основными элементами среды Delphi?
8. Укажите дополнительные элементы среды Delphi.
9. Каковы Инструментальные средства Delphi?
10. Что подразумевается под стандартными компонентами Delphi?
11. В чем суть сохранения проекта в Delphi?
12. В чем состоит основное назначение страницы Standard?
13. В чем заключается основное назначение страницы Additional?
14. В чем сущность страницы Dialogs?
15. В чем заключается основное назначение страницы System?
16. В чем заключается основное назначение страницы VBX?

Рекомендуемая литература

1. Архангельский А. Программирование в Delphi 7. - М.: ООО «Бином – Пресс», 2004. -1152с.
2. 2.Брауде Э. Технология разработки программного обеспечения. Питер – 2004. - 325 с.
3. 3.Бобров И.С. Delphi 7. Учебный курс. Москва. Санкт-Петербург, Нижний Новгород, Воронеж, Питер.- 2003.- 625
4. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Бином · 1998.
5. Джон Влиссидес, Эрих Гамма, Ричард Хелм, Ральф Джонсон. Приемы объектно-ориентированного проектирования. Паттерны проектирования. - Питер. 2003. - 256 с.
6. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. - М.: Изд. МГТУ имени Н.Э. Баумана. 2003.- 320с.
7. Кьоу Дж., Джеанини М.Объектно-ориентированное программирование. Просто и понятно. − М.: Питер, 2005. -403с.
8. Лесневский А.С. Объектно-ориентированное программирование для начинающих. - М.: Бином. Лаборатория знаний. 2005. - 382 с.
9. Синтес Антони. Освой самостоятельно объектно-ориентированное программирование за 21 день. Просто и понятно− М.: Вильямс. 2002. -284с.
10. http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».
11. http://.borland.com/delphi_net/ - Официальный сайт Borland Delphi
12. http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

Глава 3. СВОЙСТВА, МЕТОДЫ И СОБЫТИЯ В DELPHI

3.1. Управление свойствами визуальных компонент в режиме выполнения

Каждый компонент, помещаемый на форму, имеет свое отражение в окне Инспектора Объектов (Object Inspector). Object Inspector имеет две “странички” - “Properties” (Свойства) и “Events” (События). Создание программы в Delphi сводится к “нанесению” компонентов на форму (которая, кстати, также является компонентом) и настройке взаимодействия между ними путем:

- изменения значения ***свойств*** этих компонентов
- написания адекватных реакций на ***события***.

Свойство является важным атрибутом компонента. Для пользователя (программиста) свойство выглядит как простое поле какой-либо структуры, содержащее некоторое значение. Однако, в отличие от “просто” поля, любое изменение значения некоторого свойства любого компонента сразу же приводит к изменению визуального представления этого компонента, поскольку свойство инкапсулирует в себе методы (действия), связанные с

чтением и записью этого поля (которые, в свою очередь, включают в себя необходимую перерисовку). [22,25,30,53] Свойства служат двум главным целям: во-первых, они определяют внешний вид формы или компонента; во-вторых, свойства определяют поведение формы или компонента.

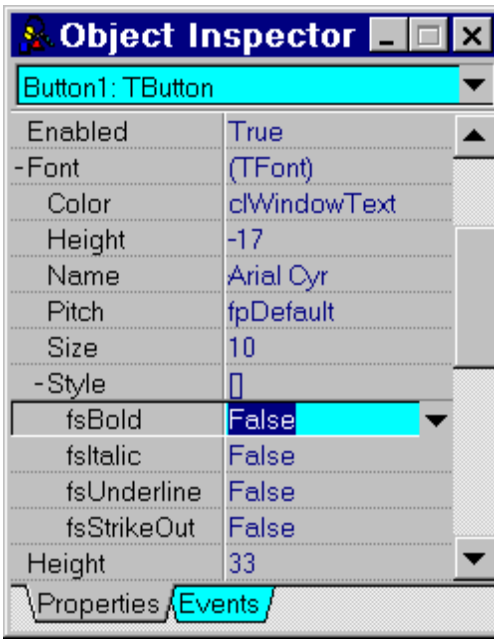


Рис. 1: Отображение комбинированных значений вложенных

Существует несколько типов свойств, в зависимости от их "природы", т.е. внутреннего устройства.

- `Простые` свойства - это те, значения которых являются числами или строками. Например, свойства `Left` и `Top` принимают целые значения, определяющие положение левого верхнего угла компонента или формы. Свойства `Caption` и `Name` представляют собой строки и определяют заголовок и имя компонента или формы.

- `Перечислимые` свойства - это те, которые могут принимать значения из предопределенного набора (списка). Простейший пример - это свойство типа **Boolean**, которое может принимать значения *True* или *False*.

- `Вложенные` свойства - это те, которые поддерживают вложенные значения (или объекты). Object Inspector изображает знак "+" слева от названия таких свойств. Имеются два вида таких свойств: *множества* и *комбинированные значения*. Object Inspector изображает множества в квадратных скобках. Если множество пусто, оно отображается как `[]`. Установки для вложенных свойств вида "множество" обычно имеют значения типа **Boolean**. Наиболее распространенным примером такого свойства является свойство `Style` с вложенным множеством булевых значений. Комбинированные значения отображаются в Инспекторе Объектов как коллекция некоторых величин, каждый со своим типом данных (рис 1). Некоторые свойства, например, `Font`, для изменения своих значений имеют возможность вызвать диалоговое окно. Для этого достаточно щелкнуть маленькую кнопку с тремя точками в правой части строки Инспектора Объектов, показывающей данное свойство.

Delphi позволяет легко манипулировать свойствами компонентов как в режиме проектирования (design time), так и в режиме выполнения программы (run time).

В режиме проектирования манипулирование свойствами осуществляется с помощью Дизайнера Форм (Forms Designer) или на страничке "Properties" Инспектора Объектов. Например, для того чтобы изменить свойства `Height` (высоту) и `Width` (ширину) кнопки, достаточно "зацепить" мышкой за любой ее угол и раздвинуть до нужного

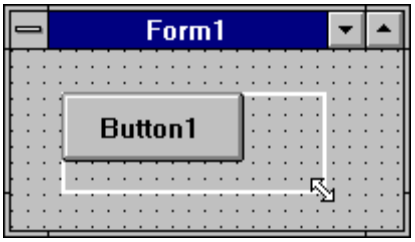


Рис. 2: Изменение размеров с помощью

представления. Того же результата можно добиться, просто подставив новые значения свойств `Height` и `Width` в окне Object Inspector.

С другой стороны, в режиме выполнения пользователь (программист) имеет возможность не только манипулировать всеми свойствами, отображаемыми в Инспекторе Объектов, но и управлять более обширным их списком. В следующем разделе мы рассмотрим, как это делается.

Управление свойствами визуальных компонентов в режиме выполнения осуществляется в следующем порядке.

Все изменения значений свойств компонентов в режиме выполнения должны осуществляться путем прямой записи строк кода на языке Паскаль. В режиме выполнения невозможно использовать Object Inspector.

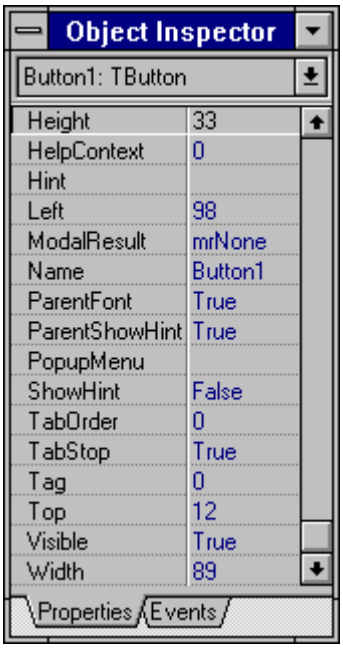


Рис. 3: Изменение размеров с помощью Инспектора Объектов

Однако доступ к свойствам компонентов довольно легко получить программным путем.[30] Все, что Вы должны сделать для изменения какого-либо свойства - это написать простую строчку кода аналогично следующей:

```
MyComponent.Width := 35;
```

Вышеприведенная строка устанавливает ширину (`Width`) компонента в значение 35. Если свойство `Width` компонента еще не было равно 35 к моменту выполнения данной строки программы, Вы увидите, как компонента визуально изменит свою ширину.

Таким образом, нет ничего магического в Инспекторе Объектов. Object Inspector просто является удобным способом выполнения в режиме проектирования того, что может быть осуществлено программным путем в режиме выполнения. Более того, как уже было сказано выше, у компонента могут быть свойства, не отображаемые в окне Инспектора Объектов.

Объектно-ориентированный язык Паскаль, лежащий в основе Delphi, в качестве базового имеет принцип соответствия визуальных компонент тем вещам, которые они представляют. Разработчики Delphi поставили перед собой цель, чтобы, например, представление компонента `Button` (кнопка), инкапсулирующее некий код, соответствовало визуальному изображению кнопки на экране и являлось как можно более близким эквивалентом реальной кнопки, которую Вы можете найти на клавиатуре. И именно из этого принципа родилось понятие свойства.

Если изменить свойства `Width` и `Height` компонента `Button`, кнопка соответствующим образом изменит свои ширину и высоту. Пользователю нет необходимости после изменения свойства `Width` указывать объекту, чтобы он перерисовал себя, хотя при обычном программировании именно так и нужно поступать. Свойства - это более чем просто данные. Напротив, они делают эти данные “живыми”, и все это происходит перед глазами пользователя. Свойства

создают иллюзию, как будто пользователь имеет дело с реальными объектами, а не с их программным представлением.

3.1.1. Программа SHAPEDEM

Программа SHAPEDEM.DPR, изображенная на рис. 4, демонстрирует различные способы, с помощью которых можно изменять пользовательский интерфейс при выполнении программы. Эта программа не производит никаких полезных действий кроме демонстрации того, как легко можно создать “дельфийское” приложение с настраиваемым интерфейсом.

Программа SHAPEDEM содержит всего лишь объект `TShape`, размещенный на форме, вместе с двумя полосами прокрутки и несколькими кнопками. Эта программа интересна тем, что позволяет в режиме выполнения изменять размер, цвет и внешний вид объекта `TShape`, равно как размер и цвет самой формы.

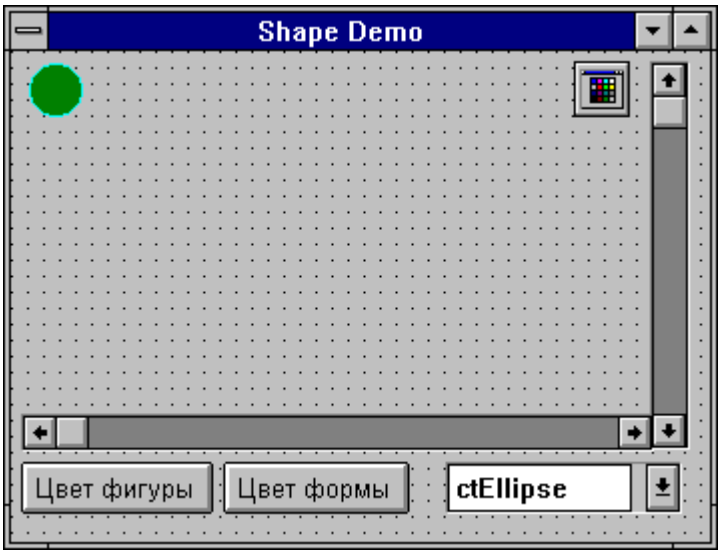


Рис. 4. Программа SHAPEDEM имеет две полосы прокрутки и несколько кнопок

Листинг А показывает код программы SHAPEDEM.

Код головного модуля этой программы мы приведем по частям - по мере его написания.

Листинг А: Исходный код программы SHAPEDEM.DPR.

```
program Shapedem;

uses

 Forms,

 Mina in 'MAIN.PAS' {Form1};

Begin
```

```
  Application.CreateForm(TForm1, Form1);

 Application.Run;

end.
```

На нашем примере полосы прокрутки (ScrollBars) используются для изменения размера фигуры, изображенной в средней части экрана, как показано на рис.5. Для выбора нового вида фигуры используется выпадающий список (ComboBox), а для изменения цвета фигуры или окна (формы) используется стандартное диалоговое окно выбора цвета, вызываемое кнопками “Цвет фигуры” и “Цвет формы”.

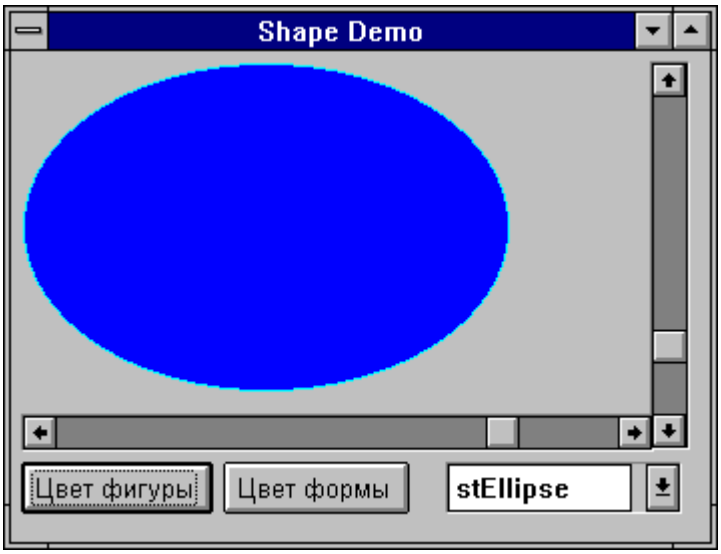


Рис. 5: Вы можете использовать полосы прокрутки, кнопки и список для изменения внешнего вида приложения

Что нужно сделать пользователю (программисту) для того чтобы получить возможность “в режиме выполнения” изменять цвет какого-либо элемента или всего окна (формы)? Для этого достаточно выполнить всего лишь несколько действий. [52]

Для изменения цвета окна просто выберите компонент `ColorDialog` из палитры компонентов (она находится на страничке “Dialogs”) и поместите его на форму. Кроме того, поместите на форму обычную кнопку (компонент `Button`, находится на страничке “Standard”). Для удобства чтения с помощью Object Inspector измените имя компонента (свойство `Name`) с “Button1” (которое дается по умолчанию) на “FormColor”, а его заголовок (свойство `Caption`) - на “Цвет формы”. Дважды щелкните по кнопке “Цвет формы” - Delphi сгенерирует заготовку метода, который выглядит следующим образом:

```
procedure TForm1.FormColorClick(Sender: TObject);
begin
end;
```

Теперь введите две простые строчки кода:

```
procedure TForm1.FormColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Form1.Color := ColorDialog1.Color;
end;
```

Введите эти две строки

Данный код во время выполнения при нажатии кнопки “Цвет формы” вызывает стандартное диалоговое окно выбора цвета, как показано на рис.6.Если в этом диалоговом окне Вы щелкните кнопку OK, выполнится следующая строка:

```
Form1.Color:=ColorDialog1.Color;
```

Этот код установит свойство `Color` формы `Form1` в цвет, который был выбран с помощью диалогового окна `ColorDialog1`. Это весьма просто.

Та же самая техника может использоваться для изменения цвета фигуры (компонент `Shape`, объект TShape). Все, что Вам нужно сделать - это поместить на форму другую кнопку, изменить (при желании) ее имя на “ShapeColor”, а заголовок - на “Цвет Фигуры”, дважды щелкнуть по ней мышкой и создать метод аналогичный следующему:

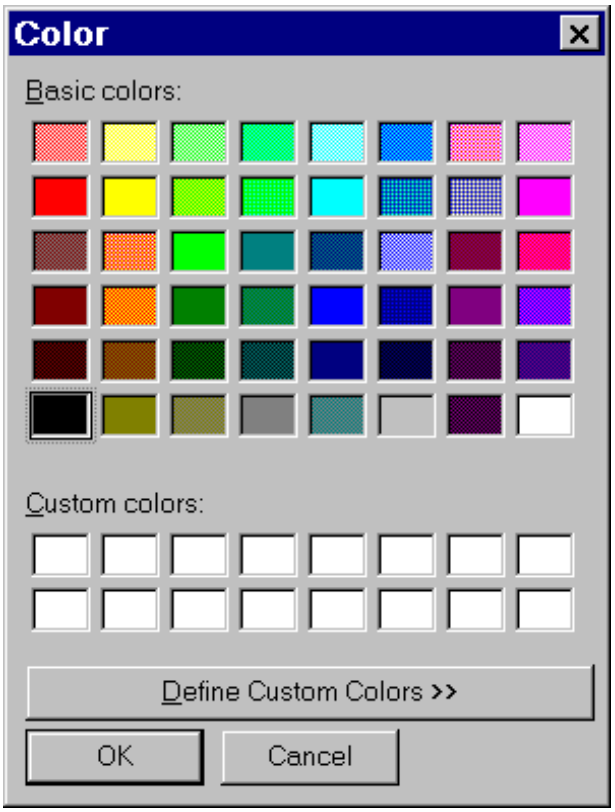


Рис. 6: Диалоговое окно “Color” дает возможность изменить цвет “во время

```
procedure TForm1.ShapeColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Shape1.Brush.Color := ColorDialog1.Color;
end;
```

Введите эти две строки

Отметим, что, написанный здесь код, является самодокументированным, т.е. любой, мало-мальски знакомый с программированием человек, сможет без особого труда разобраться, что же делают эти строки; а если перед этим он прочтет документацию, то для него вообще все будет прозрачно.

Все эти действия можно проделать и автоматически - например, изменить цвет определенного элемента формы, чтобы привлечь внимание пользователя к какому-либо действию.

Весь механизм Windows-сообщений, используемый при взаимодействии компонент во время выполнения, оказывается скрытым от программиста, делая процесс создания программ наиболее легким. Сложное программирование в среде Windows становится доступным “широким” массам программистов. Например, программирование изменения размера фигуры с помощью полос прокрутки, требовавшее в “чистом” Windows сложной обработки сообщений в конструкции типа “case”, в Delphi сводится к написанию одной-единственной строчки кода.

Для начала поместите два компонента `ScrollBar` на форму (находится на страничке “Standard”) и установите свойство `Kind` первого компонента в **sbHorizontal**, а второго - в **sbVertical**. Переключитесь на страничку “Events” в Инспекторе Объектов и создайте заготовки метода для отклика на событие `OnChange` в каждой полосе прокрутки. Напишите в каждом из методов по одной строчке следующим образом:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
 Shape1.Width := ScrollBar1.Position * 3;
end;
```

Введите эту строку

```
procedure TForm1.ScrollBar2Change(Sender: TObject);
begin
 Shape1.Height := ScrollBar2.Position * 2;
end;
```

Введите эту строку

Показанный здесь, код, устанавливает свойства Width и Height фигуры TShape в соответствии с положением “бегунка” на полосах прокрутки (сомножители 3 и 2 введены только для лучшего представления).

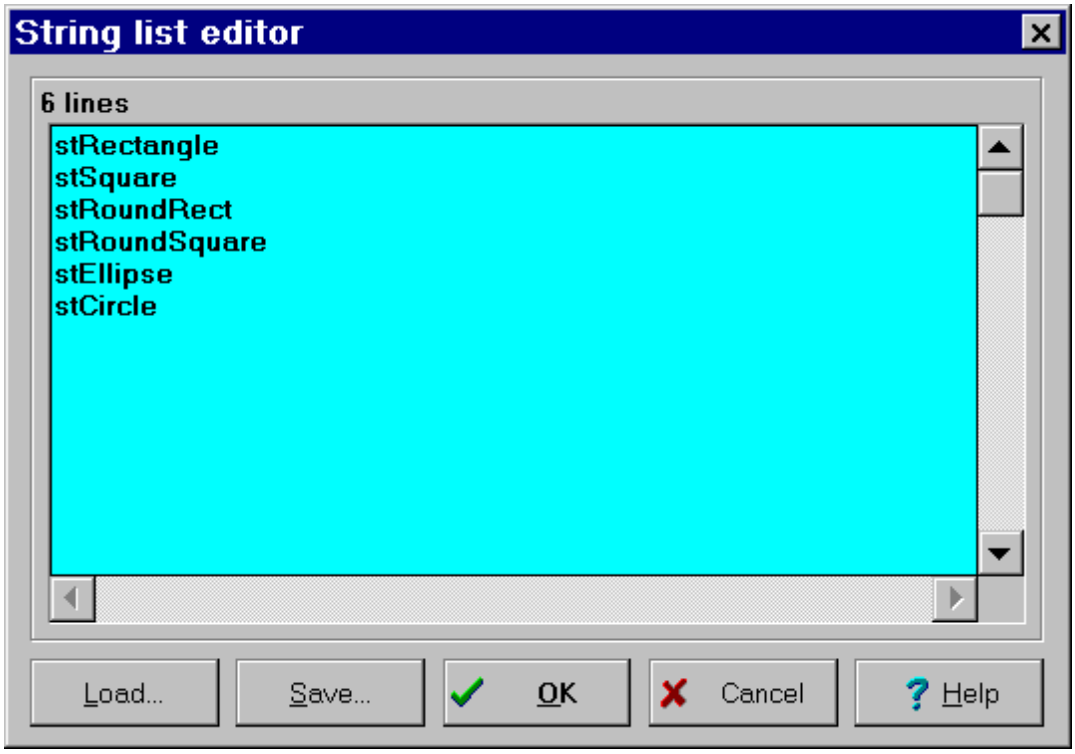
Последняя часть программы SHAPEDEM демонстрирует большие возможности языка Object Pascal, на основе которого построен Delphi. можно ввести элементы в список компонента `ComboBox` как в режиме проектирования, так и при выполнении программы. При этом в режиме проектирования можно просто ввести нужные элементы в список `Items`, щелкнув маленькую кнопку с тремя точками в правой части строки Инспектора Объектов, показывающей данное свойство (`Items`).

Перед Вами появится диалоговое окно текстового редактора (String List Editor), в которое Вы и введете элементы (рис.7). Вы могли заметить, что список этих элементов совпадает со списком опций свойства `Shape` компонента Shape1 (`Shape`). Иными словами, если Вы выделите компонент Shape1 на форме (просто щелкнув по нему) и посмотрите свойство `Shape` в Инспекторе Объектов, Вы увидите список возможных видов фигур, которые может принимать данный компонент. Это как раз те самые виды фигур, которые мы перечисляли в списке у компонента ComboBox1. Данный список Вы можете найти в on-line справочнике по Delphi по контексту “TShapeType”. Или же, если Вы заглянете в исходный код класса TShape, там увидите те же элементы, формирующие перечислимый тип TShapeType:

```
 TShapeType = (stRectangle, stSquare, stRoundRect,
stRoundSquare, stEllipse, stCircle);
```

Итак, смысл всего сказанного заключается в том, что за всеми объектами, которые Вы видите в “дельфийской” программе, стоит некий код на Паскале, к которому

Вы имеете доступ при “прямом” программировании. Ничто от Вас не скрыто. Это значит, что Вы можете изменить поведение любой части Вашей программы во время ее выполнения посредством введения соответствующего



кода.

В нашем конкретном случае, Вам нужно написать только одну строчку кода, которая будет выполнена в качестве отклика на щелчок пользователем по выпадающему списку ComboBox1. Чтобы написать код этого отклика, в режиме проектирования выделите компонент ComboBox1 на форме (как всегда, просто щелкнув по нему левой кнопкой мыши), затем перейдите на страничку “Events” в Инспекторе Объектов. Дважды щелкните по пустому полю напротив события `OnClick`. В редакторе автоматически сгенерируется следующая заготовка метода:

```
procedure TForm1.ComboBox1Click(Sender: TObject);
begin

end;
```

Теперь вставьте одну строчку кода, чтобы метод выглядел следующим образом:

```
procedure TForm1.ComboBox1Click(Sender: TObject);
begin
 Shape1.Shape := TShapeType(ComboBox1.ItemIndex);
end;
```

Введите эту строку

Эта строчка кода устанавливает свойство `Shape` компонента Shape1 в вид, который пользователь выберет в выпадающем списке. Данный код работает благодаря соответствию между порядковыми членами перечислимого типа и числовыми значениями разных элементов в `ComboBox`. Другими словами, первый элемент перечислявмого типа имеет значение 0, что соответствует первому элементу, показанному в `ComboBox` (см. рис.7). Рассмотрим этот подход несколько подробнее.

Если Вы рассмотрите декларацию перечислимого типа TShapeType, то увидите, что первый его элемент называется “stRectangle”. По определению, компилятор назначает этому элементу порядковый номер 0. Следующему по порядку элементу назначается номер 1 и т.д. Таким образом, слова “stRectangle”, “stSquare” и т.п., в действительности, просто символизируют порядковые номера в данном перечислимом типе. На элементы в списке ComboBox также можно сослаться по их порядковому номеру, начиная с 0. Именно поэтому так важно (в данном случае) вводить указанные строки в строгом соответствии с декларацией типа TShapeType. Таким образом, используя преобразование типа “`TShapeType(ComboBox1.ItemIndex)`”, Вы можете указать компилятору, что общего имеют элементы в `ComboBox` и перечислимый тип в TShapeType: а именно, порядковые номера.

Итак, Вы видите, что Delphi является весьма гибким и мощным программным средством, которое позволяет Вам быстро реализовать логику Вашей программы и предоставляет полное управление приложением.

Программа SHAPEDEM2

Программа SHAPEDEM проста в написании и в освоении.[51,53] Однако при изменении пользователем размера окна она будет выглядеть “некрасиво”. Давайте изменим ее таким образом, чтобы программа сама обрабатывала изменение размера окна, а заодно изучим компонент меню. Для достижения этих целей сделаем следующее:

- Кнопки и выпадающий список уберем с экрана и вместо них поместим на форму компонент меню (`MainMenu`)
- “Заставим” полосы прокрутки изменять свое положение в зависимости от размера окна
- “Заставим” свойство `Position` полос прокрутки изменяться так, чтобы правильно отражать размер формы.

Взглянув на рис.8, Вы сможете увидеть, как будет выглядеть программа после этих изменений.

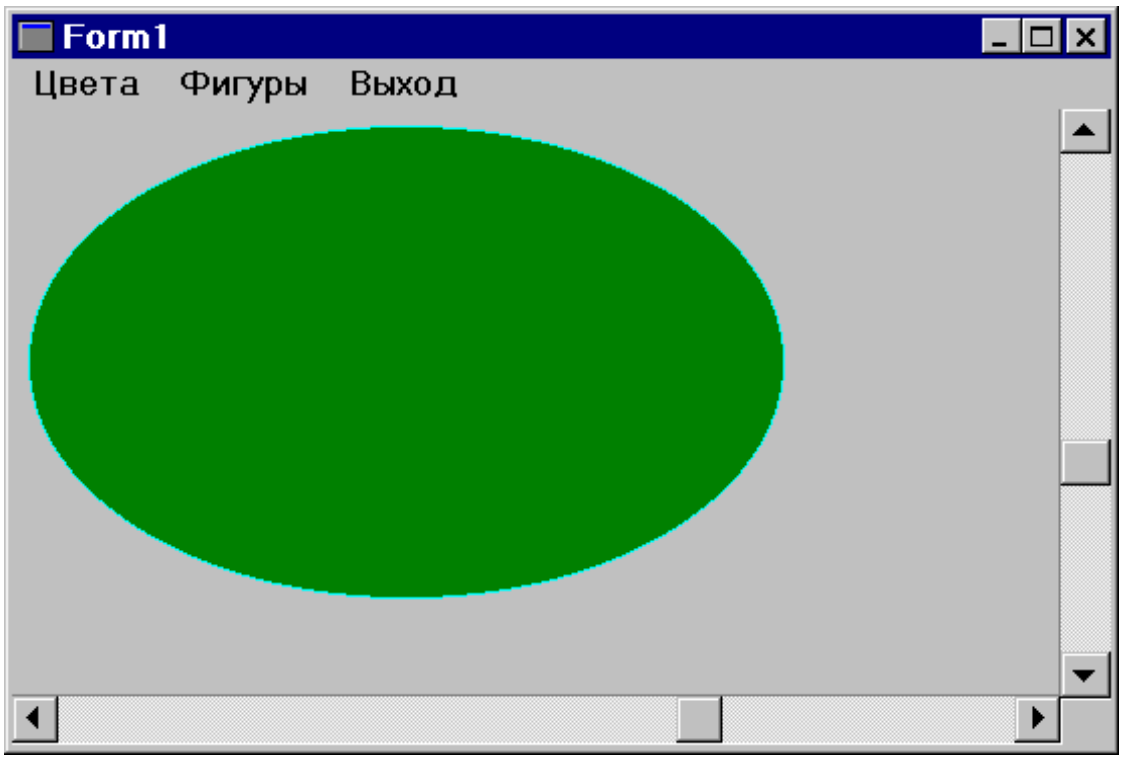


Рис. 8. Программа SHAPDEM2 имеет возможность реагировать на изменение пользователем размера окна

Листинг B: Программа SHAPDEM2 включает метод FormOnResize. Представлен главный модуль.

```
unit Main;
interface
uses
 WinTypes, WinProcs, Classes, Graphics, Forms, Controls,
ColorDlg, StdCtrls, Menus, Dialogs, ExtCtrls;
type
 TForm1 = class(TForm)
 Shape1: TShape;
 ColorDialog1: TColorDialog;
 ScrollBar1: TScrollBar;
 ScrollBar2: TScrollBar;
 MainMenu1: TMainMenu;
 Shapes1: TMenuItem;
 ShapeColor1: TMenuItem;
 FormColor1: TMenuItem;
 Shapes2: TMenuItem;
Rectangle1: TMenuItem;
 Square1: TMenuItem;
 RoundRect1: TMenuItem;
 RoundSquare1: TMenuItem;
 Ellipes1: TMenuItem;
```

```
 Circle1: TMenuItem;
 Exit1: TMenuItem;
 procedure NewShapeClick(Sender: TObject);
 procedure ShapeColorClick(Sender: TObject);
 procedure FormColorClick(Sender: TObject);
 procedure ScrollBar2Change(Sender: TObject);
 procedure ScrollBar1Change(Sender: TObject);
 procedure FormResize(Sender: TObject);
 procedure Exit1Click(Sender: TObject);
 private
 { Private declarations }
 public
 { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.NewShapeClick(Sender: TObject);
begin
 Shape1.Shape := TShapeType((Sender as TMenuItem).Tag);
end;

procedure TForm1.ShapeColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Shape1.Brush.Color := ColorDialog1.Color;
end;

procedure TForm1.FormColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Form1.Color := ColorDialog1.Color;
end;

procedure TForm1.ScrollBar2Change(Sender: TObject);
begin
 Shape1.Height := ScrollBar2.Position;
end;
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
```

```
 Shape1.Width := ScrollBar1.Position;
end;

procedure TForm1.FormResize(Sender: TObject);
var
 Menu,
 Caption,
 Frame: Integer;
begin
 Caption := GetSystemMetrics(sm_cyCaption);
 Frame := GetSystemMetrics(sm_cxFrame) * 2;
 Menu := GetSystemMetrics(sm_cyMenu);
 Scrollbar1.Max := Width;
 Scrollbar2.Max := Height;
 Scrollbar2.Left := Width - Frame - Scrollbar2.Width;
 Scrollbar1.Top := Height - ScrollBar2.Width - Frame -
                                    Caption - Menu;
 Scrollbar1.Width := Width - Scrollbar2.Width - Frame;
 Scrollbar2.Height := Height - Frame - Caption - Menu -
                                    Scrollbar1.Height;
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
 Close;
end;

end.
```

Главное меню для программы создается с помощью компонента MainMenu (находится на страничке “Standard” палитры компонентов). Поместив его на форму, дважды щелкните по нему мышкой - откроется редактор меню, в котором Вы сможете ввести нужные Вам названия пунктов меню и, при желании, изменить их имена (задаваемые Delphi по умолчанию) для удобочитаемости. Создадим меню программы SHAPEDEM2 с тремя главными пунктами: “Цвета”, “Фигуры”, “Выход”.

Для первого пункта создадим следующие подпункты:

- Цвет фигуры
- Цвет окна

Для второго:

- Прямоугольник
- Квадрат
- Закругленный прямоугольник
- Закругленный квадрат

- Эллипс
- Окружность

Третий пункт меню не будет содержать никаких подпунктов.

После создания всех пунктов и подпунктов меню для работы программы SHAPEDEM2 нужно назначить номера для каждого из подпунктов меню, связанных с типом фигуры. Для этого воспользуемся свойством Tag, имеющимся у каждого пункта меню. Свойство Tag (типа Integer) специально введено в каждый компонент Delphi с тем, чтобы программисты могли использовать его по своему усмотрению. Назначим 0 свойству Tag пункта “Прямоугольник”, 1 - пункту “Квадрат”, 2 - пункту “Закругленный прямоугольник” и т.д. Цель такого назначения будет объяснена позднее.

Два метода, созданные для подпунктов изменения цвета, аналогичны тем, которые были в программе SHAPEDEM:

```
procedure TForm1.ShapeColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Shape1.Brush.Color := ColorDialog1.Color;
end;

procedure TForm1.FormColorClick(Sender: TObject);
begin
 if ColorDialog1.Execute then
 Form1.Color := ColorDialog1.Color;
end;
```

Как Вы видите, ничего не изменилось по сравнению с первой версией программы, хотя данные методы уже вызываются из меню, а не из кнопок.

Аналогично методы, реализующие реакцию на выбор подпунктов меню изменения вида фигуры также весьма похожи на методы выбора фигуры через выпадающий список:

```
procedure TForm1.NewShapeClick(Sender: TObject);
begin
 Shape1.Shape := TShapeType((Sender as TMenuItem).Tag);
end;
```

Этот код “работает” правильно благодаря тому, что перечислимый тип TShapeType в качестве начального имеет значение 0 и в свойство Tag подпунктов меню мы также записали порядковые номера, начинающиеся с нуля.

Отметим, что мы использовали оператор as, который позволяет надежно преобразовывать типы из одного в другой: в частности, преобразовать параметр Sender (имеющий общий тип TObject) в тип TMenuItem. Как правило, параметр Sender в Delphi - это управляющий элемент, посылающий сообщения функции,

в которой он фигурирует. В данном случае, Sender является пунктом меню, и, следовательно, Вы можете работать с этим параметром так, как если бы он был декларирован с типом TMenuItem.

Главная причина использования оператора as заключается в том, что он обеспечивает весьма ясный синтаксис, даже если Вы проводите сложное двухуровневое преобразование типов. Более того, оператор as обеспечивает проверку преобразования в режиме выполнения программы. Когда Вы используете оператор as, Вы можете быть уверены в том, что преобразование Sender в TMenuItem реально будет произведено лишь в том случае, если Sender действительно имеет тип TMenuItem.

Две полосы прокрутки в программе SHAPEDEM2 всегда будут располагаться возле границ окна, независимо от его размеров. Выполнение этих действий требует от Вас написания несколько более сложного программирования, чем было ранее. Как было указано выше, Delphi, хотя и скрывает от программиста детали Windows-программирования, однако не запрещает обращаться к функциям Windows API (прикладного пользовательского интерфейса). Таким образом, Delphi поддерживает низкоуровневое программирование на уровне Windows API. Иначе говоря, если Вам нужно углубиться в дебри низкоуровневого программирования - пожалуйста!

```
procedure TForm1.FormResize(Sender: TObject);
var
 Menu,
 Caption,
 Frame: Integer;
begin
 Caption := GetSystemMetrics(sm_cyCaption);
 Frame := GetSystemMetrics(sm_cxFrame) * 2;
 Menu := GetSystemMetrics(sm_cyMenu);
 Scrollbar1.Max := Width;
 Scrollbar2.Max := Height;
 Scrollbar2.Left := Width - Frame - Scrollbar2.Width;
 Scrollbar2.Height := Height - Frame - Caption - Menu;
Scrollbar1.Top :=
 Height - Scrollbar2.Width - Frame - Caption - Menu;
 Scrollbar1.Width := Width - Scrollbar2.Width - Frame;
end;
```

Это вызовы функций Windows API

Код, показанный здесь, является реакцией на событие `OnResize`. Это событие перечислено среди других на страничке “Events” Инспектора Объектов в состоянии, когда выбрана форма (окно). Как Вы можете ожидать, событие (сообщение) `OnResize` посылается форме (окну) каждый раз, когда пользователь “захватывает” мышкой за какой-либо край окна и делает размер

окна большим или меньшим. Однако это же сообщение (событие) посылается окну и тогда, когда происходит максимизация окна (но не минимизация).

Первое, что делается в данном методе, - запрашиваются системные параметры, определяющие размеры заголовка окна, огибающей его рамки и меню. Эта информация “добывается” путем вызова функции GetSystemMetrics, являющейся частью Windows API. Функции GetSystemMetrics передается один аргумент в виде константы, определяющей вид запрашиваемой информации. Например, если Вы передадите функции константу sm_cyCaption, получите в качестве результата высоту заголовка окна (в пикселах). Полный список этих констант имеется в on-line справочнике Delphi (Help|Windows API|Alphabetical functions|User functions|GetSystemMetrics), здесь же мы приведем небольшую выдержку из справочника:

SM_CXBORDER Ширина огибающей окно рамки, размер которой не может быть изменен.
SM_CYBORDER Высота огибающей окно рамки, размер которой не может быть изменен.
SM_CYCAPTION Высота заголовка окна, включая высоту огибающей окно рамки, размер которой не может быть изменен (SM_CYBORDER).
SM_CXCURSOR Ширина курсора.
SM_CYCURSOR Высота курсора.
SM_CXFRAME Ширина огибающей окно рамки, размер которой может быть изменен.
SM_CYFRAME Высота огибающей окно рамки, размер которой может быть изменен.
SM_CXFULLSCREEN Ширина клиентской части для полноэкранного окна.
SM_CYFULLSCREEN Высота клиентской части для полноэкранного окна (эквивалентна высоте экрана за вычетом высоты заголовка окна).
SM_CXICON Ширина иконки.
SM_CYICON Высота иконки.

SM_CYMENU Высота полосы меню в одну строку. Это высота меню за вычетом высоты огибающей окно рамки, размер которой не может быть изменен (SM_CYBORDER).
SM_CXMIN Минимальная ширина окна.
SM_CYMIN Минимальная высота окна.
SM_CXSCREEN Ширина экрана.
SM_CYSCREEN Высота экрана.
SM_MOUSEPRESENT Не 0, если мышь установлена.

В методе FormResize программа вычисляет новые размеры полос прокрутки:

```
Scrollbar1.Max := Width;
Scrollbar2.Max := Height;
```

```
Scrollbar2.Left := Width - Frame - Scrollbar2.Width;
Scrollbar2.Height := Height - Frame - Caption - Menu;
Scrollbar1.Top :=
 Height - Scrollbar2.Width - Frame - Caption - Menu;
Scrollbar1.Width := Width - Scrollbar2.Width - Frame;
```

Вычисления, приведенные здесь, включают простые математические действия. Например, левая сторона вертикальной полосы прокрутки должна быть равна ширине всего окна (формы) за вычетом ширины рамки и ширины самой полосы прокрутки. Это элементарная логика и, реализовав ее в программе, мы получим вертикальную полосу прокрутки, всегда располагающуюся возле правого края окна (формы).

В программе SHAPEDEM свойство `Max` каждой полосы прокрутки оставалось равным значению по умолчанию - 100; это означало, что после того, как бегунок полосы прокрутки пройдет все доступное расстояние (как для вертикальной, так и для горизонтальной полосы прокрутки), свойство `Position` будет установлено в 100. Если бегунок возвращался к началу, свойство `Position` устанавливалось равным свойству `Min`, которое, по умолчанию, 0.

В программе SHAPEDEM2 Вы можете изменять значения свойств `Min` и `Max` так, чтобы диапазон значений `Position` полос прокрутки отражал текущий размер окна (формы) даже при изменении формой своего размера в режиме выполнения. Здесь приведены соответствующие строки из метода FormResize.

```
procedure TForm1.FormResize(Sender: TObject);
begin
 ...
 Scrollbar1.Max := Width;
 Scrollbar2.Max := Height;
 ...
 end;
```

Две показанные выше, строчки кода, просто устанавливают максимальные значения полос прокрутки равными ширине и высоте формы соответственно. После этого Вы всегда сможете сделать помещенную на форму фигуру такой же “большой”, как и сама форма. После введения таких изменений Вам больше не потребуется умножать свойство `Position` на какой-либо множитель.

```
procedure TForm1.Scrollbar2Change (Sender: TObject);
begin
 Shape1.Height := Scrollbar2.Position;
end;
```

Если Вы после этого запустите программу SHAPDEM2 на выполнение, Вы увидите, что она работает корректно при любом изменении размера формы.

Более того, теперь то можете выбирать фигуры и цвета из меню, что придает программе более строгий вид.

В конце хотелось бы сделать одно маленькое замечание: каждая форма, по умолчанию, имеет две полосы прокрутки (`HorzScrollbar` и `VertScrollbar`), которые автоматически появляются всякий раз, когда размер формы становится меньше, чем область, занимаемая управляющими элементами, расположенными на этой форме. Иногда эти полосы прокрутки могут быть весьма полезными, но в нашей ситуации они сделают совсем не то, что хотелось бы. Поэтому, для надежности, можно установить их вложенные свойства `Visible` в False.

Таким образом, мы рассмотрели, как изменять свойства компонентов во время выполнения. В целом, такие действия ненамного сложнее, чем изменение свойств в режиме проектирования с помощью Object Inspector.

3.2. Методы в Delphi

3.2.1. Создание методов с помощью визуальных средств

Чтобы полностью понять и почувствовать все преимущества Delphi, желательно хорошо изучить язык Object Pascal. И хотя возможности визуальной части Delphi чрезвычайно богаты, программистом может стать только тот, кто хорошо разбирается в технике ручного написания кода.

По мере обсуждения темы данной лекции мы рассмотрим несколько простых примеров, которые, тем не менее, демонстрируют технику использования важных управляющих элементов Windows.

Из предыдущей лекции можно сделать вывод о том, что синтаксический “скелет” метода может быть сгенерирован с помощью визуальных средств. Для этого, напомним, нужно в Инспекторе Объектов дважды щелкнуть мышкой на пустой строчке напротив названия интересующего Вас события в требуемом компоненте. Заметим, если эта строчка не пуста, то двойной щелчок на ней просто переместит Вас в окне Редактора Кода в то место, где находится данный метод.

Для более глубокого понимания дальнейшего изложения, кратко остановимся на концепции объектно-ориентированного программирования. Для начала определим базовое понятие объектно-ориентированного программирования - класс. ***Класс*** - это категория объектов, обладающих одинаковыми свойствами и поведением. При этом ***объект*** представляет собой просто экземпляр какого-либо класса. Например, в Delphi тип “форма” (окно) является классом, а переменная этого типа - объектом. Метод - это процедура, которая определена как часть класса и инкапсулирована (содержится) в нем. Методы манипулируют полями и свойствами классов (хотя могут работать и с любыми переменными) и имеют автоматический доступ к *любым* полям и методам своего класса. Доступ к полям и методам других классов зависит от уровня “защищенности” этих полей и методов. Пока же для нас важно то, что

методы можно создавать как визуальными средствами, так и путем написания кода вручную.

Давайте рассмотрим процесс создания программы CONTROL1, которая поможет нам изучить технику написания методов в Delphi.

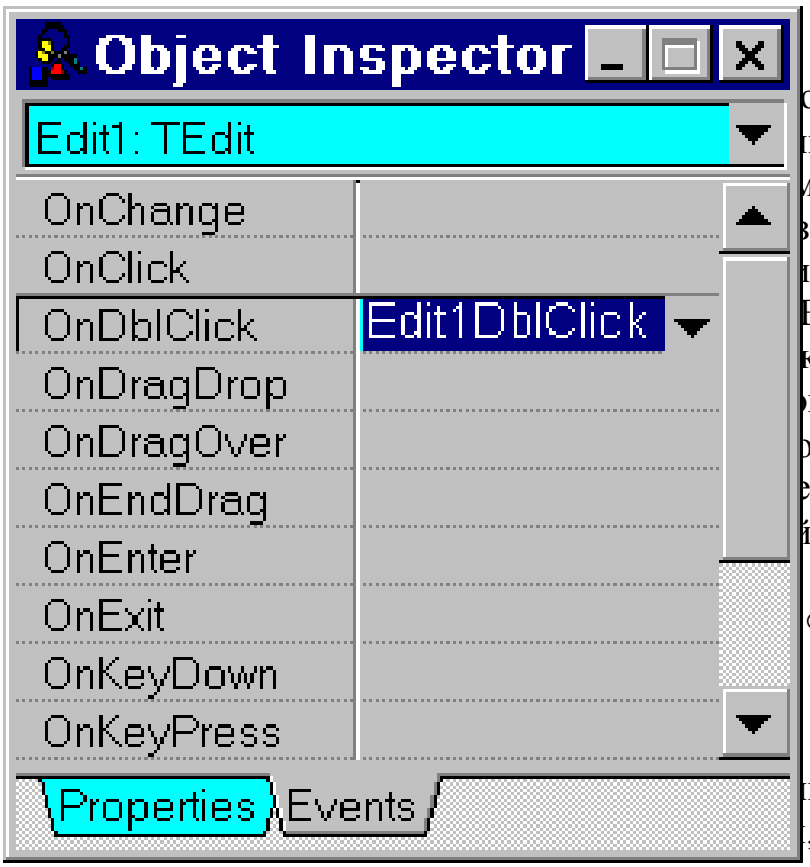


Рис. 8-A. Чтобы создать метод, просто дважды щелкните справа от

программы CONTROL1
о мышки компонент Edit
ичке “Standard” Палитры
му. После этого ваша форма
занный на .
ите в Object Inspector,
Events” и дважды щелкните
ке напротив события
казано на . После этого в
окне Редактора Вы увидите
елет” метода Edit1DblClick,
й на событие OnDblClick:

```
procedure
dit1DblClick(Sender:
```

и процедуры Вы можете
, каким “установил” Delphi,
или изменить его на любое другое (для этого просто введите новое имя в указанной выше строке Инспектора Объектов справа от требуемого события и нажмите **Enter**).

Теперь в окне Редактора Кода введите смысловую часть метода:

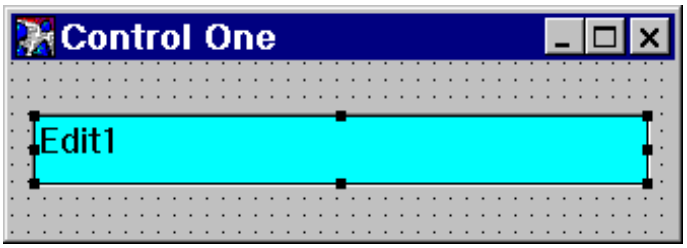


Рис. -B: Главная форма программы CONTROL1

```
procedure TForm1.Edit1DblClick(Sender: TObject);
begin
 Edit1.Text:= 'Вы дважды щелкнули в строке редактирования';
end;
```

Сохраните программу. Во время выполнения дважды щелкните на строке редактирования. Текст в этой строке изменится в соответствии с тем, что мы написали в методе Edit1DblClick: см. Рис. **8-C** .

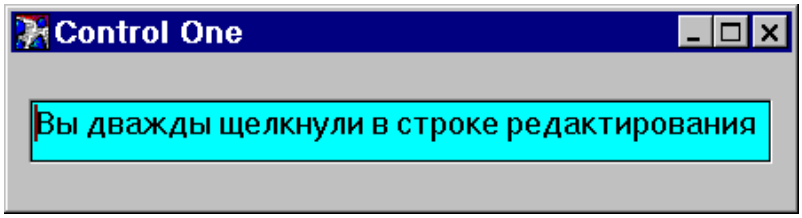


Рис. 8-C: Содержимое управляющего элемента TEdit изменяется после двойного щелчка по нему.

Листинг 8-A и Листинг 8-B предоставляют полный код программы CONTROL1.

Листинг 8-A: Программа CONTROL1 демонстрирует, как создавать и использовать методы в Delphi.

```
program Control1;

uses
 Forms,
 Main in 'MAIN.PAS' {Form1};

begin
 Application.CreateForm(TForm1, Form1);
 Application.Run;
end.
```

Листинг 8-B: Головной модуль программы CONTROL1.

```
unit Main;

interface

uses
 WinTypes, WinProcs,
 Classes, Graphics, Controls,
 Printers, Menus, Forms, StdCtrls;

type
 TForm1 = class(TForm)
 Edit1: TEdit;
 procedure Edit1DblClick(Sender: TObject);
 end;

var
 Form1: TForm1;
```

```
implementation


{$R *.DFM}

procedure TForm1.Edit1DblClick(Sender: TObject);
begin
 Edit1.Text := 'Вы дважды щелкнули в строке редактирования';
end;

end.
```

После того, как Ваша программа загрузится в память, выполняются две строчки кода в CONTROL1.DPR, автоматически сгенерированные компилятором:[30,51,52]

```
Application.CreateForm(TForm1, Form1);
Application.Run;
```

Первая строка запрашивает память у операционной системы и создает там объект Form1, являющийся экземпляром класса TForm1. Вторая строка указывает объекту Application, “по умолчанию” декларированному в Delphi, чтобы он запустил на выполнение главную форму приложения. В данном месте мы не будем подробно останавливаться на классе TApplication и на автоматически создаваемом его экземпляре - Application. Важно понять, что главное его предназначение - быть неким ядром, управляющим выполнением Вашей программы.

Как правило, у большинства примеров, которыми мы будем оперировать в наших уроках, файлы проектов *.DPR* практически одинаковы. Поэтому в дальнейшем там, где они не отличаются кардинально друг от друга, мы не будем приводить их текст. Более того, в файл *.DPR*, автоматически генерируемый Delphi, в большинстве случаев нет необходимости заглядывать, поскольку все, производимые им действия, являются стандартными.

Итак, мы видели, что большинство кода Delphi генерирует автоматически. [22,25,30] В большинстве приложений все, что Вам остается сделать - это вставить одну или несколько строк кода, как в методе Edit1DblClick:

```
Edit1.Text := 'Вы дважды щелкнули в строке редактирования';
```

Хотя внешний интерфейс программы CONTROL1 достаточно прост, она (программа) имеет строгую внутреннюю структуру. Каждая программа в Delphi состоит из файла проекта, с расширением *.DPR* и одного или нескольких модулей, имеющих расширение *.PAS*. Модуль, в котором содержится главная форма проекта, называется головным. Указанием компилятору о связях между модулями служит предложение **Uses**, которое определяет зависимость модулей.

Нет никакого функционального различия между модулями, созданными Вам в Редакторе, и модулями, сгенерированными Delphi автоматически. В любом случае модуль подразделяется на три секции:

- Заголовок

- Секция Interface
- Секция Implementation

Таким образом, “скелет” модуля выглядит следующим образом:

```
unit Main;
                                    {Заголовок модуля}
interface                           {Секция Interface}
implementation                      {Секция Implementation}
end.
```

В интерфейсной секции (**interface**) описывается все то, что должно быть видимо для других модулей (типы, переменные, классы, константы, процедуры, функции). В секции **implementation** помещается код, реализующий классы, процедуры или функции.

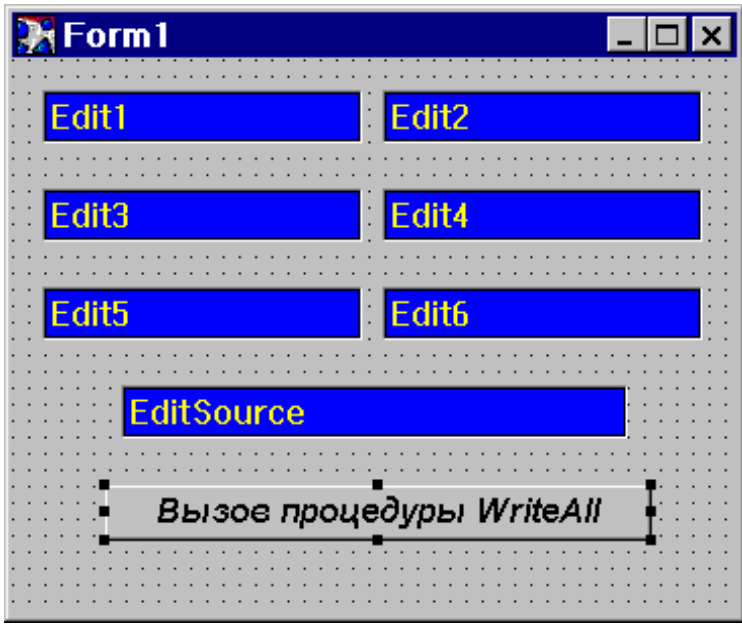


Рис. -D. Программа PARAMS позволяет вызовом одной процедуры заполнить шесть строк

3.2.2. Передача параметров

В Delphi процедурам и функциям (а, следовательно, и методам классов) могут передаваться параметры для того, чтобы обеспечить их необходимой для работы информацией. Программа PARAMS демонстрирует, как использовать передачу параметров в методы Delphi. Кроме того, мы узнаем, как:

- создавать свои собственные процедуры;
- добавлять процедуру в класс, формируя метод класса;
- вызывать одну процедуру из другой.

Программа PARAMS позволяет Вам вводить фразы в строки редактирования. После нажатия кнопки *“Вызов процедуры WriteAll”* строка из управляющего элемента EditSource скопируется в шесть управляющих элементов - строк редактирования, как показано на .

Далее мы не будем подробно останавливаться на том, как размещать компоненты на форме - считаем, что это Вы уже умеете. После того, как Вы разместили на форме семь компонентов `Edit`, переименуйте с помощью Инспектора Объектов седьмой компонент (Edit7) в EditSource. Положите на форму компонент `Button`, и в Object Inspector измените его заголовок (свойство `Caption`) на *“Вызов процедуры WriteAll”* (естественно, Вы можете заменить его шрифт, цвет и т.д.).

После завершения проектирования формы класс TForm1 будет выглядеть следующим образом:

```
TForm1 = class(TForm)
Edit1: TEdit;
Edit2: TEdit;
Edit3: TEdit;
Edit4: TEdit;
Edit5: TEdit;
Edit6: TEdit;
EditSource: TEdit;
Button1: TButton;
end;
```

Следующий шаг заключается в добавлении метода, вызываемого по нажатию пользователем кнопки Button1. Это, напомним, можно сделать двумя способами:

- перейти в Инспекторе Объектов на страничку “Events” (предварительно выбрав компонент Button1 на форме), выбрать слово `OnClick` и дважды щелкнуть мышкой на пустой строчке справа от него
- просто дважды щелкнуть на компоненте Button1 на форме.

Delphi сгенерирует следующую “заготовку”:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Цель программы PARAMS - научиться писать процедуры и передавать в них параметры. В частности, программа PARAMS реагирует на нажатие кнопки Button1 путем вызова процедуры WriteAll и передачи ей в качестве параметра содержимого строки редактирования EditSource (EditSource.Text).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 WriteAll(EditSource.Text);
end;
```

Важно понять, что объект EditSource является экземпляром класса TEdit и, следовательно, имеет свойство Text, содержащее набранный в строке редактирования текст. Как Вы уже, наверное, успели заметить, по умолчанию свойство `Text` содержит значение, совпадающее со значением имени компонента (`Name`) - в данном случае “EditSource”. Свойство `Text` Вы, естественно, можете редактировать как в режиме проектирования, так и во время выполнения.

Текст, который должен быть отображен в шести строках редактирования, передается процедуре WriteAll как параметр. Чтобы передать параметр процедуре, просто напишите имя этой процедуры и заключите передаваемый параметр (параметры) в скобки - вот так:

```
WriteAll(EditSource.Text);
```

Заголовок этой процедуры выглядит следующим образом:

```
procedure TForm1.WriteAll(NewString: String);
```

где указано, что передаваемый процедуре параметр NewString должен иметь тип String.

Вспомним, что задача процедуры WriteAll состоит в копировании содержимого строки редактирования `EditSource` в шесть других строк редактирования `Edit1-Edit6`. Поэтому процедура должна выглядеть следующим образом:

```
procedure TForm1.WriteAll(NewString: String);
begin
 Edit1.Text := NewString;
 Edit2.Text := NewString;
 Edit3.Text := NewString;
 Edit4.Text := NewString;
 Edit5.Text := NewString;
 Edit6.Text := NewString;
end;
```

Поскольку процедура WriteAll не является откликом на какое-либо событие в Delphi, то ее нужно полностью написать “вручную”. Простейший способ сделать это - скопировать заголовок какой-либо уже имеющейся процедуры, исправить его, а затем дописать необходимый код.

Еще возвратимся раз к заголовку процедуры, который состоит из пяти частей:

```
procedure TForm1.WriteAll(NewString: String);
```

- **Первая** часть - зарезервированное слово “**`procedure`**”; **пятая** часть - концевая точка с запятой “`;`”. Обе эти части служат определенным синтаксическим целям, а именно: первая информирует компилятор о том, что определен синтаксический блок “процедура”, а вторая указывает на окончание заголовка (собственно говоря, все операторы в Delphi должны заканчиваться точкой с запятой).

- **Вторая** часть заголовка - слово “TForm1”, которое квалифицирует то обстоятельство, что данная процедура является методом класса TForm1.

- **Третья** часть заголовка - имя процедуры. Вы можете выбрать его любым, по вашему усмотрению. В данном случае мы назвали процедуру “WriteAll”.

- **Четвертая** часть заголовка - параметр. Параметр декларируется внутри скобок и, в свою очередь, состоит из двух частей. Первая часть - имя параметра, вторая - его тип. Эти части разделены двоеточием. Если Вы описываете в процедуре более чем один параметр, то их следует разделить их точкой с запятой, например:

```
procedure Example(Param1: String; Param2: String);
```

После того, как Вы “вручную” создали заголовок процедуры, являющейся методом класса, Вы должны включить его в декларацию класса, например, путем копирования (еще раз напомним, что для методов, являющихся откликами на дельфийские события, данное включение производится автоматически):

```
TForm1 = class(TForm)
 Edit1: TEdit;
 Edit2: TEdit;
 Edit3: TEdit;
 Edit4: TEdit;
 Edit5: TEdit;
 Edit6: TEdit;
 EditSource: TEdit;
 Button1: TButton;
 procedure Button1Click(Sender: TObject);
 procedure WriteAll(NewString: String);
 end;
```

В данном месте нет необходимости оставлять в заголовке метода слово “TForm1”, так как оно уже присутствует в описании класса.

Листинг 8-C показывает полный текст головного модуля программы PARAMS. Мы не включили сюда файл проекта, поскольку, как уже упоминалось, он практически одинаков для всех программ.

Листинг 8-C: Исходный код головного модуля программы PARAMS показывает, как использовать строки редактирования и как передавать параметры.

```
Unit Main;

interface
uses
 WinTypes, WinProcs, Classes,
```

```
 Graphics, Controls,
 Printers, Forms, StdCtrls;
type
 TForm1 = class(TForm)
 Edit1: TEdit;
 Edit2: TEdit;
 Edit3: TEdit;
 Edit4: TEdit;
 Edit5: TEdit;
 Edit6: TEdit;
 EditSource: TEdit;
 Button1: TButton;
 procedure Button1Click(Sender: TObject);
 procedure WriteAll(NewString: String);
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WriteAll(NewString: String);
begin
 Edit1.Text := NewString;
 Edit2.Text := NewString;
 Edit3.Text := NewString;
 Edit4.Text := NewString;
 Edit5.Text := NewString;
 Edit6.Text := NewString;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
 WriteAll(EditSource.Text);
end;

end.
```

При экспериментах с программой PARAMS Вы можете попробовать изменить имена процедур и параметров. Однако следует помнить, что ряд слов в Delphi является ***зарезервированными***, и употреблять их в идентификаторах (именах процедур, функций, переменных, типов, констант) не разрешается - компилятор сразу же обнаружит ошибку. К ним относятся такие слова, как “procedure”, “string”, “begin”, “end” и т.п.; полный же их список приведен в on-line справочнике Delphi.

Запомнить сразу все зарезервированные слова необязательно - компилятор “напомнит” Вам о неправильном их использовании выдачей сообщения типа “Identifier expected.” (Ожидался идентификатор, а обнаружено зарезервированное слово).

Более сложные методы и управляющие элементы

Теперь, когда изучены базовые понятия в системе программирования Delphi, можно продолжить изучение компонент и способов создания их методов.

В программе CONTROL1, рассмотренной в начале лекции, был сгенерирован метод, являющийся откликом на событие `OnClick` строки редактирования Edit1. Аналогично, можно сгенерировать метод, являющийся реакцией на событие `OnDbl Click`.

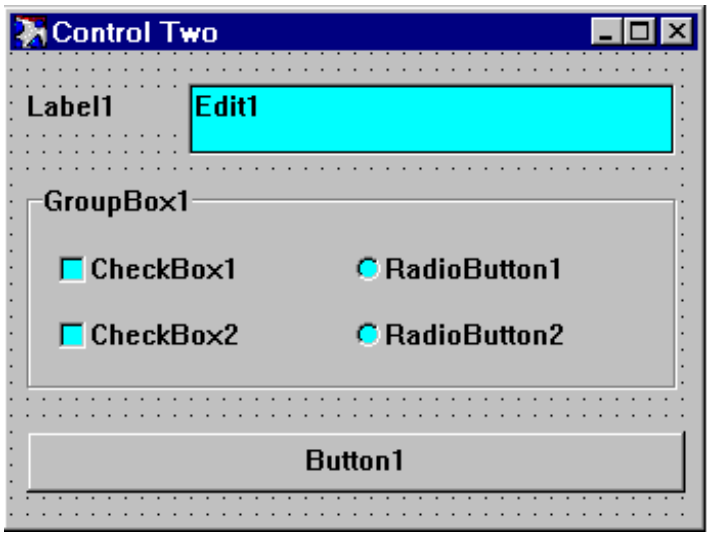


Рис. 8-E: Внешний вид программы

В программе CONTROL2, имеющейся на диске, расширен список находящихся на форме компонентов и для многих из них определены события `OnClick` и `OnDblClick`. Для исследования можно просто скопировать файлы проекта CONTROL1 в новую директорию CONTROL2, изменить имя проекта на CONTROL2.DPR (в этом файле после ключевого слова “**`program`**” также должно стоять название “CONTROL2”) и добавить компоненты `Label`, `GroupBox`, `CheckBox`, `RadioButton`, `Button` на форму (эти компоненты находятся на страничке “Standard” Палитры Компонентов). Форма будет иметь примерно следующий вид - .

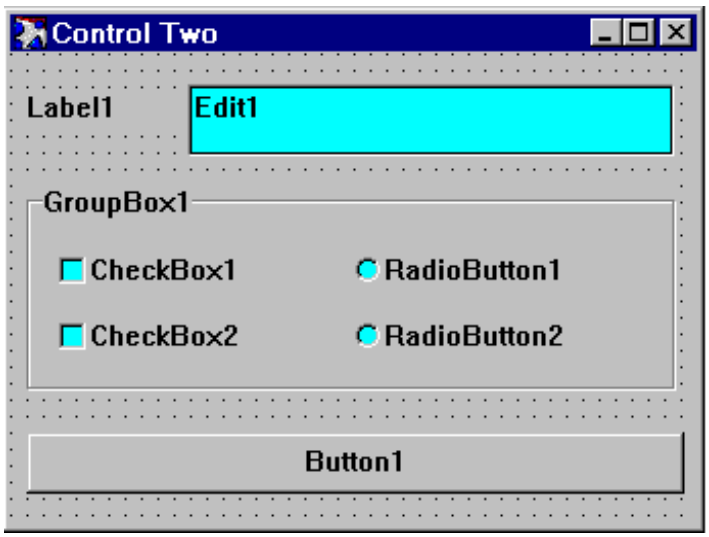


Рис. 8-F. Внешний вид программы CONTROL2

Заметим, что необходимо “положить” компонент `GroupBox` на форму до того, как Вы добавите компоненты `CheckBox` и `RadioButton`, которые, в нашем примере, должны быть “внутри” группового элемента. Иначе, объекты CheckBox1, CheckBox2, RadioButton1 и RadioButton2 будут “думать”, что их родителем является форма Form1 и при перемещении GroupBox1 по форме не будут перемещаться вместе с ней. Таким образом, во избежание проблем, компонент, который должен быть “родителем” других компонентов (Panel, GroupBox, Notebook, StringGrid, ScrollBox и т.д.), нужно помещать на форму до помещения на нее его “детей”. Если Вы все же забыли об этом и поместили “родителя” (например, GroupBox) на форму после размещения на ней его “потомков” (например, CheckBox и RadioButton) - не отчаивайтесь! Отметьте все необходимые объекты и скопируйте (с удалением) их в буфер обмена с помощью команд меню Edit|Cut. После этого отметьте на форме нужный Вам объект (GroupBox1) и выполните команду меню Edit|Paste. После этого все выделенные Вами ранее объекты будут помещены на форму, и их “родителем” будет GroupBox1. Описанный механизм является стандартным и может быть использован для всех видимых компонент.

Выберите объект Label1. Создайте для него метод, являющийся откликом на событие OnDblClick (см. с. 109). Введите в метод одну строчку, например:

```
procedure TForm1.Label1DblClick(Sender: TObject);
begin
 Edit1.Text := 'Двойной щелчок на Label1';
end;
```

Запустите программу на выполнение и дважды щелкните мышкой на метке Label1. Вы увидите, что строка редактирования изменится и в ней появится текст “Двойной щелчок на Label1”.

Теперь закройте приложение и возвратитесь в режим проектирования. Добавьте обработчики событий `OnClick` и `OnDblClick` для каждого объекта,

имеющегося на форме. Текст вашего головного модуля будет выглядеть следующим образом:

Листинг 8-D: Головной модуль программы CONTROL2.

```
Unit Main;

interface

uses
 WinTypes, WinProcs, Classes,
 Graphics, Controls, StdCtrls,
 Printers, Menus, Forms;

type
 TForm1 = class(TForm)
 Label1: TLabel;
 Edit1: TEdit;
 Button1: TButton;
 GroupBox1: TGroupBox;
 CheckBox1: TCheckBox;
 CheckBox2: TCheckBox;
 RadioButton1: TRadioButton;
 RadioButton2: TRadioButton;
 procedure Edit1DblClick(Sender: TObject);
 procedure Label1DblClick(Sender: TObject);
 procedure CheckBox1Click(Sender: TObject);
 procedure CheckBox2Click(Sender: TObject);
 procedure RadioButton1Click(Sender: TObject);
 procedure RadioButton2Click(Sender: TObject);
 procedure Button1Click(Sender: TObject);
 end;

var
 Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Edit1DblClick(Sender: TObject);
begin
 Edit1.Text := 'Двойной щелчок на Edit1';
end;

procedure TForm1.Label1DblClick(Sender: TObject);
begin
 Edit1.Text := 'Двойной щелчок на Label1';
```

```
end ;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на CheckBox1';
end;

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на CheckBox2';
end;

procedure TForm1.RadioButton1Click(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на RadioButton1';
end;

procedure TForm1.RadioButton2Click(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на Radiobutton2';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на Button1';
end;

end.
```

Эта программа служит двум целям:

- она показывает, как создавать процедуры (методы) и как “наполнять” их содержательной “начинкой”;
- она демонстрирует технику работы с управляющими элементами Windows.

3.2.3. Информация периода выполнения. Программа CONTROL3

Как Вы, наверное, заметили, методы программы CONTROL2, являющиеся откликами на события `OnClick` и `OnDblClick`, во многом похожи друг на друга.

Открытость среды Delphi позволяет получать и оперировать информацией особого рода, называемой информацией периода выполнения (RTTI - run-time `type` information), которая организована в виде нескольких уровней.

Верхний уровень RTTI представлен как средство проверки и приведения типов с использованием ключевых слов is и as.

Ключевое слово is дает программисту возможность определить, имеет ли данный объект требуемый тип или является одним из наследников данного типа, например, таким образом:

if MyObject **is** TSomeObj **then** ...

Имеется возможность использовать RTTI и для процесса приведения объектного типа, используя ключевое слово as:

if MyObject **is** TSomeObj **then**
(MyObject **as** TSomeObj).MyField:=...

что эквивалентно:

TSomeObj(MyObject).MyField:=...

Средний уровень RTTI использует методы объектов и классов для подмены операций as и is на этапе компиляции. В основном, все эти методы заложены в базовом классе TObject, от которого наследуются все классы библиотеки компонент VCL. Для любого потомка TObject доступны, в числе прочих, следующие информационные методы:

- `ClassName` - возвращает имя класса, экземпляром которого является объект;
- `ClassInfo` - возвращает указатель на таблицу с RTTI, содержащей информацию о типе объекта, типе его родителя, а также о всех его публикуемых свойствах, методах и событиях;
- `ClassParent` - возвращает тип родителя объекта;
- `ClassType` - возвращает тип самого объекта;
- `InheritsFrom` - возвращает логическое значение, определяющее, является ли объект потомком указанного класса;
- `InstanceSize` - возвращает размер объекта в байтах.

Эти методы могут использоваться в Вашем коде напрямую.

Нижний уровень RTTI определяется в дельфийском модуле *TypInfo* и представляет особый интерес для разработчиков компонентов. Через него можно получить доступ к внутренним структурам Delphi, в том числе, к ресурсам форм, инспектору объектов и т.п.

Итак, доступ к информации периода выполнения в Delphi позволяет динамически получать как имя объекта, находящегося на форме, так и название класса, которому он принадлежит (и еще много другой полезной информации; но об этом - в дальнейших уроках). Для этого используется свойство Name, имеющееся у любого класса-наследника TComponent (а таковыми являются все компоненты, входящие в дельфийскую библиотеку VCL), и метод ClassName, доступный для любого потомка класса базового TObject. А, поскольку класс TComponent, в свою очередь, является наследником класса TObject, то он доступен для всех компонент из библиотеки VCL.

Вернувшись к нашим примерам, мы можем заменить целую “кучу” методов двумя, реализующими события OnClick и OnDblClick для всех объектов сразу. Для этого можно скопировать все файлы из CONTROL2 в новый директорий CONTROL3 или использовать для работы уже имеющуюся на диске программу. Создадим стандартным образом методы ControlDblClick и ControlClick для какого-либо объекта (например, для Label1). Введем в них следующие строки:

```
procedure TForm1.ControlDblClick(Sender: TObject);
begin
 Edit1.Text := 'Двойной щелчок на ' +
 (Sender as TComponent).Name +
 ' (класс ' + Sender.ClassName + ')';
end;

procedure TForm1.ControlClick(Sender: TObject);
begin
 Edit1.Text := 'Щелчок на ' +
 (Sender as TComponent).Name +
 ' (класс ' + Sender.ClassName + ')';
end;
```

Введите эти три строки

Введите эти три строки

Теперь назначим данные методы всем событиям `OnClick` и `OnDblClick`, имеющимся у расположенных на форме объектов. Мы видим, что размер программы существенно сократился, а ее функциональность значительно возросла. В режиме выполнения после, например, щелчка на объекте CheckBox1 приложение будет иметь вид, изображенный на .

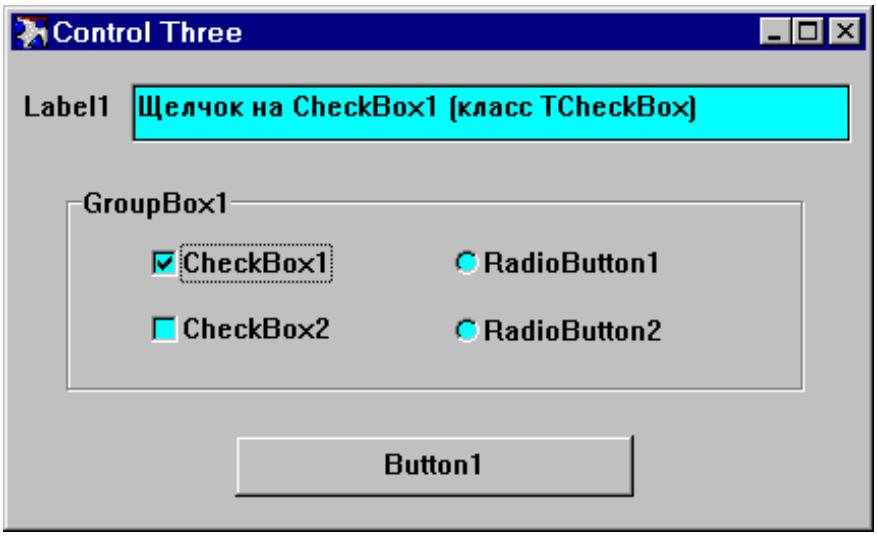


Рис. 8-G. Программа CONTROL3 выводит информацию не только об имени объекта, но и о названии его класса (типа)

Итак, мы видим, что используя информацию периода выполнения, можно сделать программу весьма гибкой и универсальной.

В этой лекции мы рассмотрели, как управлять методами компонентов во время выполнения программы. Помимо этого, мы изучили, что такое информация периода выполнения и научились использовать ее в целях создания гибких и универсальных приложений.

3.3. События в Delphi

Одна из ключевых целей среды визуального программирования - скрыть от пользователя сложность программирования в Windows. При этом, однако, хочется, чтобы такая среда не была упрощена слишком, не до такой степени, что программисты потеряют доступ к самой операционной системе.

Программирование, ориентированное на события, - неотъемлемая черта Windows. Некоторые программные среды для быстрой разработки приложений (RAD) пытаются скрыть от пользователя эту черту совсем, как будто она столь сложна, что большинство не могут ее понять. Истина заключается в том, что событийное программирование само по себе не так уж сложно. Однако есть некоторые особенности воплощения данной концепции в Windows, которые в некоторых ситуациях могут вызвать затруднения.

Delphi, с одной стороны предоставляет полный доступ к подструктуре событий, предоставляемой Windows, с другой - Delphi упрощает программирование обработчиков таких событий.

В данной лекции приводится несколько примеров того, как обрабатывать события в Delphi, дается более детальное объяснение работы системы, ориентированной на события.

Объекты из библиотеки визуальных компонентов (VCL) Delphi, равно как и объекты реального мира, имеют свой набор свойств и свое поведение - набор откликов на события, происходящие с ними. Список событий для данного объекта, на которые он реагирует, можно посмотреть, например, в Инспекторе Объектов на странице событий. (На самом деле, на этой странице представлен список свойств, которые имеют тип вроде TMouseMoveEvent и представляют собой процедуры-обработчики событий. Существует соглашение по названиям данных свойств. Например, OnDblClick соответствует двойному щелчку мыши, а OnKeyUp - событию, когда нажатая клавиша была отпущена.) Среди набора событий для различных объектов из VCL есть как события, портируемые из Windows (MouseMove, KeyDown), так и события, порождаемые непосредственно в программе (DataChange для TDataSource).

Поведение объекта определяется тем, какие обработчики и для каких событий он имеет. Создание приложения в Delphi состоит из настройки свойств используемых объектов и создания обработчиков событий.

Простейшие события, на которые порой нужно реагировать, - это, например, события, связанные с мышкой (они есть практически у всех видимых объектов) или событие Click для кнопки TButton. Предположим, что вы хотите перехватить щелчок левой кнопки мыши на форме. Чтобы сделать это - создайте новый проект, в Инспекторе Объектов выберите страницу событий и сделайте двойной щелчок на правой части для свойства OnClick. Вы получите заготовку для обработчика данного события:

```
procedure TForm1.FormClick(Sender: TObject);
begin
```

```
end;
Напишите здесь следующее:
procedure TForm1.FormClick(Sender: TObject);
begin
 MessageDlg('Hello', mtInformation, [mbOk], 0);
end;
```

Каждый раз, когда делается щелчок левой кнопки мыши над формой будет появляться окно диалога (см. рис.1).

Рис.1. Диалог, появляющийся при щелчке мыши на форме.

Код, приведенный выше, представляет собой простейший случай ответа на событие в программе на Delphi. Он настолько прост, что многие программисты могут написать такой код и без понимания того, что они на самом деле отвечают на сообщение о событии, посланное им операционной системой. Хотя программист получает это событие через третьи руки, тем не менее он на него отвечает.

Опытные программисты в Windows знают, что при возникновении события, операционная система передает вам не только уведомление о нем, но и некоторую связанную с ним информацию. Например, при возникновении события “нажата левая кнопка мыши” программа информируется о том, в каком месте это произошло. Если вы хотите получить доступ к такой информации, то должны вернуться в Инспектор Объектов и создать обработчик события OnMouseDown:

```
procedure TForm1.FormMouseDown(Sender: Tobject;
 Button: TmouseButton;
 Shift: TshiftState;
X, Y: Integer);
begin
 Canvas.TextOut(X, Y, ‘X=’+IntToStr(X)+’ Y=’+IntToStr(Y));
end;
```

Запустите программу, пощелкайте мышкой на форме:

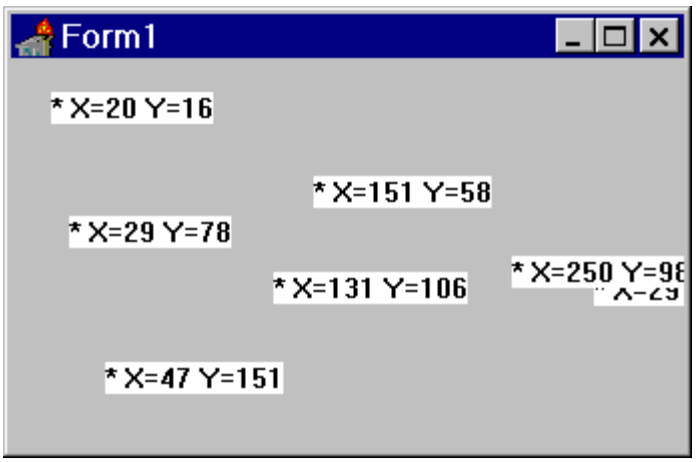


Рис.2.

Как видите, в Delphi весьма просто отвечать на события. И не только на события, связанные с мышкой. Например, можно создать обработчик для OnKeyDown (нажата клавиша):

```
procedure TForm1.FormKeyDown(Sender: TObject;
 var Key: Word;
                              Shift: TShiftState);
begin
 MessageDlg(Chr(Key), mtInformation, [mbOk], 0);
end;
```

Теперь, когда вы имеете начальные знания о программировании событий в Delphi, самое время вернуться назад и посмотреть на теорию, стоящую за тем кодом, что вы написали. После получения представления о том, как работает система, можно вернуться к среде Delphi и посмотреть, как использовать полностью имеющиеся возможности.

3.3.1. Понимание событий

Событийное программирование есть не только в Windows, и данную черту можно реализовать не только в операционной системе. Например, любая DOS программа может быть основана на простом цикле, работающем все время жизни программы в памяти. Рассмотрим гипотетический пример, как данный код может выглядеть:

```
 begin
 InitializeMemory;
 repeat
 CheckForMouseEvent(Events);
 CheckForKeyPress(Events)
 HandleEvents(Events);
 until Done := True;
 DisposeMemory;
end.
```

Это типичный пример программы, ориентированной на события. Она начинается и заканчивается инициализацией и освобождением памяти. В программе присутствует простой цикл **repeat..until,** который проверяет появление событий от мыши и клавиатуры и затем дает возможность программисту ответить на эти события.

Переменная Events может быть записью с простой структурой:

```
TEvent = record
 X, Y: Integer;
 MouseButton: TButton;
 Key: Word;
end;
```

Тип TButton, указанный выше, можно декларировать так:

```
TButton = (lButton, rButton);
```

Эти структуры позволяют проследить, где находится мышь, каково состояние ее кнопок, и значение нажатой клавиши на клавиатуре.

Конечно, это пример весьма простой структуры, но заложенные здесь принципы отражают то, что происходит внутри Windows или внутри других систем, ориентированных на события, вроде Turbo Vision. Если бы, приведенная выше программа, была редактором текста, то обработчик HandleEvent для такой программы мог бы иметь вид:

```
procedure HandleEvent(Events: TEvent);
begin
 case Events.Key of
 'A'..'z': Write(Events.Key);
 EnterKey: Write(CarriageReturn);
 EscapeKey: Done := True;
 end;
end;
```

Согласно коду выше, программа будет печатать букву ‘a’ при нажатии этой клавиши и перейдет на новую строку, если нажата клавиша ‘Enter’. Нажатие ‘Esc’ приведет к завершению программы.

Код такого типа может быть весьма удобным, в частности, если вы пишете программу, в которой требуется анимация. Например, если нужно перемещать несколько картинок по экрану, то может понадобиться сдвинуть их на несколько точек, затем проверить, нажимал ли пользователь кнопки. Если такое событие было, то его можно обработать, если нет, то двигать дальше.

Надеюсь, что приведенный пример дает некоторое понимание работы ориентированной на события системы. Единственное, что осталось, пропущенным, - почему система Windows так спроектирована.

Одной из важных причин, почему Microsoft сделал Windows по такой схеме, является тот факт, что множество задач работает в среде одновременно. В многозадачных системах операционная система должна знать, щелкнул ли пользователь мышкой на определенное окно. Если данное окно было частично перекрыто другим, то это становится известно операционной системе и она перемещает окно на передний план. Понятно, что неудобно заставлять само окно выполнять эти действия. Операционной системе лучше обрабатывать все нажатия клавиш и кнопок на мыши и затем передавать их в остальные программы в виде событий.

Если кратко, программист в Windows почти никогда не должен напрямую проверять “железо”. Система выполняет эту задачу и передает информацию программе в виде сообщений.

Когда пользователь щелкает мышкой, операционная система обрабатывает это событие и передает его в окно, которое должно обработать данное событие. Созданное сообщение, в данном случае, пересылается в некую процедуру DefWindowProc окна (default window procedure). DefWindowProc - аналог процедуры HandleEvent из примера, приведенного выше.

Каждое окно в Windows имеет свою DefWindowProc. Чтобы полностью понять данное утверждение, представьте, что каждая кнопка, каждый ListBox, каждое поле ввода и т.д. на самом деле являются окнами и имеют свою процедуру DefWindowProc. Это весьма гибкая и мощная система, но она может заставить программиста писать весьма сложный код. Delphi дает возможность быть защищенным от такой структуры программы.

Почти все, что происходит в Windows принимает форму сообщений и, если вы хотите их использовать в Delphi (в большей мере это необходимо при написании своих собственных компонентов), то нужно понять, как эти сообщения работают.

Если посмотреть на DefWindowProc в справочнике по Windows API, то можно увидеть следующее определение:

```
function DefWindowProc(Wnd: HWnd; Msg, wParam: Word;
 lParam: LongInt);
```

Каждое сообщение, посылаемое в окно, состоит из четырех частей: первая часть - handle окна, получающего сообщение, Msg сообщает, что произошло а третья и четвертая части (wParam и lParam) содержат дополнительную информацию о событии. Вместе эти четыре части являются аналогом показанной выше структуры TEvent.

Вторая часть сообщения имеет длину 16 бит и сообщает, что за событие произошло. Например, если нажата левая кнопка на мыши, то переменная Msg содержит значение WM_LBUTTONDOWN. Существуют десятки различного типа сообщений и они называются вроде WM_GETTEXT, WM_HSCROLL, WM_GETTEXTLENGTH и т.п. Список всех сообщений можно видеть в справочнике по Windows API (on-line help).

Последние две переменные, длиной 16 и 32 бита, называются wParam и lParam. Они сообщают программисту важную дополнительную информацию о

каждом событии. Например, при нажатии кнопки мыши, lParam содержит координаты указателя мыши.

Одна из хитростей заключается в том, как выделить нужную информацию из этих переменных. В большинстве случаев Delphi освобождает вас от необходимости выполнять данную задачу. Например, в обработчике события OnMouseDown для формы вы просто используете координаты X и Y. Как программисту, вам не нужно прилагать усилия для получения сообщения и связанных с ним параметров. Все, что связано с событиями, представлено в простом и непосредственном виде:

```
procedure TForm1.FormMouseDown(Sender: TObject;
Button: TMouseButton;
Shift: TShiftState;
X, Y: Integer);
```

Итак, если подвести итог, то должно стать ясным следующее:

- Windows является системой, ориентированной на события;
- события в Windows принимают форму сообщений;
- в недрах VCL Delphi сообщения Windows обрабатываются и преобразуются в более простую для программиста форму;
- обработка событий в Delphi сводится к написанию для каждого объекта своих обработчиков;
- события в программе на Delphi вызываются не только сообщениями Windows, но и внутренними процессами.

3.3.2. Обработка сообщений Windows в Delphi

Конечно, нельзя придумать такую библиотеку объектов, которые бы полностью соответствовали потребностям программистов. Всегда возникнет необходимость дополнения или изменения свойств и поведения объектов. В этом случае, так же, как и при создании собственных своих компонентов в Delphi, часто требуется обрабатывать сообщения Windows. Поскольку Object Pascal является развитием и продолжением Borland Pascal 7.0, то это выполняется сходным с BP способом.

Общий синтаксис для декларации обработчика сообщений Windows:

```
procedure Handler_Name(var Msg : MessageType);
message WM_XXXXX;
```

Handler_Name обозначает имя метода; Msg - имя передаваемого параметра; MessageType - какой либо тип записи, подходящий для данного сообщения; директива message указывает, что данный метод является обработчиком

сообщения; WM_XXXXX - константа или выражение, которое определяет номер обрабатываемого сообщения Windows.

Рассмотрим обработку сообщений на примере. Например, при нажатии правой кнопки мыши на форме в программе появляется всплывающее меню (pop-up menu, если оно было привязано к этой форме). Программист может захотеть привязать к правой кнопке какое-нибудь другое событие. Это можно сделать так:

```
type
TForm1 = class(TForm)
 PopupMenu1: TPopupMenu;
 MenuItem1: TMenuItem;
 MenuItem2: TMenuItem;
 MenuItem3: TMenuItem;
private
{ Private declarations }
 procedure WMRButtonDown(var Msg : TWMMouse); message
WM_RBUTTONDOWN;
public
{ Public declarations }
end;
```

Подчеркнут код, добавленный в декларацию объекта TForm1 вручную. Далее, в секции **implementation,** нужно написать обработчик:

```
procedure TForm1.WMRButtonDown(var Msg : TWMMouse);
begin
MessageDlg('Right mouse button click.', mtInformation,
 [mbOK], 0);
end;
```

В данном случае при нажатии правой кнопки мыши будет появляться диалог.

Вообще-то, у класса TForm уже есть унаследованный от дальнего предка обработчик данного события, который называется точно также и вызывает то самое pop-up меню. Если в новом обработчике сообщения нужно выполнить действия, которые производились в старом, то для этого применяется ключевое слово **inherited**. Если слегка модифицировать наш обработчик, то после диалога будет появляться pop-up меню:

```
procedure TForm1.WMRButtonDown(var Msg : TWMMouse);
begin
 MessageDlg('Right mouse button click.', mtInformation,
 [mbOK], 0);
```

```
 inherited;
end;
```

Однако есть еще способ обработки всех сообщений, которые получает приложение. Для этого используется свойство OnMessage объекта Application, который автоматически создается при запуске программы. Если определен обработчик события OnMessage, то он получает управление при любом событии, сообщение о котором направлено в программу. Следующий код будет приводить к появлению диалога при двойном щелчке мыши на любом объекте в приложении.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.OnMessage:=AOM;
end;

 procedure TForm1.AOM(var Msg: TMsg; var Handled: Boolean);
 begin
 Handled:=False;
 if Msg.Message = WM_LBUTTONDBLCLK then begin
 MessageDlg('Double click.', mtInformation, [mbOK], 0);
 Handled:=True;
 end;
 end;
```

Конечно, в обработчике нельзя выполнять операции, требующие длительного времени, поскольку это приведет к замедлению выполнения всего приложения.

Краткие выводы

Свойство является важным атрибутом компонента. Для пользователя (программиста) свойство выглядит как простое поле какой-либо структуры, содержащее некоторое значение

Создание программы в Delphi сводится к "нанесению" компонентоа на форму (которая, кстати, также является компонентом) и настройке взаимодействия между ними путем: изменения значения ***свойств*** этих компонентов, написания адекватных реакций на ***события.***

Существует несколько типов свойств, в зависимости от их "природы", т.е. внутреннего устройства.

- **Простые свойства** - это те, значения которых являются числами или строками. Например, свойства *Left* и *Top* принимают целые значения, определяющие положение левого верхнего угла компонента или формы. Свойства *Caption* и *Name* представляют собой строки и определяют заголовок и имя компонента или формы.

- **Перечислимые свойства** - это те, которые могут принимать значения из предопределенного набора (списка). Простейший пример - это свойство типа *Boolean*, которое может принимать значения *True* или *False*.
- **Вложенные свойства** - это те, которые поддерживают вложенные значения (или объекты). Object Inspector изображает знак "+" слева от названия таких свойств. Имеются два вида таких свойств: *множества* и *комбинированные значения*. Object Inspector изображает множества в квадратных скобках. Если множество пусто, оно отображается как []. Установки для вложенных свойств вида "множество" обычно имеют значения типа Boolean. Наиболее распространенным примером такого свойства является свойство *Style* с вложенным множеством булевых значений. Комбинированные значения отображаются в Инспекторе Объектов как коллекция некоторых величин, каждый со своим типом данных (рис 1). Некоторые свойства, например, *Font*, для изменения своих значений имеют возможность вызвать диалоговое окно. Для этого достаточно щелкнуть маленькую кнопку с тремя точками в правой части строки Инспектора Объектов, показывающей данное свойство.

Класс - это категория объектов, обладающих одинаковыми свойствами и поведением. При этом ***объект*** представляет собой просто экземпляр какого-либо класса. Например, в Delphi тип “форма” (окно) является классом, а переменная этого типа - объектом. **Метод** - это процедура, которая определена как часть класса и инкапсулирована (содержится) в нем. Методы манипулируют полями и свойствами классов (хотя могут работать и с любыми переменными) и имеют автоматический доступ к *любым* полям и методам своего класса. Доступ к полям и методам других классов зависит от уровня “защищенности” этих полей и методов. Пока же для нас важно то, что методы можно создавать как визуальными средствами, так и путем написания кода вручную.

Ключевые слова

Простые, перечислимые и вложенные свойства, кнопки и выпадающий список, изменение положения полос, управление свойствами визуальных компонентов в режиме выполнения.

Объект, класс, техника написания методов в Delphi, экземпляр класса, передача параметров, информация периода выполнения.

Событие, обработка событий, многозадачные системы, структура, состояние и поведение объекта.

Вопросы для самоконтроля

1. Какие типы свойств в Delphi Вы знаете?
2. В каких режимах Delphi позволяет легко манипулировать свойствами компонентов?

3. Какое диалоговое окно в Delphi дает возможность изменить цвет «во время выполнения»?

4. Как происходит управление свойствами визуальных компонентов в режиме выполнения Delphi?

5. Как Вы можете определить базовое понятие объектно – ориентированного программирования **класс?** И как в этом случае представляется **объект**?

6. Дайте характеристику понятию «методы в Delphi».
7. Из скольких частей может состоять заголовок процедуры?
8. Что подразумевается под понятием Событие в Delphi?
9. Что обозначает высказывание «программирование событий» в Delphi?
10. Как происходит обработка сообщений Windows в Delphi?

Рекомендуемая литература

1. Ален Э. Типичные ошибки проектирования. Библиотека программиста. − М.: Питер, 2004.

2.Брауде Э. Технология разработки программного обеспечения. − М.: Питер, 2004. - 325 с.

3. Бобров И.С. Delphi 7. Учебный курс. Москва. Санкт-Петербург, Нижний Новгород, Воронеж: Питер. 2003.- 625

4. Гради Буч. Объектно – ориентированный анализ и проектирование с примерами приложений на С++. − М.: Бином. 1998

5. Джон Влиссидес, Эрих Гамма, Ричард Хелм, Ральф Джонсон. Приемы объектно – ориентированного проектирования. Паттерны проектирования. − М.: Питер. 2003. - 256 с.

6. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно- ориентированное программирование. − М.: Издательство МГТУ имени Н.Э. Баумана. 2003.- 320с.

7.Кьоу Дж.,Джеанини М.Объектно-ориентированное программирование. Просто и понятно. − М.: Питер. 2005. -403с.

8. Лесневский А.С. Объектно – ориентированное программирование для начинающих. − М.: Бином. Лаборатория знаний. 2005. - 382 с.

9.Майер Б. Объектно – ориентированное программирование. Концепции разработки. − М.: Русская редакция. 2004.- 456 с.

10.Синтес Антони. Освой самостоятельно объектно – ориентированное программирование за 21 день. своих. − М.: Вильямс · 2002. -284с.

11.http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».

12.http://.borland.com/delphi_net/ - Официальный сайт Borland Delphi

13.http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

14. http://www/spb.runnet.ru – сервер Санк – Петербургского центра научно – технической информации

15. http://www.ournet.md/~delphi - Сайт часто задаваемых вопросов и ответов по DELPHI, документация по функциям Windows API, уроки по Delphi, Delphi 4 - интерактивно, работа с DirectX и MMX.

Глава 4. РАБОТА С БИБЛИОТЕКАМИ DDE И DLL В DELPHI

4.1. Библиотека управления динамическим обменом данными (DDE) в Delphi

4.1.1. Понятие DDE и использование в Delphi.

Аббревиатура DDEML обозначает Dynamic Data Exchange Management Library (библиотека управления динамическим обменом данными). DDEML это надстройка над сложной системой сообщений, называемой Dynamic Data Exchange (DDE). Библиотека, содержащая DDE, была разработана для усиления возможностей первоначальной системы сообщений Windows.

DDE дает возможность перейти через рамки приложения и взаимодействовать с другими приложениями и системами Windows.

Dynamic Data Exchange получило свое имя потому, что позволяет двум приложениям обмениваться данными (текстовыми, через глобальную память) динамически во время выполнения. Связь между двумя программами можно установить таким образом, что изменения в одном приложении будут отражаться во втором. Например, если Вы меняете число в электронной таблице, то во втором приложении данные обновятся автоматически и отобразят изменения. Кроме того, с помощью DDE можно из своего приложения управлять другими приложениями, такими, как Word for Windows, Report Smith, Excel и др.

Приложение, получающее данные из другого приложения по DDE и/или управляющее другим приложением с помощью команд через DDE является DDE-клиентом. В этом случае второе приложение является DDE-сервером. Одно и то-же приложение может быть одновременно и сервером, и клиентом (например, MicroSoft Word). Построение DDE-серверов и DDE-клиентов удобно рассмотреть на примере, поставляемом с Delphi (каталог x:\delphi\demos\ddedemo). Сперва давайте рассмотрим логику работы примера. Для начала нужно откомпилировать проекты DDESRVR.DPR и DDECLI.DPR, после этого запустите программу DDECLI.EXE (DDE-клиент) и выберите пункт меню File|New Link. При этом должна запуститься программа DDESRVR (DDE-сервер). Если теперь редактировать текст в окне сервера, то изменения мгновенно отразятся в приложении-клиенте (см. рис.1, 2)

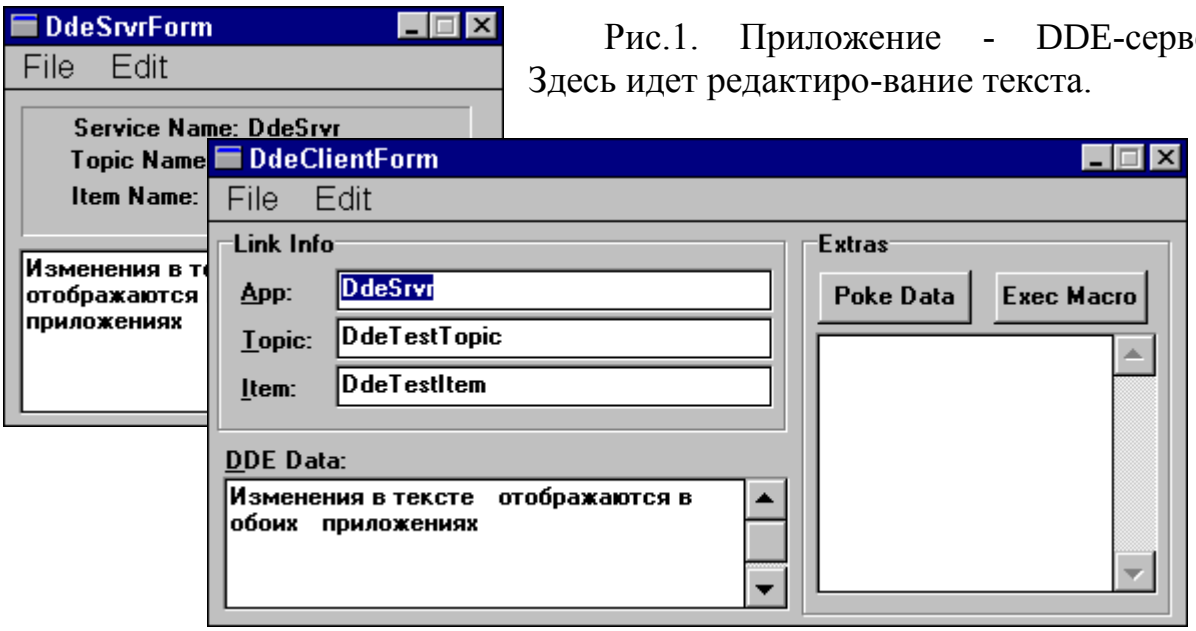


Рис.1. Приложение - DDE-сервер. Здесь идет редактиро-вание текста.

Рис.2. Приложение - DDE-клиент. Здесь отображаются изменения.

Пример демонстрирует и другие возможности DDE: пересылка данных из клиента на сервер (Poke Data); наберите любой текст в правом окне DDE-клиента и нажмите кнопку Poke Data, этот текст появится в окне сервера.

- исполнение команд (макросов) на сервере; наберите любой текст в правом окне DDE-клиента и нажмите кнопку Exec Macro, DDE-сервер выдаст соответствующее диалоговое окно.

- установление связи через Clipboard; закройте оба DDE-приложения и запустите их заново, затем в DDE-сервере выберите пункт меню Edit|Copy, далее в клиенте выберите пункт меню Edit|Paste Link.

Теперь давайте рассмотрим эти демонстрационные программы с технической точки зрения и узнаем, каким образом в Delphi можно создать DDE-приложения. Начнем с DDE-сервера.

4.1.2. DDE-серверы

На рис.3 представлен пример DDE-сервера во время дизайна в среде Delphi.

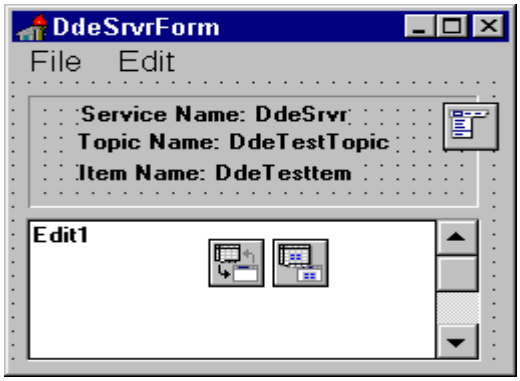


Рис.3. DDE-сервер в среде Delphi.

Для построении DDE-сервера в Delphi имеются два объекта, расположенные на странице System Палитры Компонент - TDdeServerConv и TDdeServerItem. Обычно в проекте используется один объект TDdeServerConv и один или более TDdeServerItem. Для получения доступа к сервису DDE-сервера, клиенту потребуется знать несколько параметров : имя сервиса (Service Name) - это имя приложения (обычно - имя выполняемого файла без расширения EXE, возможно с полным путем); Topic Name - в Delphi это имя компоненты TDdeServerConv; Item Name - в Delphi это имя нужной компоненты TDdeServerItem.

Назначение объекта TDdeServerConv - общее управление DDE и обработка запросов от клиентов на выполнение макроса. Последнее выполняется в обработчике события OnExecuteMacro, например, как это сделано в нашем случае:

```
procedure TDdeSrvrForm.doMacro(Sender: TObject;
 Msg: TStrings);
var
 Text: string;
begin
 Text := '';
 if Msg.Count > 0 then Text := Msg.Strings[0];
 MessageDlg ('Executing Macro - ' + Text, mtInformation,
 [mbOK], 0);
end;
```

Объект TDdeServerItem связывается с TDdeServerConv и определяет, что, собственно, будет пересылаться по DDE. Для этого у него есть свойства Text и Lines. (Text имеет то же значение, что и Lines[0].) При изменении значения этих свойств автоматически происходит пересылка обновленных данных во все приложения-клиенты, установившие связь с сервером. В нашем приложении

Изменение значения свойства Lines происходит в обработчике события OnChange компонента Edit1:

```
procedure TDdeSrvrForm.doOnChange(Sender: TObject);
begin
 if not FInPoke then
 DdeTestItem.Lines := Edit1.Lines;
end;
```

Этот же компонент отвечает за получение данных от клиента, в нашем примере это происходило при нажатии кнопки Poke Data, это выполняется в обработчике события OnPokeData:

```
procedure TDdeSrvrForm.doOnPoke(Sender: TObject);
begin
 FInPoke := True;
 Edit1.Lines := DdeTestItem.Lines;
```

```
 FInPoke := False;
end;
```

И последнее - для установление связи через Clipboard служит метод CopyToClipboard объекта TDdeServerItem. Необходимая информация помещается в Clipboard и может быть вызвана из приложения-клиента при установлении связи. Обычно в DDE-серверах для этого есть специальный пункт меню Paste Special или Paste Link.

Итак, мы рассмотрели пример полнофункционального DDE-сервера, построенного с помощью компонентов Delphi. Весьма часто существующие DDE-серверы неполностью реализуют возможности DDE и предоставляют только часть сервиса. Например, ReportSmith позволяет по DDE только выполнять команды (макросы).

4.1.3. DDE-клиенты

На рис.4 представлен пример DDE-клиента во время дизайна в среде Delphi.

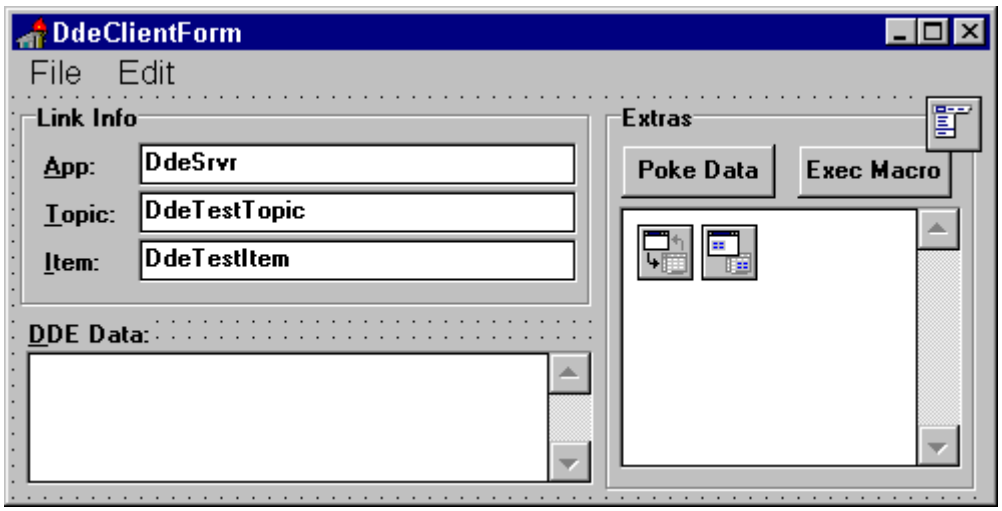


Рис.4. DDE-клиент в среде Delphi.

Для построения DDE-клиента в Delphi используются два компонента: TDDEClientConv и TDDEClientItem. Аналогично серверу, в программе обычно используются один объект TDDEClientConv и один и более связанный ним TDDEClientItem.

TDDEClientConv служит для установления связи с сервером и общим управлением DDE-связью. Установить связь с DDE-сервером можно как в процессе дизайна, так и во время выполнения программы, причем двумя способами. Первый способ - заполнить вручную необходимые свойства компонента: это DdeService, DdeTopic и ServiceApplication. Во время дизайна щелкните дважды на одно из первых двух свойств в Инспекторе Объектов - Вы получите диалог для определения DDE-связи (см. рис.5).

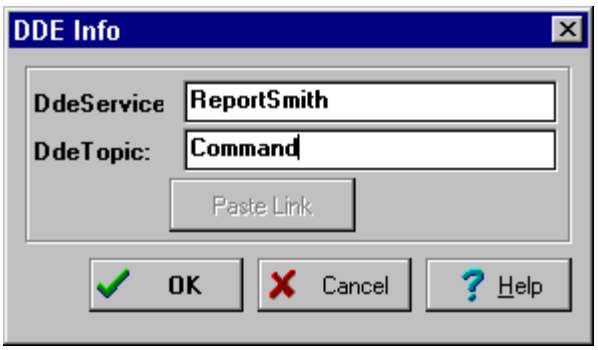


Рис.5. Диалог для установления связи с DDE-сервером (Report Smith).

Укажите в диалоге имена DDE Service и DDE Topic. Их можно узнать из документации по тому DDE-серверу, с которым Вы работаете. В случае DDE-сервера, созданного на Delphi, это имя программы (без .EXE) и имя объекта TDdeServerConv. Для установления связи через Clipboard в диалоге есть специальная кнопка Past Link. Ею можно воспользоваться, если Вы запустили DDE-сервер, сохранили каким-то образом информацию о связи и вошли в этот диалог. Например, если DDE-сервером является DataBase Desktop, то нужно загрузить в него какую-нибудь таблицу Paradox, выбрать любое поле и пункт меню Edit|Copy. После этого войдите в диалог и нажмите кнопку Paste Link. Поля в диалоге заполнятся соответствующим образом.

Свойство ServiceApplication заполняется в том случае, если в поле DDEService содержится имя, отличное от имени программы, либо если эта программа не находится в текущей директории. В данном поле указываются полный путь и имя программы без расширения (.EXE). При работе с Report Smith здесь нужно указать, например : C:\RPTSMITH\RPTSMITH

Данная информация нужна для автоматического запуска сервера при установлении связи по DDE, если тот еще не был запущен.

В нашей демо-программе связь устанавливается во время выполнения программы в пунктах меню File|New Link и Edit|Paste Link. В пункте меню File|New Link программно устанавливается связь по DDE с помощью соответствующего метода объекта TDdeServerConv, OpenLink делать не надо, поскольку свойство ConnectMode имеет значение ddeAutomatic:

```
procedure TFormD.doNewLink(Sender: TObject);
begin
 DdeClient.SetLink(AppName.Text, TopicName.Text);
 DdeClientItem.DdeConv := DdeClient;
 DdeClientItem.DdeItem := ItemName.Text;
end;
```

Здесь же заполняются свойства объекта TDdeClietItem.

В пункте меню Edit|Past Link программно устанавливается связь по DDE с использованием информации из Clipboard:

```
procedure TFormD.doPasteLink(Sender: TObject);
var
 Service, Topic, Item : String;
begin
 if GetPasteLinkInfo (Service, Topic, Item) then
 begin
 AppName.Text := Service;
 TopicName.Text := Topic;
 ItemName.Text := Item;
 DdeClient.SetLink (Service, Topic);
 DdeClientItem.DdeConv := DdeClient;
 DdeClientItem.DdeItem := ItemName.Text;
 end;
end;
```

После того, как установлена связь, нужно позаботиться о поступающих по DDE данных, это делается в обработчике события OnChange объекта TDdeClietItem:

```
procedure TFormD.DdeClientItemChange(Sender: TObject);
begin
 DdeDat.Lines := DdeClientItem.Lines;
end;
```

Это единственная задача объекта TDdeClientItem.

На объект TDdeClientConv возлагаются еще две задачи : пересылка данных на сервер и выполнение макросов, для чего у данного объекта имеются соответствующие методы. Посмотрим, как это можно было бы сделать. Выполнение макроса на сервере:

```
procedure TFormD.doMacro(Sender: TObject);
begin
 DdeClient.ExecuteMacroLines(XEdit.Lines, True);
end;
```

Пересылка данных на сервер:

```
procedure TFormD.doPoke (Sender: TObject);
begin
 DdeClient.PokeDataLines(DdeClientItem.DdeItem,XEdit.Lines);
end;
```

Управление ReportSmith по DDE

В прилагаемом примере run-time версия ReportSmith выполняет команду, переданную по DDE. Имена DDE сервиса для ReportSmith и некоторых других приложений можно узнать в Справочнике в среде ReportSmith.

Перед запуском примера следует правильно установить в свойстве ServiceApplication путь к run-time версии ReportSmith и в тексте программы в строке

```
Cmd:='LoadReport
"c:\d\r\video\summary.rpt","@Repvar1=<40>,@Repvar2=<'#39'Smith'#39'>'"#0;
```

правильно указать путь к отчету.

4.2. Работа с динамически подсоединяемыми библиотеками (dynamic link libraries – DLL) в Delphi

4.2.1. Понятие DLL

Вспомним процесс программирования в DOS. Преобразование исходного текста программы в машинный код включал в себя два процесса - компиляцию и линковку. В процессе линковки редактор связей, компоновавший отдельные модули программы, помещал в код программы не только объявления функций и процедур, но и их полный код. Вы готовили, таким образом, одну программу, другую, третью ... И везде код одних и тех же функций помещался в программу полностью (см. рис 1).

Программа1	Программа2
…	…
MyFunc(…)	MyFunc(…)
…	…
код функции MyFunc	код функции MyFunc
код других функций	код других функций

Рис.1 Вызов функций при использовании статической компоновки

В многозадачной среде такой подход был бы, по меньшей мере, безрассудным, так как очевидно, что огромное количество одних и тех же функций, отвечающих за прорисовку элементов пользовательского интерфейса, доступ к системным ресурсам и т.п. дублировались бы полностью во всех приложениях, что привело бы к быстрому истощению самого дорогого ресурса - оперативной памяти. В качестве решения возникшей проблемы еще на UNIX-подобных платформах была предложена концепция динамической компоновки (см. рис . 2).

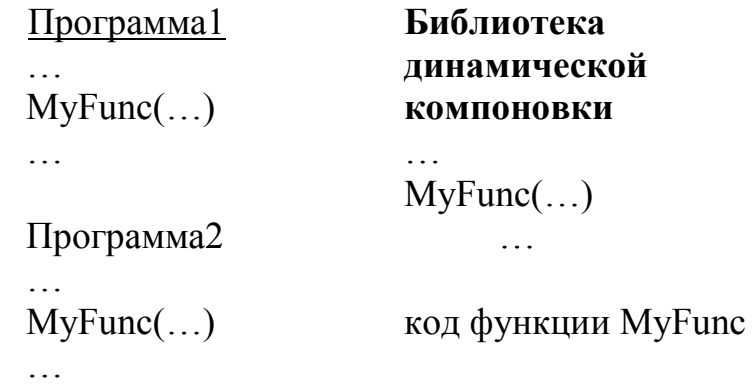


Рис. 2. Вызов функций при использовании динамической компоновки.

Но чем же отличаются Dynamic Link Library (DLL) от обычных приложений? Для понимания этого требуется уточнить понятия задачи (task), экземпляра (копии) приложения (instance) и модуля (module).

При запуске нескольких экземпляров одного приложения Windows загружает в оперативную память только одну копию кода и ресурсов - модуль приложения, создавая несколько отдельных сегментов данных, стека и очереди сообщений (см. рис. 3), каждый набор которых представляет собой задачу в понимании Windows. Копия приложения представляет собой контекст, в котором выполняется модуль приложения.

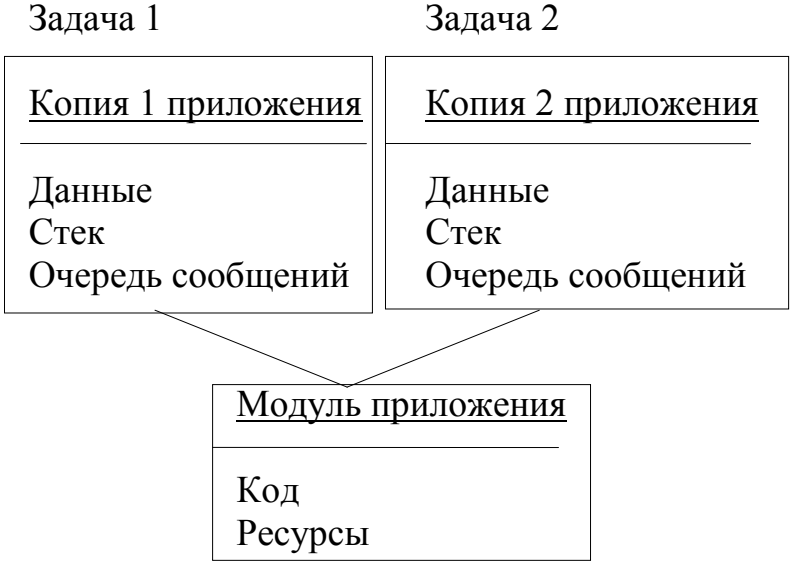


Рис.3. Копии и модуль приложения.

DLL - библиотека также является модулем. Она находится в памяти в единственном экземпляре и содержит сегмент кода и ресурсы, а также сегмент данных (см. рис. 4).

DLL-библиотека

Рис.4. Структура DLL в памяти.

DLL - библиотека, в отличие от приложения, не имеет ни стека, ни очереди сообщений. Функции, помещенные в DLL, выполняются в контексте вызвавшего приложения, пользуясь его стеком. Но эти же функции используют сегмент данных, принадлежащий библиотеке, а не копии приложения.

В силу такой организации DLL, экономия памяти достигается за счет того, что все запущенные приложения используют один модуль DLL, не включая те или иные стандартные функции в состав своих модулей.

Часто, в виде DLL создаются отдельные наборы функций, объединенные по тем или иным логическим признакам, аналогично тому, как концептуально происходит планирование модулей (в смысле unit) в Pascal. Отличие заключается в том, что функции из модулей Pascal компонуются статически - на этапе линковки, а функции из DLL компонуются динамически, то есть в run-time.

Достаточно взглянуть на список файлов, расположенных в системном каталоге Windows, - порой количество используемых операционной системой динамических библиотек достигает нескольких сотен. DLL являются неотъемлемой частью функционирования операционных систем семейства Microsoft Windows.

Итак, DLL - это один или несколько логически законченных фрагментов кода, сохраненных в файле с расширением.dll. Этот код может быть запущен на выполнение в процессе функционирования какой-либо другой программы (такие приложения называются вызывающими по отношению к библиотеке), но сама DLL не является запускаемым файлом.

Существует два типа динамических библиотек - исполняемые и библиотеки ресурсов. Однако это не означает, что в одном файле не может находиться и код некоторой функции и какие-либо ресурсы. Просто иногда бывает удобно разнести реализацию исполняемых процедур и используемые приложением ресурсы в разные файлы.

Итак, процедуры и функции, содержащиеся в динамической библиотеке, можно разделить на два типа: те, которые могут быть вызваны из других приложений. Рассмотрим следующий пример:

```
Screen.Cursors[myCursor] := LoadCursor(HInstance, MYCURSOR');
```

LoadCursor - функция Windows API, которая вызывается приложением из динамической библиотеки User 32.dll. Кстати, примером хранимых в

динамической библиотеке ресурсов могут являться такие стандартные диалоги Windows, как диалог открытия файла, печати или настройки принтера, которые находятся в файле Comctl32.dll. Однако многие прикладные разработчики используют функции вызова форм этих диалогов, совершенно не задумываясь, где хранится их описание. Второй тип процедур - это те, которые используются только внутри самого файла библиотеки.

4.2.2. Основные преимущества использования DLL

Применение DLL имеет множество положительных моментов.

Во-первых, это повторное использование кода. Думаем, нет необходимости пояснять удобство использования один раз разработанных процедур и функций при создании нескольких приложений. Кроме того, имеется возможность использовать некоторые из своих библиотек, не раскрывая исходных кодов. Чем тогда это лучше компонентов? Тем, что функции, хранящиеся в библиотеке, могут быть вызваны на выполнение из приложений, разработанных не на Object Pascal, а, например, с использованием C++Builder, Visual Basic, Visual C++ и т.д. Такой подход накладывает некоторые ограничения на принцип разработки библиотеки, но это возможно.

Во-вторых, использование DLL предоставляет возможность использования один раз загруженного в оперативную память кода несколькими приложениями. К примеру, если вы разрабатываете программное обеспечение для большого предприятия, то вполне возможно, что в различных созданных вами приложениях будут использоваться одни и те же функции, процедуры, формы и другие ресурсы. Естественно, что выделение общих для нескольких приложений данных в DLL может привести к экономии как дискового пространства, так и оперативной памяти, порой весьма даже существенному.

В-третьих, следует поговорить вот о чем. Всего несколько лет назад при разработке программного обеспечения вы могли совершенно не волноваться по поводу распространения ваших продуктов где-либо, кроме вашей страны. Хотим сказать, что проблема перевода на другие языки текста на элементах управления (пункты меню, кнопки, выпадающие списки, подсказки), сообщений об ошибках и т.д. не стояла так остро, как сейчас. Однако, с повсеместным внедрением Интернета у пользователей появилась возможность быстрой передачи готовых программных продуктов практически в любую точку мира. И что будут делать с вашей программой где-нибудь в Объединенных Арабских Эмиратах, если кроме как по-русски, она с пользователем общаться не умеет? Вы сами можете оценить этот эффект, если хоть раз на экране вашего компьютера вместо с детства знакомого русского языка появится "арабская вязь" (например, из-за "сбоя" шрифтов). Итак, уже сейчас вы должны планировать возможность распространения ваших приложений в других странах (если, конечно, у вас есть желание получить как можно больше прибыли). Соответственно, встает вопрос быстрого перевода

интерфейса вашей программы на другие языки. Одним из путей может являться создание ресурсов интерфейсов внутри DLL. К примеру, можно создать одно приложение, которое в зависимости от версии динамической библиотеки, будет выводить сообщения на различных языках.

Естественно, выше приведены лишь некоторые из аргументов в пользу использования динамически подключаемых библиотек при разработке приложений.

4.2.3. Основы разработки DLL

Разработка динамических библиотек не представляет собой некий сверхсложный процесс, доступный лишь избранным.

Как и любой другой модуль, модуль динамической библиотеки имеет фиксированный формат. Взгляните на листинг, представленный ниже.

```
library MyFirstDLL;
uses
 SysUtils,
 Classes,
 Forms,
 Windows;
procedure HelloWorld(AForm : TForm);
 begin
 MessageBox(AForm.Handle, Hello world!',
 DLL Message Box', MB_OK or MB_ICONEXCLAMATION);
 end;
exports
 HelloWorld;
begin
end.
```

Первое, на что следует обратить внимание, это ключевое слово library, находящееся вверху страницы. Library определяет этот модуль как модуль библиотеки DLL. Далее идет название библиотеки. В нашем примере мы имеем дело с динамической библиотекой, содержащей единственную процедуру: HelloWorld. Причем обратите внимание, что данная процедура по структуре ничем не отличается от тех, которые вы помещаете в модули своих приложений. Ключевое слово exports сигнализирует компилятору о том, что перечисленные ниже функции и/или процедуры должны быть доступны из вызывающих приложений (т.е. они как бы "экспортируются" из библиотеки).

И, наконец, в конце модуля можно увидеть ключевые слова begin и end. Внутри данного блока вы можете поместить код, который должен выполняться в процессе загрузки библиотеки. Достаточно часто этот блок остается пустым.

Как уже отмечалось выше, все процедуры и функции, помещаемые в DLL, могут быть разделены на две группы: экспортируемые (вызываемые из других приложений) и локальные. Естественно, внутри библиотеки также могут быть описаны классы, которые, в свою очередь, содержат методы.

Описание и реализация процедур и функций, вызываемых в пределах текущей DLL, ничем не отличаются от их аналогов в обычных проектах-приложениях. Их специфика заключается лишь в том, что вызывающая программа не будет иметь к ним доступа. Она просто не будет ничего знать об их существовании, так же, как одни классы ничего не знают о тех методах, которые описаны в секции private других классов.

В дополнение к процедурам и функциям, DLL может содержать глобальные данные, доступ к которым разрешен для всех процедур и функций в библиотеке. Для 16-битных приложений эти данные существовали в единственном экземпляре независимо от количества загруженных в оперативную память программ, которые используют текущую библиотеку. Иными словами, если одна программа изменяет значение глобальной переменной a на 100, то для всех остальных приложений a будет значение 100. Для 32-битных приложений это не так. Теперь для каждого приложения создается отдельная копия глобальной области данных.

4.2.4. Экспорт функций из DLL

Как уже отмечалось выше, для экспорта процедур и функций из DLL необходимо использовать ключевое слово export. Еще раз обратите внимание на представленный выше листинг библиотеки MiFirstDll. Поскольку процедура HelloWorld определена как экспортируемая, то она может быть вызвана на выполнение из других библиотек или приложений. Существуют следующие способы экспорта процедур и функций: экспорт по имени и порядковому номеру.

Наиболее распространенный способ экспорта - по имени. Взглянем на приведенный ниже текст:

```
exports
SayHello,
DoSomething,
DoSomethingReallyCool;
```

Следует обратить внимание на то, что Delphi автоматически назначает порядковый номер каждой экспортируемой функции (процедуре) независимо от того, определяете вы его явно или нет. Явное определение индекса позволяет вам лично управлять порядковым номером экспортируемой функции или процедуры.

Для того, чтобы определить выполняется ли ваш кодек в DLL или вызывающем приложении, можно воспользоваться глобальной переменной IsLibrary. Она принимает значение true в том случае, если код вызывается из библиотеки и false в случае выполнения процедуры или функции из вызывающего приложения.

Кроме того, в поставку Delphi входит весьма полезная утилита tdump, которая предоставляет данные о том, какая информация экспортируется из указанной DLL.

4.2.5. Использование DLLProc

Как уже отмечалось, код инициализации динамической библиотеки может быть помещен в блок begin...end. Однако, кроме того зачастую необходимо предусмотреть некоторые действия, выполняемые в процессе выгрузки DLL из оперативной памяти. В отличие от других типов модулей, модуль DLL не имеет секции initialization или finalization. К примеру, вы можете динамически выделить память в главном блоке, однако непонятно, где эта память должна быть освобождена. Для решения данной проблемы существует DLLProc - специальная процедура, вызываемая в определенные моменты функционирования DLL.

Для начала следует сказать о самой причине существования DLLProc. Динамическая библиотека получает сообщения от Windows в моменты своей загрузки и выгрузки из оперативной памяти, а также в тех случаях, когда какой-нибудь очередной процесс, использующий функции и/или ресурсы, хранящиеся в библиотеке, загружается в память. Такая ситуация возможна в том случае, когда библиотека необходима для функционирования нескольких приложений. А для того, чтобы иметь возможность указывать, что именно должно происходить в такие моменты, необходимо описать специальную процедуру, которая и будет ответственна за такие действия. К примеру, она может выглядеть следующим образом:

```
procedure MyFirstDLLProc(Reason: Integer);
begin
 if Reason = DLL_PROCESS_DETACH then
 {DLL is unloading. Cleanup code here.}
end;
```

Однако системе совершенно неочевидно, что именно процедура MyFirstDllProc ответственна за обработку рассмотренных выше ситуаций. Поэтому нужно поставить в соответствие адрес нашей процедуры глобальной переменной DLLProc. Это необходимо сделать в блоке begin...end примерно так:

```
begin
DLLProc := @MyDLLProc;
{ Что-нибудь еще, что должно выполняться в
процессе инициализации библиотеки }
end.
```

Ниже представлен код, демонстрирующий один из возможных вариантов применения DLLProc.

```
library MyFirstDLL;
```

```
uses
 SysUtils,
 Classes,
 Forms,
 Windows;
var
 SomeBuffer : Pointer;
procedure MyFirstDLLProc(Reason: Integer);
begin
 if Reason = DLL_PROCESS_DETACH then
 {DLL is выгружается из памяти.
 Освобождаем память, выделенную под буфер.}
 FreeMem(SomeBuffer);
end;
procedure HelloWorld(AForm : TForm);
begin
 MessageBox(AForm.Handle, Hello world!',
 DLL Message Box', MB_OK or MB_ICONEXCLAMATION);
end;

{Какой-нибудь код, в котором используется
SomeBuffer.}

exports
 HelloWorld;
begin
 {Ставим в соответствие с переменной
 DLLProc адрес нашей процедуры.}
 DLLProc := @MyFirstDLLProc;
 SomeBuffer := AllocMem(1024);
end.
```

Как из этого явствует, в качестве признака того или иного события, в результате которого вызывается процедура MyFirstDll, является значение переменной Reason. Ниже приведены возможные значения данной переменной.

DLL_PROCESS_DETACH - библиотека выгружается из памяти; используется один раз;

DLL_THREAD_ATTACH - в оперативную память загружается новый процесс, использующий ресурсы и/или код из данной библиотеки;

DLL_THREAD_DETACH - один из процессов, использующих библиотеку, "выгружается" из памяти.

4.2.6. Загрузка DLL

Прежде чем начать использование какой-либо процедуры или функции, находящейся в динамической библиотеке, необходимо загрузить DLL в оперативную память. Загрузка библиотеки может быть осуществлена одним из двух способов: статической загрузка и динамической загрузки. Оба метода имеют как преимущества, так и недостатки.

Статическая загрузка означает, что динамическая библиотека загружается автоматически при запуске на выполнение использующего ее приложения. Для того чтобы использовать такой способ загрузки, необходимо воспользоваться ключевым словом external при описании экспортируемой из динамической библиотеки функции или процедуры. DLL автоматически загружается при старте программы, тогда можно использовать любые экспортируемые из нее подпрограммы точно так же, как если бы они были описаны внутри модулей приложения. Это наиболее легкий способ использования кода, помещенного в DLL. Недостаток метода заключается в том, что если файл библиотеки, на который имеется ссылка в приложении, отсутствует, программа откажется загружаться.

Смысл динамического метода заключается в том, что пользователь загружает библиотеку не при старте приложения, а в тот момент, когда это действительно необходимо. Естественно, ведь если функция, описанная в динамической библиотеке, используется только при 10% запусков программы, то совершенно не имеет смысла применить статический метод загрузки. Выгрузка библиотеки из памяти в данном случае также осуществляется под контролем пользователя. Еще одно преимущество такого способа загрузки DLL - это уменьшение (по понятным причинам) времени старта приложения пользователя. Какие же у этого способа имеются недостатки? Основной - это то, что использование данного метода является более хлопотным, чем рассмотренная выше статическая загрузка. Сначала необходимо воспользоваться функцией Windows API LoadLibrary. Для получения указателя на экспортируемой процедуры или функции должна использоваться функция GetProcAddress. После завершения использования библиотеки DLL должна быть выгружена с применением FreeLibrary.

4.2.7. Вызов процедур и функций, загруженных из DLL

Способ вызова процедур и функций зависит от того, каким образом была загружена динамическая библиотека, в которой эти подпрограммы находятся.

Вызов функций и процедур из статически загруженных DLL достаточно прост. Первоначально в приложении должно содержаться описание экспортируемой функции (процедуры). После этого можно их использовать точно так же, как если бы они были описаны в одном из модулей приложения пользователя. Для импорта функции или процедуры, содержащейся в DLL, необходимо использовать модификатор external в их объявлении. К примеру,

для рассмотренной нами выше процедуры HelloWorld в вызывающем приложении должна быть помещена следующая строка:

```
    procedure SayHello(AForm : TForm); external
myfirstdll.dll';
```

Ключевое слово external сообщает компилятору, что данная процедура может быть найдена в динамической библиотеке (в нашем случае - myfirstdll.dll). Далее вызов этой процедуры выглядит следующим образом:

```
...
HelloWorld(self);
...
```

При импорте функции и процедур нужно быть особенно внимательным при написании их имен и интерфейсов. Дело в том, что в процессе компиляции приложения не производится проверки на правильность имен объектов, экспортируемых из DLL, и если неправильно была описана какая-нибудь функция, то исключение будет сгенерировано только на этапе выполнения приложения.

Импорт из DLL может проводиться по имени процедуры (функции), порядковому номеру или с присвоением другого имени.

В первом случае просто объявляется имя процедуры и библиотеки, из которой ее импортируете (мы это рассмотрели чуть выше). Импорт по порядковому номеру требует от пользователя указание самого этого номера:

```
procedure HelloWorld(AForm : TForm);
external myfirstdll.dll' index 15;
```

В данном случае имя, которое дается процедуре при импорте, не обязательно должно совпадать с тем, которое было указано для нее в самой DLL. Т.е. приведенная выше запись означает, что вы импортируете из динамической библиотеки myfirstdll.dll процедуру, которая в ней экспортировалась пятнадцатой, и при этом в рамках вашего приложения данной процедуре присваивается имя SayHello.

Если вы по каким-то причинам не применяете описанный выше способ импорта, но, тем не менее, хотите изменить имя импортируемой функции (процедуры), то можно воспользоваться третьим методом:

```
procedure CoolProcedure;
external myfirstdll.dll' name DoSomethingReallyCool';
```

Здесь импортируемой процедуре CoolProcedure дается имя DoSomethingReallyCool. Вызов процедур и функций, импортируемых из динамически загружаемых библиотек, несколько более сложен, чем рассмотренный нами выше способ. В данном случае требуется объявить указатель на функцию или процедуру, которую далее собираетесь использовать. Помните процедуру HelloWorld? Давайте посмотрим, что необходимо сделать для того, чтобы вызвать ее на выполнение в случае

динамической загрузки DLL. Во-первых, необходимо объявить тип, который описывал бы эту процедуру:

```
type
THelloWorld = procedure(AForm : TForm);
```

Теперь важно загрузить динамическую библиотеку и с помощью GetProcAddress получить указатель на процедуру, вызвать эту процедуру на выполнение и, наконец, выгрузить DLL из памяти. Ниже приведен код, демонстрирующий, как это можно сделать:

```
var
 DLLInstance : THandle;
 HelloWorld : THelloWorld;
begin
 { загружаем DLL }
 DLLInstance := LoadLibrary(myfirstdll.dll');
 { получаем указатель }
 @HelloWorld := GetProcAddress(DLLInstance,
HelloWorld');
 { вызываем процедуру на выполнение }
 HelloWorld(Self);
 { выгружаем DLL из оперативной памяти }
 FreeLibrary(DLLInstance);
end;
```

Как уже отмечалось, одним из недостатком статической загрузки DLL является невозможность продолжения работы приложения при отсутствии одной или нескольких библиотек. В случае с динамической загрузкой появляется возможность программно обрабатывать такие ситуации и не допускать, чтобы программа "вываливалась" самостоятельно. По возвращаемому функциями LoadLibrary и GetProcAddress значениям можно определить, успешно ли прошла загрузка библиотеки и найдена ли в ней необходимая приложению процедура. Приведенный ниже код демонстрирует это.

```
procedure TForm1.DynamicLoadBtnClick(Sender:
TObject);
type
 THelloWorld = procedure(AForm : TForm);
var
 DLLInstance : THandle;
 HelloWorld : THelloWorld;
begin
 DLLInstance := LoadLibrary(myfirstdll.dll');
 if DLLInstance = 0 then begin
```

```
    MessageDlg(Невозможно загрузить DLL', mtError,
[mbOK], 0);
    Exit;
    end;
    @HelloWorld := GetProcAddress(DLLInstance,
HelloWorld');
    if @HelloWorld <> nil then
    HelloWorld (Self)
    else
    MessageDlg(Не найдена искомая процедура!.', mtError,
[mbOK], 0);
    FreeLibrary(DLLInstance);
   end;
```

В DLL можно хранить не только код, но и формы. Причем создание и помещение форм в динамическую библиотеку не слишком сильно отличается от работы с формами в обычном проекте. Сначала рассмотрим, каким образом можно написать библиотеку, содержащую формы, а затем поговорим об использовании технологии MDI в DLL.

Разработку DLL, содержащую форму, рассмотрим на примере.

Создадим, прежде, новый проект динамической библиотеки. Для этого выберем пункт меню File|New, а затем дважды щелкнем на иконку DLL. После этого можно увидеть примерно следующий код:

```
library Project2;
{здесь были комментарии}

uses
 SysUtils,
 Classes;

{$R *.RES}

begin
end.
```

Сохраним полученный проект. Назовем его DllForms.dpr.

Теперь следует создать новую форму. Это можно сделать по-разному. Например, выбрав пункт меню File|New Form. Добавим на форму какие-нибудь компоненты. Назовем форму DllForm и сохраним получившийся модуль под именем DllFormUnit.pas.

Вернемся к главному модулю проекта и поместим в него функцию ShowForm, в задачу которой будет входить создание формы и ее вывод на экран. Используем для этого приведенный ниже код.

```
function ShowForm : Integer
```

Краткие выводы

DDE (Dynamic Data Exchange – DDE) – библиотека управления динамическим обменом данными была разработана для усиления возможностей первоначальной системы сообщений Windows.

Приложение, получающее данные из другого приложения по DDE и/или управляющее другим приложением с помощью команд через DDE, является DDE – клиентом.

DLL (Dynamic Link Library) библиотека, в отличие от приложения, не имеет ни стека, ни очереди сообщений. Функции, помещенные в DLL, выполняются в контексте вызвавшего приложения, пользуясь его стеком. Но эти же функции используют сегмент данных, принадлежащий библиотеке, а не копии приложения.

Ключевые слова

Реализация классов, задача (task), экземпляр (копия), приложение (instance), модуль (module), DLL- экспорт, DLL- импорт, сервер, клиент, DDE – сервер, DDE – клиент.

Вопросы для самоконтроля

1. Для чего используется DDE в Delphi?
2. Что подразумевается под понятием DDE – сервер?
3. Что подразумевается под понятием DDE – клиент?
4. В чем заключается процесс создания DLL в Delphi (экспорт)?
5. Какова суть использования DLL в Delphi (импорт)?

Рекомендуемая литература

1. Архангельский А. Программирование в Delphi 7. - М.: ООО «Бином – Пресс», 2004. -1152с.
2. 2.Брауде Э. Технология разработки программного обеспечения. Питер – 2004. - 325 с.
3. 3.Бобров И.С. Delphi 7. Учебный курс. Москва. Санкт-Петербург, Нижний Новгород, Воронеж, Питер.- 2003.- 625
4. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. Бином · 1998.
5. Джон Влиссидес, Эрих Гамма, Ричард Хелм, Ральф Джонсон. Приемы объектно-ориентированного проектирования. Паттерны проектирования. - Питер. 2003. - 256 с.
6. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. - М.: Изд. МГТУ имени Н.Э. Баумана. 2003.- 320с.

7. Кьоу Дж., Джеанини М.Объектно-ориентированное программирование. Просто и понятно. − М.: Питер, 2005. -403с.

8. Лесневский А.С. Объектно-ориентированное программирование для начинающих. - М.: Бином. Лаборатория знаний. 2005. - 382 с.

9. Синтес Антони. Освой самостоятельно объектно-ориентированное программирование за 21 день. Просто и понятно− М.: Вильямс. 2002. -284с.

10. http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».

11. http://.borland.com/delphi_net/ - Официальный сайт Borland Delphi

12. http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

Глава 5. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В DELPHI

5.1. Структурная обработка исключительных ситуаций

Структурная обработка исключительных ситуаций - это система, позволяющая программисту при возникновении ошибки (исключительной ситуации) связаться с кодом программы, подготовленным для обработки такой ошибки. Это выполняется с помощью языковых конструкций, которые как бы “охраняют” фрагмент кода программы и определяют обработчики ошибок, которые будут вызываться, если что-то пойдет не так на “охраняемом” участке кода. В данном случае понятие исключительной ситуации относится к языку и не нужно его путать с системными исключительными ситуациями (hardware exceptions), такими, как General Protection Fault. Эти исключительные ситуации обычно используют прерывания и особые состояния “железа” для обработки критичной системной ошибки; исключительные же ситуации в Delphi независимы от “железа”, не используют прерываний и употребляются для обработки ошибочных состояний, с которыми подпрограмма неготова иметь дело. Системные исключительные ситуации, конечно, могут быть перехвачены и преобразованы в языковые исключительные ситуации, но это только одно из применений языковых исключительных ситуаций.

При традиционной обработке, ошибки, обнаруженные в процедуре, как правило передаются наружу (в вызывавшую процедуру) в виде возвращаемого значения функции, параметров или глобальных переменных (флажков). Каждая вызывающая процедура должна проверять результат вызова на наличие ошибки и выполнять соответствующие действия. Часто, это просто выход еще выше, в более верхнюю вызывающую процедуру и т.д. : функция А вызывает В, В вызывает С, С обнаруживает ошибку и возвращает код ошибки в В, В проверяет возвращаемый код, видит, что возникла ошибка и возвращает код ошибки в А, А проверяет возвращаемый код и выдает сообщение об ошибке либо решает сделать что-нибудь еще, раз первая попытка не удалась.

Такая “пожарная бригада” для обработки ошибок трудоемка, требует написания большого количества кода, в котором можно легко ошибиться и который трудно отлаживать.

Одна ошибка в коде программы или переприсвоение в цепочке возвращаемых значений могут привести к тому, что нельзя будет связать

ошибочное состояние с положением дел во внешнем мире. Результатом будет ненормальное поведение программы, потеря данных или ресурсов, или крах системы.

Структурная обработка исключительной ситуации замещает ручную обработку ошибок автоматической, сгенерированной компилятором системой уведомления. В приведенном выше примере процедура A установила бы “охрану” со связанным обработчиком ошибки на фрагмент кода, в котором вызывается B. B просто вызывает C. Когда C обнаруживает ошибку, то создает (*raise*) исключительную ситуацию. Специальный код, сгенерированный компилятором и встроенный в Run-Time Library (RTL), начинает поиск обработчика данной исключительной ситуации. При поиске “защищенного” участка кода используется информация, сохраненная в стеке. В процедурах C и B нет такого участка, а в A - есть. Если один из обработчиков ошибок, которые используются в A, подходит по типу для возникшей в C исключительной ситуации, то программа переходит на его выполнение. При этом, область стека, используемая в B и C, очищается; выполнение этих процедур прекращается.

Если в A нет подходящего обработчика, то поиск продолжается в более верхнем уровне, и так может продолжаться, до тех порпока поиск не достигнет подходящего обработчика ошибок среди используемых по умолчанию обработчиков в RTL. Обработчики ошибок из RTL только показывают сообщение об ошибке и форсированно прекращают выполнение программы. Любая исключительная ситуация, которая осталась необработанной, приведет к прекращению выполнения приложения.

Без проверки возвращаемого кода после каждого вызова подпрограммы код программы должен быть более простым, а скомпилированный код - более быстрым. При наличии исключительных ситуаций подпрограмма B не должна содержать дополнительный код для проверки возвращаемого результата и передачи его в A. B ничего не должна делать для передачи исключительной ситуации, возникшей в C, в процедуру A - встроенная система обработки исключительных ситуаций делает всю работу.

Данная система называется *структурной,* поскольку обработка ошибок определяется областью “защищенного” кода; такие области могут быть вложенными. Выполнение программы не может перейти на произвольный участок кода; выполнение программы может перейти только на обработчик исключительной ситуации активной программы.

5.2. Модель исключительных ситуаций в Delphi

Модель исключительных ситуаций в Object Pascal является невозобновляемой(*non-resumable*). При возникновении исключительной ситуации Вы уже не сможете вернуться в точку, где она возникла, для продолжения выполнения программы (это позволяет сделать возобновляемая(*resumable*) модель). Невозобновляемые исключительные

ситуации разрушают стек, поскольку они сканируют его в поисках обработчика; в возобновляемой модели необходимо сохранять стек, состояние регистров процессора в точке возникновения ошибки и выполнять поиск обработчика и его выполнение в отдельном стеке. Возобновляемую систему обработки исключительных ситуаций гораздо труднее создать и применять, нежели невозобновляемую.

5.2.1. Синтаксис обработки исключительных ситуаций

Теперь, когда мы рассмотрели, что такое исключительные ситуации, давайте дадим ясную картину, как они применяются. Новое ключевое слово, добавленное в язык Object Pascal - **try**. Оно используется для обозначения первой части защищенного участка кода. Существуют два типа защищенных участков:

- *try..except*
- *try..finally*

Первый тип используется для обработки исключительных ситуаций. Его синтаксис:

```
try
 Statement 1;
 Statement 2;
 ...
except
 on Exception1 do Statement;
 on Exception2 do Statement;
 ...
else
 Statements; {default exception-handler}
end;
```

Для уверенности в том, что ресурсы, занятые вашим приложением, освободятся в любом случае, Вы можете использовать конструкцию второго типа. Код, расположенный в части *finally,* выполняется в любом случае, даже если возникает исключительная ситуация. Соответствующий синтаксис:

```
try
 Statement1;
 Statement2;

  . . .
finally
 Statements; { These statements always execute }
end;
```

5.2.2. Примеры обработки исключительных ситуаций

Ниже приведены процедуры A, B и C, обсуждавшиеся ранее, воплощенные в новом синтаксисе Object Pascal:

```
type
 ESampleError = class(Exception);
var
 ErrorCondition: Boolean;
procedure C;
begin
 writeln('Enter C');
 if (ErrorCondition) then
 begin
 writeln('Raising exception in C');
 raise ESampleError.Create('Error!');
 end;
 writeln('Exit C');
end;
procedure B;
begin
 writeln('enter B');
 C;
 writeln('exit B');
end;
procedure A;
begin
 writeln('Enter A');
 try
 writeln('Enter A"s try block');
 B;
 writeln('After B call');
 except
 on ESampleError do
 writeln('Inside A"s ESampleError handler');
 on ESomethingElse do
 writeln('Inside A"s ESomethingElse handler');
 end;
 writeln('Exit A');
end;
begin
 writeln('begin main');
 ErrorCondition := True;
 A;
 writeln('end main');
end.
```

При ErrorCondition = True программа выдаст:

```
begin main
Enter A
Enter A's try block
enter B
Enter C
Raising exception in C
Inside A's ESampleError handler
Exit A
end main
```

Возможно, вас удивила декларация типа 'ESampleError =class' вместо '=object'; это еще одно новое расширение языка. Delphi вводит новую модель объектов, доступную через декларацию типа '=class'.

Процедура C проверяет наличие ошибки (в нашем случае это значение глобальной переменной) и, если она есть (а это так), C вызывает(raise) исключительную ситуацию класса ESampleError.

Процедура A помещает часть кода в блок *try..except.* Первая часть этого блока содержит часть кода, аналогично конструкции *begin..end*. Эта часть кода завершается ключевым словом *except*, далее следуют один или более обработчиков исключительных ситуаций *on xxxx do yyyy*, далее может быть включен необязательный блок *else,* вся конструкция заканчивается *end;*. В конструкции, назначающей определенную обработку для конкретной исключительной ситуации (*on xxxx do yyyy*), после резервного слова *on* указывается класс исключительной ситуации, а после *do* следует собственно код обработки данной ошибки. Если возникшая исключительная ситуация подходит по типу к указанному после *on*, то выполнение программы переходит сюда (на код после *do*). Исключительная ситуация подходит в том случае, если она того же класса, что указана в *on,* либо является его потомком. Например, в случае *on EFileNotFound* обрабатываться будет ситуация, когда файл не найден. А в случае *on EFileIO* - все ошибки при работе с файлами, в том числе и предыдущая ситуация. В блоке *else* обрабатываются все ошибки, до этого необработанные.

Приведенные в примере процедуры содержат код (строка с writeln), который отображает путь выполнения программы. Когда C вызывает exception, программа сразу переходит на обработчик ошибок в процедуре A, игнорируя оставшуюся часть кода в процедурах B и C.

После того, как найден подходящий обработчик ошибки, поиск оканчивается. После выполнения кода обработчика программа продолжает выполняться с оператора, стоящего после слова *end* блока *try..except* (в примере - writeln('Exit A')).

Конструкция *try..except* подходит, если известно, какой тип ошибок нужно обрабатывать в конкретной ситуации. Но что делать, если требуется

выполнить некоторые действия в любом случае, произошла ошибка или нет? Это тот случай, когда понадобится конструкция *try..finally*.

Рассмотрим модифицированную процедуру B:

```
procedure NewB;
var
 P: Pointer;
begin
 writeln('enter B');
 GetMem(P, 1000);
 C;
 FreeMem(P, 1000);
 writeln('exit B');
end;
```

Если C вызывает исключительную ситуацию, то программа уже не возвращается в процедуру B. А что же с теми 1000 байтами памяти, захваченными в B? Строка FreeMem(P,1000) не выполнится и Вы потеряете кусок памяти. Как это исправить? Нужно ненавязчиво включить процедуру B в процесс, например:

```
procedure NewB;
var
 P: Pointer;
begin
 writeln('enter NewB');
 GetMem(P, 1000);
 try
 writeln('enter NewB''s try block');
 C;
 writeln('end of NewB''s try block');
finally
 writeln('inside NewB''s finally block');
 FreeMem(P, 1000);
 end;
 writeln('exit NewB');
end;
```

Если в A поместить вызов NewB вместо B, то программа выведет сообщения следующим образом:

```
begin main
Enter A
Enter A's try block
enter NewB
enter NewB's try block
Enter C
Raising exception in C
inside NewB's finally block
Inside A's ESampleError handler
Exit A
end main
```

Код в блоке *finally* выполнится при любой ошибке, возникшей в соответствующем блоке *try*. Он же выполнится и в том случае, если ошибки не возникло. В любом случае память будет освобождена. Если возникла ошибка, то сначала выполняется блок *finally,* затем начинается поиск подходящего обработчика. В штатной ситуации, после блока *finally* программа переходит на следующее после блока предложение.

Почему вызов GetMem не помещен внутрь блока *try*? Этот вызов может окончиться неудачно и вызвать exception EOutOfMemory. Если это произошло, то FreeMem попытается освободить память, которая не была распределена. Когда мы размещаем GetMem вне защищаемого участка, то предполагаем, что B сможет получить нужное количество памяти, а если нет, то более верхняя процедура получит уведомление EOutOfMemory.

А что, если требуется в B распределить четыре области памяти по схеме «все»-или-«ничего»? Если первые две попытки удались, а третья провалилась, то как освободить захваченную памятью область? Можно так:

```
procedure NewB;
var
 p,q,r,s: Pointer;
begin
 writeln('enter B');
 P := nil;
 Q := nil;
 R := nil;

S := nil;
 try
 writeln('enter B''s try block');
 GetMem(P, 1000);
 GetMem(Q, 1000);
 GetMem(R, 1000);
 GetMem(S, 1000);
 C;
 writeln('end of B''s try block');
 finally
```

```
  writeln('inside B''s finally block');
  if P <> nil then FreeMem(P, 1000);
  if Q <> nil then FreeMem(Q, 1000);
  if R <> nil then FreeMem(R, 1000);
  if S <> nil then FreeMem(S, 1000);
  end;
  writeln('exit B');
end;
```

Поначалу установив указатели в NIL, далее можно определить, успешно ли прошел вызов GetMem.

Оба типа конструкции *try* можно использовать в любом месте, допускается вложенность любой глубины. Исключительную ситуацию можно вызывать внутри обработчика ошибки, конструкцию *try* можно использовать внутри обработчика исключительной ситуации.

5.2.3. Вызов исключительной ситуации

Из примера в процедуре C мы уже могли видели, как должна поступать программа при обнаружении состояния ошибки - она вызывает исключительную ситуацию:

```
raise ESampleError.Create(‘Error!’);
```

После ключевого слова *raise* следует код, аналогичный тому, что используется для создания нового экземпляра класса. Действительно, в момент вызова исключительной ситуации создается экземпляр указанного класса; данный экземпляр существует до момента окончания обработки исключительной ситуации и затем автоматически уничтожается. Вся информация, которую нужно сообщить в обработчик ошибки, передается на объект через его конструктор в момент создания.

Почти все существующие классы исключительных ситуаций являются наследниками базового класса Exception и не содержат новых свойств или методов. Класс Exception имеет несколько конструкторов, какой из них конкретно использовать - зависит от задачи. Описание класса Exception можно найти в on-line Help.

5.3. Доступ к экземпляру объекта exception

До сих пор мы рассматривали механизмы защиты кода и ресурсов, логику работы программы в исключительной ситуации. Теперь нужно немного разобраться с тем, как же обрабатывать возникшую ошибку, точнее, как получить дополнительную информацию о коде ошибки, текст сообщения и т.п.

Как уже отмечалось, при вызове исключительной ситуации (raise) автоматически создается экземпляр соответствующего класса, который и

содержит информацию об ошибке. Весь вопрос в том, как в обработчике данной ситуации получить доступ к данному объекту.

Рассмотрим модифицированную процедуру A в нашем примере:

```
procedure NewA;
begin
 writeln('Enter A');
 try
 writeln('Enter A"s try block');
 B;
 writeln('After B call');
 except

 on E: ESampleError do writeln(E.Message);

 on ESomethingElse do
 writeln('Inside A"s ESomethingElse handler');
 end;
 writeln('Exit A');
end;
```

Здесь все изменения внесены в строку

```
 on ESE: ESampleError do writeln(ESE.Message);
```

Пример демонстрирует еще одно новшество в языке Object Pascal - создание локальной переменной. Таковой в нашем примере является ESE - это тот самый экземпляр класса ESampleError, который был создан в процедуре C в момент вызова исключительного состояния. Переменная ESE доступна только внутри блока *do.* Свойство Message объекта ESE содержит сообщение, которое было передано в конструктор Create в процедуре C.

Есть еще один способ доступа к экземпляру exception - использовать функцию *ExceptionObject:*

```
on ESampleError do
 writeln(ESampleError(ExceptionObject).Message);
```

5.4. Предопределенные обработчики исключительных ситуаций

Для профессионального программирования в Delphi необходимо иметь справочную информацию по предопределенным исключениям. Рассмотрим следующий перечень основных предопределенных исключений:

- **Exception** - базовый класс-предок всех обработчиков исключительных ситуаций.

- **EAbort** - “скрытое” исключение. Используйте его тогда, когда хотите прервать тот или иной процесс с условием, что пользователь программы не должен видеть сообщения об ошибке. Для повышения удобства использования в модуле *SysUtils* предусмотрена процедура *Abort*, определенная, как:

```
procedure Abort;
begin
 raise EAbort.CreateRes(SOperationAborted) at ReturnAddr;
end;
```

- **EComponentError -** вызывается в двух ситуациях:

1) при попытке регистрации компоненты за пределами процедуры Register;

2) когда имя компоненты не уникально или не допустимо.

- **EConvertError -** происходит в случае возникновения ошибки при выполнении функций *StrToInt* и *StrToFloat*, когда конвертация строки в соответствующий числовой тип невозможна.
- **EInOutError** - происходит при ошибках ввода/вывода при включенной директиве {$I+}.
- **EIntError** - предок исключений, случающихся при выполнении целочисленных операций.

ERangeError - вызывается при попытке обращения к элементам массива по индексу, выходящему за пределы массива, как результат RunTime Error 201 при включенной директиве {$R+}.

- **EInvalidCast** - происходит при попытке приведения переменных одного класса к другому, несовместимому с первым (например, приведение переменной типа TListBox к TMemo).
- **EInvalidGraphic** - вызывается при попытке передачи в *LoadFromFile* файла, несовместимого графического формата.
- **EInvalidGraphicOperation** - вызывается при попытке выполнения операций, неприменимых для данного графического формата (например, Resize для TIcon).
- **EInvalidObject -** реально нигде не используется, объявлен в *Controls.pas*.
- **EInvalidOperation** - вызывается при попытке отображения или обращения по Windows-обработчику (handle) контрольного элемента, не имеющего владельца (например, сразу после вызова MyControl:=TListBox.Create(...) происходит обращение к методу Refresh).
- **EInvalidPointer** - происходит при попытке освобождения уже освобожденного или еще неинициализированного указателя, при вызове Dispose(), FreeMem() или деструктора класса.
- **EListError -** вызывается при обращении к элементу наследника TList по индексу, выходящему за пределы допустимых значений (например, объект TStringList содержит только десять строк, а происходит обращение к одиннадцатому).
- **EMathError** - предок исключений, случающихся при выполнении операций с плавающей точкой.

- **EInvalidOp** - происходит, когда математическому сопроцессору передается ошибочная инструкция. Такое исключение не будет до конца обработано, пока Вы контролируете сопроцессор напрямую из ассемблерного кода.
- **EDivByZero** - вызывается в случае деления на ноль, как результат RunTime Error 200.
- **EIntOverflow** - вызывается при попытке выполнения операций, приводящих к переполнению целых переменных, как результат RunTime Error 215 при включенной директиве {$Q+}.
- **ERangeError** - вызывается при попытке обращения к элементам массива по индексу, выходящему за пределы массива, как результат RunTime Error 201 при включенной директиве {$R+}.
- **EInvalidCast** - происходит при попытке приведения переменных одного класса к другому классу, несовместимому с первым (например, приведение переменной типа TListBox к TMemo).
- **EInvalidGraphic** - вызывается при попытке передачи в *LoadFromFile* файла, несовместимого графического формата.
- **EInvalidGraphicOperation** - вызывается при попытке выполнения операций, неприменимых для данного графического формата (например, Resize для TIcon).
- **EInvalidObject** - реально нигде не используется, объявлен в *Controls.pas*.
- **EInvalidOperation** - вызывается при попытке отображения или обращения по Windows-обработчику (handle) контрольного элемента, не имеющего владельца (например, сразу после вызова MyControl:=TListBox.Create(...) происходит обращение к методу Refresh).
- **EInvalidPointer** - происходит при попытке освобождения уже освобожденного или еще неинициализированного указателя, при вызове Dispose(), FreeMem() или деструктора класса.
- **EListError** - вызывается при обращении к элементу наследника TList по индексу, выходящему за пределы допустимых значений (например, объект TStringList содержит только десять строк, а происходит обращение к одиннадцатому).
- **EMathError** - предок исключений, случающихся при выполнении операций с плавающей точкой.

EInvalidOp - происходит, когда математическому сопроцессору передается ошибочная инструкция. Такое исключение не будет до конца обработано, пока Вы контролируете сопроцессор напрямую из ассемблерного кода.

EOverflow - происходит как результат переполнения операций с плавающей точкой при слишком больших величинах. Соответствует RunTime Error 205.

- **Underflow** - происходит как результат переполнения операций с плавающей точкой при слишком малых величинах. Соответствует RunTime Error 206.
- **EZeroDivide** - вызывается в результате деления на ноль.

- **EMenuError** - вызывается в случае любых ошибок при работе с пунктами меню для компонент TMenu, TMenuItem, TPopupMenu и их наследников.
- **EOutlineError** - вызывается в случае любых ошибок при работе с TOutLine и любыми его наследниками.
- **EOutOfMemory** - происходит в случае вызовов New(), GetMem() или конструкторов классов при невозможности распределения памяти. Соответствует RunTime Error 203.
- **EOutOfResources** - происходит в том случае, когда невозможно выполнение запроса на выделение или заполнение тех или иных Windows ресурсов (например таких, как обработчики - handles).
- **EParserError** - вызывается когда Delphi не может произвести разбор и перевод текста описания формы в двоичный вид (часто происходит в случае исправления текста описания формы вручную в IDE Delphi).
- **EPrinter** - вызывается в случае любых ошибок при работе с принтером.
- **EProcessorException** - предок исключений, вызываемых в случае прерывания процессора- hardware breakpoint. Никогда не включается в DLL, может обрабатываться только в “цельном” приложении.
- **EBreakpoint** - вызывается в случае останова на точке прерывания при отладке в IDE Delphi. Среда Delphi обрабатывает это исключение самостоятельно.
- **EFault** - предок исключений, вызываемых в случае невозможности обработки процессором тех или иных операций.
- **EGPFault** - вызывается, когда происходит “общее нарушение защиты” - General Protection Fault. Соответствует RunTime Error 216.
- **EInvalidOpCode** - вызывается, когда процессор пытается выполнить недопустимые инструкции.
- **EPageFault** - обычно происходит как результат ошибки менеджера памяти Windows, вследствие некоторых ошибок в коде Вашего приложения. После такого исключения рекомендуется перезапустить Windows.
- **EStackFault** - происходит при ошибках работы со стеком, часто вследствие некорректных попыток доступа к стеку из фрагментов кода на ассемблере. Компиляция Ваших программ со включенной проверкой работы со стеком {$S+} помогает отследить такого рода ошибки.
- **ESingleStep** - аналогично EBreakpoint, это исключение происходит при пошаговом выполнении приложения в IDE Delphi, которая сама его и обрабатывает.
- **EPropertyError** - вызывается в случае ошибок в редакторах свойств, встраиваемых в IDE Delphi. Имеет большое значение для написания надежных property editors. Определен в модуле DsgnIntf.pas.
- **EResNotFound** - происходит в случае тех или иных проблем, имеющих место при попытке загрузки ресурсов форм - файлов .DFM в режиме дизайнера. Часто причиной таких исключений бывает нарушение соответствия между определением класса формы и ее описанием на уровне ресурса

(например,вследствие изменения порядка следования полей-ссылок на компоненты, вставленные в форму в режиме дизайнера).

- **EStreamError** - предок исключений, вызываемых при работе с потоками.
- **EFCreateError** - происходит в случае ошибок создания потока (например, при некорректном задании файла потока).
- **EFilerError** - вызывается при попытке вторичной регистрации уже зарегистрированного класса (компоненты). Является, также, предком специализированных обработчиков исключений, возникающих при работе с классами компонент.
- **EClassNotFound** - обычно происходит, когда в описании класса формы удалено поле-ссылка на компоненту, вставленную в форму в режиме дизайнера. Вызывается, в отличие от EResNotFound, в RunTime
- **EInvalidImage** - вызывается при попытке чтения файла, не являющегося ресурсом, или разрушенного файла ресурса, специализированными функциями чтения ресурсов (например, функцией ReadComponent).
- **EMethodNotFound** - аналогично EClassNotFound, только при несоответствии методов, связанных с теми или иными обработчиками событий.
- **EReadError** - происходит в том случае, когда невозможно прочитать значение свойства или другого набора байт из потока (в том числе ресурса).
- **EFOpenError** - вызывается когда тот или иной специфированный поток не может быть открыт (например, когда поток не существует).
- **EStringListError** - происходит при ошибках работы с объектом TStringList (кроме ошибок, обрабатываемых TlistError).

Краткие выводы

К средствам обработки исключений относятся: специальные конструкции языка для разделения операторов основной части программы и операторов обработки исключений; иерархия классов различных исключений, определенная в Delphi; операторы генерации и обработки исключений.

Ключевые слова

*И*сключительная ситуация, структурная обработка исключительных ситуаций, модель исключительной ситуации, вызов исключительной ситуации, объект exception, предопределенные обработчики исключительных ситуаций.

Вопросы для самоконтроля

1. Какие ситуации попадают под понятие «исключительные»? Почему возникла необходимость создания средств обработки исключений?
2. Что такое структурная обработка исключительных ситуаций в Delphi?
3. Какова модель исключительных ситуаций в Delphi?
4. Как происходит вызов исключительной ситуации в Delphi?

5. Что подразумевается под понятием «предопределенные обработчики исключительных ситуаций»?

Рекомендуемая литература

1. Ален Э. Типичные ошибки проектирования. Библиотека программиста. Питер. - 2004. - 284 .

2.Брауде Э. Технология разработки программного обеспечения. Питер – 2004. - 325 с.

3.Бобров И.С. Delphi 7. Учебный курс. Москва. Санкт-Петербург, Нижний Новгород, Воронеж, Питер.- 2003.- 625

4.Гради Буч. Объектно – ориентированный анализ и проектирование с примерами приложений на C++. Бином · 1998

5. Джон Влиссидес, Эрих Гамма, Ричард Хелм, Ральф Джонсон. Приемы объектно – ориентированного проектирования. Паттерны проектирования. - Питер. 2003. - 256 с.

6. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно – ориентированное программирование. - М. Издательство МГТУ имени Н.Э. Баумана. 2003.- 320с.

7.Кьоу Дж.,Джеанини М.Объектно-ориентированное программирование. Просто и понятно.- Питер. 2005. -403с.

8. Лесневский А.С. Объектно – ориентированное программирование для начинающих. - М. Бином. Лаборатория знаний. 2005. - 382 с.

9.Майер Б. Объектно – ориентированное программирование. Концепции разработки. М. Русская редакция.2004.- 456 с.

10.Синтес Антони. Освой самостоятельно объектно – ориентированное программирование за 21 день. Вильямс · 2002. -284с.

11. Шпак Ю.А. "Delphi 7 на примерах" – М. Русская редакция 2005. - 207с.

12.http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».

13.http://.borland.com/delphi_net/ - Официальный сайт Borland Delphi

14.http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

Заключение

Объектно-ориентированное программирование представляет собой такой метод программирования, который весьма близко напоминает наше поведение и является естественной эволюцией более ранних нововведений в разработке языков программирования. Объектно-ориентированное программирование более структурное, чем все предыдущие разработки, касающиеся структурного программирования. Оно также является более модульным и абстрактным, чем предыдущие попытки абстрагирования данных и переноса деталей программирования на внутренний уровень.

Цель создания данного учебного пособия - максимально просто и одновременно подробно изложить современное представление как о самой технологии ООП, так и о средствах ее реализации с помощью Delphi, чтобы каждый, ознакомившийся с данным изданием мог использовать многочисленные возможности ООП.

Объектный подход ныне является наиболее распространенным. Основанная на компонентах, архитектура Delphi была бы невозможной, если бы не было объектов. Компоненты Delphi, это по существу, просто специализированными объектами, и их функционирование определяется объектно-ориентированной архитектурой библиотеки визуальных компонент.

Безусловно, что в рамках одного учебного пособия практически невозможно предусмотреть все ситуации и обсудить тонкости разработки программ с использованием ООП. Помимо этого, для практического освоения данной технологией необходимо разработать несколько своих оригинальных программ, постепенно изучая приемы ООП. Для уточнения более сложных и детальных вопросов на сегодняшний день существует ряд весьма интересных изданий по использованию языка Delphi, других объектно-ориентированных языков программирования, как классических, так и современных авторов, к которым следует обратится при составлении оригинальных программ.

Данное учебное пособие так же ставит своей целью – быть помощником студентов при, их самостоятельной подготовке к занятию или экзамену. Так последовательное изложение и упрощенное построение глав способствуют быстрому пониманию и усвоению данной информации, что дает общее представление о программировании в этойсреде.

Надеемся, что ознакомившись с данной разработкой, читатель сумел получить достаточно целостное представление о возможностях ООП и областях его применения.

ГЛОССАРИЙ

Абстрактная операция, abstract operation. Объявленная, но не реализованная операция в абстрактном классе.

Абстрактный класс, abstract class. Класс, который не может иметь экземпляров. Абстрактный класс пишется в предположении, что его конкретные подклассы дополнят его структуру и поведение, скорее всего, реализовав абстрактные операции.

Абстракция, abstraction. Существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют его концептуальные границы для наблюдателя. Абстрагирование - процесс выявления абстракций. Один из основных элементов объектной модели.

Агент, agent. Объект, который подвергается воздействию со стороны и сам воздействует на другие объекты. Обычно агенты создаются для выполнения некоторой работы по поручению актеров или других агентов.

Актер, actor. Объект, воздействующий на другие объекты, но сам не подвергающийся воздействию с их стороны. В некоторых контекстах то же самое, что активный объект.

Активный объект, active object. Объект, которому выделен свой поток управления.

Алгоритмическая декомпозиция, algorithmic decomposition. Процесс разделения системы на части, каждая из которых отражает этап общего процесса. Применение структурного подхода к проектированию приводит к алгоритмической декомпозиции, которая фокусируется на потоке управления в системе

Архитектура модулей, module architecture. Граф, вершины которого соответствуют модулям, а ребра - отношениям модулей между собой. Архитектура модулей системы представляется совокупностью диаграмм модулей.

Архитектура процессов, process architecture. Граф, вершины которого соответствуют процессорам и устройствам, а ребра - соединениям между ними. Для описания архитектуры процессов системы используются диаграммы процессов.

Архитектура, architecture. Логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями.

Ассоциация, association. Отношение, означающее некоторую смысловую связь между классами.

Атрибут, attribute. Часть составного объекта (агрегата).

Базовый класс, base class. Наиболее общий класс в какой-либо структуре классов. В большинстве приложений есть несколько таких корневых классов. В некоторых языках программирования определяется всеобщий базовый класс, который является суперклассом для всех остальных классов.

Блокирующий объект, blocking object. Пассивный объект, способный работать в многопоточном окружении. Вызов операции блокирующего объекта блокирует клиента на все время операции.

Видимость, visibility. Способность одной абстракции видеть другую и, таким образом, ссылаться на ее ресурсы извне. Абстракции видимы друг другу, только если они находятся в одном пространстве имен. Контроль экспорта может еще более ограничить доступ к видимым абстракциям.

Виртуальная функция, virtual function. Какая-либо операция над объектом. Виртуальная функция может быть переопределена в подклассах, следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины "обобщенная функция" и "виртуальная функция" взаимозаменяемы.

Временная сложность, time complexity. Относительное или абсолютное время, за которое выполняется операция.

Действие, action. Некое происшествие в системе, требующее, с практической точки зрения, нулевого времени для своего завершения. Действием может быть вызов операции, запуск другого события, начало или остановка деятельности.

Делегирование, delegation. При делегировании один объект, ответственный за операцию, передает выполнение этой операции другому объекту.

Деструктор, destructor. Операция класса, которая освобождает состояние объекта и/или уничтожает сам объект.

Деятельность, activity. Операция, выполнение которой требует некоторого времени.

Диаграмма взаимодействий, interaction diagram. Часть системы обозначении объектно-ориентированного проектирования; используется для демонстрации выполнения какого-либо сценария в контексте диаграммы объектов.

Диаграмма классов, class diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать классы и их взаимоотношения в логическом проекте системы. Может представлять всю структуру классов или ее часть.

Диаграмма модулей, module diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации

разбиения классов и объектов по модулям в физическом проекте системы. Диаграмма модулей отображает архитектуру модулей системы.

Диаграмма объектов, object diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать объекты и отношения между ними в логическом проекте системы. Может отражать всю объектную структуру или часть ее; обычно иллюстрирует смысл механизмов в логическом проекте. Отдельная диаграмма объектов - моментальный снимок из жизни системы.

Диаграмма переходов и состояний, state transition diagram. Часть обозначений объектно-ориентированного проектирования; используется для отображения пространства состояний данного класса, событий, которые вызывают переход из одного состояния в другое, и действий, возникающих в результате смены состояния.

Диаграмма процессов, process diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать, как процессы размешены по процессорам в физическом проекте системы. Диаграмма процессов отражает архитектуру процессов.

Динамическое связывание, dynamic binding. Связывание означает установление соответствия имени (например, объявленной переменной) с классом. Динамическое связывание происходит при выполнении программы в тот момент, когда создается объект, обозначенный именем.

Друг, friend. Класс или операция, имеющие доступ к закрытым операциям или данным некоторого класса. Только сам класс может называть своих друзей.

Закрытая часть, private. Часть интерфейса какого-либо класса, объекта или модуля, закрытая (невидимая) для других классов, объектов и модулей.

Защищенная часть, protected. Часть интерфейса какого-либо класса, объекта или модуля, невидимая для всех других классов, объектов и модулей за исключением подклассов.

Идентичность, identity. Природа объекта; то, что отличает его от других объектов.

Идиома, idiom. Выражение, общепринятое в каком-либо языке программирования или культуре какого-либо приложения, отражающее общепринятый способ использования данного языка.

Иерархия, hierarchy. Подчинение или упорядочение абстракций. Две типичных иерархии в сложной системе - структура классов

(включая иерархию "обшее/частное") и структура объектов (включая иерархию "целое/часть"); иерархии можно также обнаружить в архитектурах модулей и процессов.

Инвариант, invariant. Логическое выражение некоторого условия, истинность которого необходимо соблюдать.

Инкапсуляция, encapsulation. Процесс разделения элементов абстракции, которые образуют ее структуру и поведение. Служит для отделения внешних обязательств объекта от его реализации.

Инстанцирование, instantiation. Подстановка параметров шаблона обобщенного или параметризованного класса; в результате создается конкретный класс, который может иметь экземпляры.

Интерфейс, interface. Внешний вид класса, объекта или модуля, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведения.

Исключение, exception. Возбуждение исключения показывает, что некоторый логический инвариант не соблюдается. В c++ мы возбуждаем исключение, чтобы избежать неправомочное исполнение операций и дать знать о возникшей проблеме другим объектам, которые могут перехватить исключение и принять меры.

Использовать, use. Ссылаться на абстракцию извне.

Итератор, iterator. Операция, позволяющая навещать части некоторого объекта.

Категория классов, class category. Логически полный набор классов, одни из которых видимы для других категорий классов, а другие - нет. Классы в категории сотрудничают для предоставления некоторого набора услуг.

Класс, class. Множество объектов с общей структурой и поведением. Термины "класс" и "тип" в большинстве случаев (но не всегда) взаимозаменяемы. Понятие класса отличается от понятия типа тем, что концентрируется на классификации по структуре и поведению.

Класс-контейнер, container class. Класс, экземпляры которого представляют собой коллекции других объектов. Контейнер может быть однородным (коллекции включают экземпляры только одного класса) либо неоднородным (коллекции включают экземпляры разных классов, имеющих обычно общий суперкласс). В c++ контейнеры обычно определяются как параметризованные классы с параметром, обозначающим класс объектов коллекции.

Клиент, client. Объект, который пользуется услугами другого объекта либо выполняя операции над последним, либо через доступ к его состоянию.

Ключ, key. Атрибут, значение которого однозначно идентифицирует объект.

Ключевая абстракция, key abstraction. Класс или объект, являющийся частью словаря предметной области.

Конкретный класс, concrete class. Класс, реализация которого завершена и который, поэтому, может иметь экземпляры.

Конструктор, constructor. Операция, создающая объект и/или инициализирующая его состояние.

Метакласс, metaclass. Класс класса; класс, экземпляры которого сами являются классами.

Метод, method. Операция над объектом, определенная как часть описания класса. Не любая операция является методом, но все методы - операции. Термины "метод", "сообщение" и "операция" обычно взаимозаменяемы. В некоторых языках методы существуют сами по себе и могут переопределяться подклассами; в других языках метод не может быть переопределен, - он служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

Механизм, mechanism. Структура, посредством которой объекты сотрудничают друг с другом, осуществляя поведение, которое соответствует требованиям системы.

Модификатор, modifier. Операция, изменяющая состояние объекта.

Модуль, module. Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в соответствии с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.

Модульность, modularity. Свойство системы, которая была разделена на связные и слабо зацепленные между собой модули.

Мономорфизм, monomorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать только объекты одного и того же класса.

Мощность, cardinality. Число экземпляров класса: число экземпляров, участвующих в связи классов.

Наследование, inheritance. Отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное" в которой подкласс наследует от одного или нескольких более общих

суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

Обобщенная функция, generic function. Какая-либо операция над объектом. Обобщенная функция класса может быть переопределена в подклассах; следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины "обобщенная функция" и "виртуальная функция" взаимозаменяемы.

Обобщенный класс, generic class. Класс, служащий шаблоном для создания других классов: шаблон параметризуется другими классами, объектами и/или операциями. Обобщенный класс до создания объектов должен быть инстанцирован. Обобщенные классы используются как контейнерные классы. Термины "обобщенный класс" и "параметризованный класс" взаимозаменяемы.

Обратный инжиниринг, reverse-engineering. Восстановление логической или физической модели системы по коду. Противопоставляется прямому инжинирингу.

Объект, object. Нечто, чем можно оперировать. Объект имеет состояние, поведение и идентичность. Структура и поведение сходных объектов определены в общем для них классе. Термины "экземпляр" и "объект" взаимозаменяемы.

Объектная модель, object model. Совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования; парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости.

Объектное программирование, object-based programming. Метод программирования, основанный на представлении программы как совокупности объектов, каждый из которых является экземпляром некоторого типа. Типы образуют иерархию, но не наследственную. В таких программах типы рассматриваются как статические, а объекты имеют более динамическую природу, которую ограничивают статическое связывание и мономорфизм.

Объектно-ориентированная декомпозиция, object-oriented decomposition. Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования приводит к объектно-ориентированной декомпозиции, при которой мы рассматриваем мир как совокупность объектов, согласованно действующих для обеспечения требуемого поведения.

Объектно-ориентированное программирование, object-oriented programming (oop). Методология реализации, при которой

программа организуется, как совокупность сотрудничающих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты весьма динамичны, что поощряется динамическим связыванием и полиморфизмом.

Объектно-ориентированное проектирование, object-oriented design (ood). Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов.

Объектно-ориентированный анализ, object-oriented analysis. Метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области.

Объект-член, member object. Часть состояния объекта. В совокупности объекты-члены полностью определяют структуру объекта. Термины "переменная экземпляра", "поле". "объект-член" и "слот" взаимозаменяемы.

Ограничение, constraint. Выражение некоторого смыслового условия, которое должно выполняться.

Операция класса, class operation. Операция, например, конструктор или деструктор, общая для всего класса и не принадлежащая конкретному объекту.

Операция, operation. Нечто, проделываемое одним объектом над другим, чтобы вызвать реакцию. Все операции, которые можно выполнить над каким-либо объектом, сосредоточены в свободных подпрограммах и функциях-членах (методах). Термины "операция", "метод" и "сообщение" взаимозаменяемы.

Ответственность, responsibility. Поведение, за которое ответственен объект.

Открытая часть, public. Часть интерфейса какого-либо класса, объекта или модуля, открытая (видимая) для всех классов, объектов и модулей.

Параллелизм, concurrency. Свойство, отличающее активные объекты от неактивных.

Параллельный объект, concurrent object. Активный объект, способный работать в многопоточной среде.

Параметризованный класс, parameterized class. Класс, служащий шаблоном для других классов; шаблон параметризуется другими классами, объектами и/или операциями. Параметризованный класс должен быть инстацирован до создания объектов. Параметризованные

классы используются как контейнеры. Термины "обобщенный класс" и "параметризованный класс" взаимозаменяемы.

Пассивный объект, passive object. Объект, не имеющий собственного потока управления.

Переменная класса, class variable. Часть состояния класса. Совокупность всех переменных класса образует его структуру. Переменные класса совместно используются всеми его экземплярами. В c++ переменная класса объявляется как статический член.

Переменная экземпляра, instance variable. Часть состояния объекта. В совокупности переменные экземпляра полностью определяют структуру объекта. Термины "переменная экземпляра", "поле", "объект-член" и "слот" взаимозаменяемы.

Переход, transition. Переход из одного состояния в другое.

Поведение, behavior. Действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

Подкласс, subclass. Класс, наследующий от одного или нескольких классов (которые называются его непосредственными суперклассами).

Подсистема, subsystem. Совокупность модулей, часть которых видима для других подсистем, а часть - скрыта.

Поле, field. Часть состояния объекта; совокупность полей объекта образуют его структуру. Термины "поле", "переменная экземпляра", "объект-член" и "слот" означают одно и то же.

Полиморфизм, polymorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

Последовательное проектирование, round-trip gestalt design. Стиль проектирования, который подчеркивает последовательность и итеративность в развитии системы: посредством уточнения различных, хотя и согласованных логических и физических представлений системы в целом; объектно-ориентированное проектирование основывается на последовательном проектировании, что является выражением взаимозависимости общей картины проекта и его деталей.

Последовательный объект, sequential object. Пассивный объект, рассчитанный на работу в однопоточном окружении.

Постусловие, postcondition. Инвариант, соблюдаемый на выходе из операции.

Поток управления, thread of control. Отдельный процесс. Запуск потока управления приводит к возникновению независимой динамической деятельности в системе; данная система может иметь несколько одновременно выполняемых потоков, некоторые из них могут динамически возникать и уничтожаться. Многопроцессорные системы допускают истинную многопоточность в то время как на однопроцессорных компьютерах возможна только иллюзия многопоточности. (термин "thread of control" переводится также как "нить управления". В данном издании принят перевод "поток управления" как более распространенный. Отметим, что в некоторых случаях автор использует термин "flow of control", который переведен также. - *примеч. Ред.)*

Предусловие, precondition. Инвариант, предполагаемый на входе в операцию.

Примесь, mixin. Класс, реализующий какое-либо четко выделенное поведение; используется для уточнения поведения других классов посредством наследования; поведение примеси обычно ортогонально поведению класса, с которым она смешивается.

Пространственная сложность, space complexity. Относительный или абсолютный объем памяти, занимаемый объектом.

Пространство состояний, state space. Перечислимое множество всех возможных состояний объекта. Пространство состояний программы содержит неопределенное, но конечное число состояний (не обязательно желаемых или ожидаемых).

Протокол, protocol. Способы, которыми объекты могут действовать и реагировать; полное статическое и динамическое представление объекта; протокол объекта определяет допустимое поведение объекта.

Процесс, process. Запуск одного потока управления.

Процессор, processor. Часть аппаратного обеспечения, имеющая вычислительные ресурсы.

Прямой инжиниринг, forward-engineering. Создание исполнимого кода по логической или физической модели. Противопоставляется обратному инжинирингу.

Раздел, partition. Категории классов или подсистемы, составляющие часть данного уровня абстракции.

Реактивная система, reactive system. Система, движимая событиями. Поведение такой системы не определяется простым отображением "вход-выход".

Реализация, implementation. Внутреннее представление класса, объекта или модуля, включая секреты его поведения.

Роль, role. Способность или цель, с которой класс или объект участвуют в отношениях с другими; некоторая четко выделяемая черта поведения объекта в определенный момент времени; роль - это лицо, которое объект являет миру в данный момент.

Свободная подпрограмма, free subprogram. Процедура или функция, которая выполняется как непримитивная операция над объектом или объектами одного и тоже или различных классов. Свободная подпрограмма - это любая подпрограмма, которая не является методом какого-либо класса.

Связь, link. Связь между объектами, экземпляр ассоциации.

Селектор, selector. Операция, имеющая доступ к состоянию объекта, но не изменяющая его.

Сервер, server. Объект, который никогда не воздействует на другие объекты, но используется ими; объект, предоставляющий некоторые услуги.

Сигнатура, signature. Полная спецификация операции с указанием типов аргументов и возвращаемого значения.

Сильно типизированный, strongly typed. Свойство языка программирования, в соответствии с которым во всех выражениях гарантируется согласованность типов.

Синхронизация, synchronization. Семантика параллельности операции. Операция может быть простой (присутствует только один поток управления); синхронной (рандеву двух потоков); односторонняя (рандеву, при котором одному из потоков приходится ждать);по истечении времени (рандеву, в котором один процесс ждет другого определенное время); асинхронной (два процесса независимы друг от друга).

Система реального времени, real-time system. Система, в которой некоторые существенные процессы должны укладываться в отведенное время. Система "жесткого" реального времени должна быть детерминированной; запаздывание с реакцией грозит катастрофой.

Скрытие информации, information hiding. Процесс скрытия всех секретов объекта, которые ничего не добавляют к его существенным характеристикам; обычно скрывают структуру объекта и реализацию его методов.

Словарь данных, data dictionary. Полный перечень всех классов в системе.

Слой, layer. Совокупность категорий классов или подсистем одного уровня абстракции.

Слот, slot. Часть состояния объекта; совокупность слотов образует структуру объекта. Термины "поле", "переменная экземпляра", "объект-член" и "слот" означают одно и то же.

Событие, event. Что-то, что может изменить состояние системы.

Сообщение, message. Операция, которую один объект может выполнять над другим. Термины "сообщение", "метод" и "операция" обычно взаимозаменяемы.

Составной объект (агрегат), aggregate object. Объект, состоящий из других объектов (его частей).

Состояние, state. Совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя перечень (обычно, статический) свойств объекта и текущие значения (обычно, динамические) этих свойств.

Сотрудничество, collaboration. Процесс, в котором несколько объектов сотрудничают для обеспечения требуемого поведения верхнего уровня.

Сохраняемость, persistence. Способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из одного адресного пространства в другое.

Среда разработки, framework. Набор классов, предоставляющих некоторые базовые услуги в определенной области. Таким образом, среда разработки экспортирует классы и механизмы, которые клиенты могут использовать или адаптировать.

Статическое связывание, static binding. Связывание означает установление соответствия имени (например, объявленной переменной) классу. Статическое связывание происходит при объявлении имени (во время компиляции), до того, как объект будет создан.

Страж, guard. Логическое выражение, применяемое к событию; если выражение истинно, то событие происходит и система изменяет состояние.

Стратегическое проектное решение, strategic design decision. Проектные решения, которые имеют решающее влияние на архитектуру.

Структура классов, class structure. Граф, вершины которого соответствуют классам, а ребра - отношениям классов. Структура классов для конкретной системы представляется в виде совокупности диаграмм классов.

Структура объектов, object structure. Граф, вершины которого соответствуют объектам, а ребра - отношениям объектов. Для отражения структуры объектов или ее части используются диаграммы объектов.

Структура, structure. Конкретное представление состояния объекта. Каждый объект имеет собственное состояние, независимое от других объектов, хотя все объекты одного класса имеют одинаковое представление состояния.

Структурное проектирование, structured design. Метод проектирования, основанный на алгоритмической декомпозиции.

Суперкласс, superclass. Класс, которому наследуют другие классы (называемые непосредственными подклассами).

Сценарий, scenario. Последовательность событий, выражающая некий аспект поведения системы.

Тактическое проектное решение, tactical design decision. Проектное решение, имеющее ограниченное значение для архитектуры.

Тип, type. Определение области допустимых значений, которые может принимать объект, и множества операций, которые могут выполняться над объектом. Термины "класс" и "тип" обычно (но не всегда) взаимозаменяемы; тип отличается от класса тем, что фокусируется на поддержке общего протокола.

Типизация, typing. Механизмы, препятствующие замене объектов одного типа на другой или, в крайнем случае, жестко ограничивающие такую замену.

Трансформационная система, transformational system. Система, поведение которой определяется в терминах отображения "вход-выход".

Управление доступом, access control. Механизм доступа к данным и операциям класса. В c++ открытые элементы доступны всем, защищенные элементы доступны подклассам, так называемым друзьям класса и файлам реализации, закрытые элементы доступны реализации и друзьям класса. Наконец, элементы с доступом на уровне реализации доступны только в файле реализации класса.

Уровень абстракции, level of abstraction. Относительное упорядочение абстракций по структурам классов, объектов, модулей или процессов. В терминах иерархии "часть/целое" объект находится на более высоком уровне абстракции, чем другие, если он строится на основе этих объектов: в терминах иерархии "общее/частное", высокоуровневые абстракции носят более обобщенный характер, чем низкоуровневые.

Услуга, service. Поведение, обеспечиваемое некоторой частью системы.

Устройство, device. Часть аппаратуры, не имеющая собственных вычислительных ресурсов.

Утверждение, assertion. Логическое выражение некоторого условия, истинность которого необходимо обеспечить.

Утилита класса, class utility. Совокупность свободных подпрограмм. На c++ - класс, который состоит только из статических членов и/или функций-членов.

Функциональная точка, function point. В контексте анализа требований к системе - отдельное поведение, видимое извне и поддающееся проверке.

Функция, function. Некоторое преобразование "вход-выход", вытекающее из поведения объекта.

Функция-член, member function. Операция над объектом, определенная как часть описания класса. Все функции-члены - операции, но не все операции - функции-члены. Термины "функции-члены" и "методы" взаимозаменяемы. В некоторых языках функции-члены существуют сами по себе и могут переопределяться подклассами; в других языках функция-член не может быть переопределена, - она служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

Экземпляр, instance. Нечто, чем можно оперировать. Экземпляр имеет состояние, поведение и идентичность. Структура и поведение всех экземпляров класса определяются этим классом. Термины "объект" и "экземпляр" взаимозаменяемы.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Законы Республики Узбекистан

1.Конституция Республики Узбекистан, − Т.: Узбекистан, 2003г., 48с.

2. Закон Республики Узбекистан «Об информатизации». //Народное слово, 2004 г., 11 февраля.

3.Закон Республики Узбекистан «Об электронном документообороте», //Народное слово, 2004 г., 30 апреля

4. Закон Республики Узбекистан «Об электронной коммерции», //Народное слово, 2004 г., 30 апреля

2. Указы Президента Республики Узбекистан

5.Указ Президента Республики Узбекистан от 14 июня 2005 г. «О мерах по ускорению реализации приоритетных направлений в сфере углубления рыночных реформ и дальнейшей либерализации экономики». //Народное слово. 2005 г.

6.Указ Президента Республики Узбекистан « О мерах по дальнейшему совершенствованию системы правовой защиты субъектов предпринимательства». //Народное слово. 2005 год.

7.Указ Президента Республики Узбекистан от 20 июня 2005 г. N УП-3620 «О дополнительных мерах по стимулированию развития микрофирм и малых предприятий» //Народное слово, 2005 г.

8.Указ президента Республики Узбекистан « О дальнейшем развитии компьютеризации и внедрении информационно – коммуникационных технологий», //Народное слово, 2002 г.

3.Постановления Кабинета Министров Республики Узбекистан

9.Постановление Кабинета Министров Республики Узбекистан «О дополнительных мерах по дальнейшему развитию информационно – коммуникационных технологий», 2005 г.

10.Постановление Кабинета Министров Республики Узбекистан «О совершенствовании системы подготовки кадров в сфере информационных технологий» от 02.06.2005 год, // Народное слово. 2005 г.

11.Постановление Кабинета Министров Республики Узбекистан «О дальнейших мерах развития компьютеризации и внедрения информационно – коммуникационных технологий», //Народное слово, 2002 г.

4. Труды Президента Республики Узбекистан.

12. Каримов И.А. Наша главная цель – демократизация и обновление общества, реформирование и модернизация страны: Доклад на совместном заседании Законодательной палаты и Сената Олий Мажлиса Республики Узбекистан 28 января 2005 г. – Т: Узбекистан, 2005 г, 49с.

13.3. Каримов И.А. Узбекский народ никогда и ни от кого не будет зависеть. Избр.соч. Т13. − Т.: Узбекистан 2005-264с.

14.Каримов И.А. Избранный нами путь – это путь демократического развития и сотрудничества с прогрессивным миром. Избр. соч.т.11. − Т.: Узбекистан, 2003. 296стр.

15.Каримов И.А.За безопасность и мир надо бороться. Избр.соч. т 10. − Т.: Узбекистан,2002

5. Документы министерств и ведомств

16. Положение о порядке создания и использования электронных баз данных в государственных организациях республики. Утверждено Государственным комитетом Республики Узбекистан по науке и технике, от 29.20.1995.

17. Положение о порядке и правилах создания, внедрения и эксплуатации локальных, ведомственных, региональных и других информационно-вычислительных сетей на территории республики. Утверждено Государственным комитетом Республики Узбекистан по науке и технике, от 30.01.1995.

6. Учебники

18. Гулямов С.С., Алимов Р.Х., Лутфуллаев Х.С. Информационные системы и технологии. − Т.: Шарк, 2000 г. - 592 с.

19. Гулямов С.С., Шермухамедов А. Т., Бегалов Б.А. «Экономическая информатика». − Т.: Узбекистон, 1999. - 528 б.

20. Ален Э. Типичные ошибки проектирования. Библиотека программиста. Питер – 2004 г.

21.Арчер Т. Основы C#. Новейшие технологии. Перевод с английского. − М.: Издательско – торговый дом «Русская редакция», 2004 г.

22.Архангелский А. Программирование в Delphi 7. - М.: ООО «Бином – Пресс», 2004. -1152с.

23.Голицина О.Л., Партыка Т.Л., Попов И.И. Информационные технологии. – М.: Финансы и статистика, 2005, 203с.

24. Бандурка А.М., Бочарова С.П., Землянская Е.В. «Психология управления» Харьков: ООО «Фортуна-пресс», 2004г.480 с.

25. Бобров И.С. «DELPHI 7». Учебный курс. Москва. Санкт – Петербург, Нижний Новгород, Воронеж. Питер – 2003 г.

26. Вингерс К. Разработка требования к программному обеспечению. «Русская редакция» - 2004 г.

27. Гради Буч Объектно - ориентированный анализ и проектирование с примерами приложений на C++ Бином. - 1998 г., 560 с.

28. Гамма Э., Влиссидес, Джонсон Р., Хелм Р. Приемы объектно – ориентированного проектирования. Библиотека программиста. 2004 г.

29. Иан Грэхем Объектно-ориентированные методы. Принципы и практика. Изд.3, Издательство Диалектика, 2005 г.

30.Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. Издательство МГТУ им. Н.Э. Баумана, 2003 г.

31.«Информатика»/ Под редакцией Н.В. Макаровой, − М.: Финансы и статистика, 2006 г., 768с.

32.Попов В.Б. Основы информационных и телекоммуникационных технологий. − М.: Финансы и статистика, 2005 г., 224.

33. Preiss B.R. Data structures and algoritms with object-oriented design patterns in C++. John Wiley and Sons. 2005.

34.Naiburg Eric UML for Database Design Addison Wesley, 2004

7. Учебные пособия и тексты лекций

35. Б.А.Бегалов, С.А.Новосардова. Образовательная технология по дисциплине «Информационные технологии и системы». Из серии «Технология обучения в экономическом образовании». – Т.: ТГЭУ, 2006 г., 128 с.

36.Информатика . Учеб. пособ / Под общ. ред. И.А.Чернопустовой. - СПб.: Питер, 2005. – 272с.

37.Жуковская И.Е. Объектно-ориентированное программирование. Тексты лекций – Т.: ТГЭУ, 2005 г. 173 с.

8. Монографии

38. Бегалов Б.А. Технология процессов формирования информационно-коммуникационного рынка. Монография. – Т.: Фан, 2000.

9. Докторские, кандидатские и магистерские диссертации

39. Охунов Д.М. Исследование и разработка маркетинговых автоматизированных информационных систем предприятий. Диссертация на соискание степени кандидата экономических наук. − Т.: ТГЭУ, 2005, 138с.

40. Худайкулова Б.Б. Исследование рынка информационных технологий в туризме. Магистерская диссертация. − Т.: ТГЭУ, 2005, 110 с.

10.Материалы научно – практических конференций

41. Труды международной конференции «Роль и значение телекоммуникаций и информационных технологий в современном обществе», − Т.: 27-30 сентября, 2005 г.

42. «Применение INTERNET в учебном процессе» материалы научно практической конференции, 2002 год, Москва-Ташкент.

11. Газеты и журналы

43. Журнал «Информационные технологии».

44. Журнал «Научно-техническая информация».

45. Журнал «Информатика».

46. Журнал «Мир ПК».

47.Журнал «Компьютер Пресс».

12. Статистические сборники

48. Мониторинг развития информационно-коммуникационных технологий в Узбекистане. 2003 – 2005 гг. Т., 70 с.

49. Деловая среда в Узбекистане глазами представителей малого бизнеса. Ташкент. 2002-2005 гг. 170 с.

13. Интернет сайты

50.http://www.iite – сайт ЮНЕСКО «Информационные технологии в образовании».

51.http://www.borland.com/delphi_net/ - Официальный сайт Borland Delphi.

52.http://delphin.xost.ru/ - сайт помощи для программирования в среде Delphi.

53.http://www.ournet.md/~delphi - Сайт часто задаваемых вопросов и ответов по DELPHI, документация по функциям Windows API, уроки по Delphi, Delphi 4 - интерактивно, работа с DirectX и MMX.

14.Электронные учебники виртуальной библиотеки

54. Б.А.Бегалов. Введение в базы данных. Электронное учебное пособие. Программисты: А.Бобожонов, У.Муслимов. Ташкент. ТГЭУ. 2004 г.

15. Выпускные квалификационные работы

55.Касимова Л.С. Автоматизация управления торгово–закупочной деятельностью фирмы. Выпускная квалификационная работа. Ташкент, ТГЭУ, 2004 г., 81 с.

56. Ермакова П.Н. «Автоматизация расчета проектируемой суммы от реализации товаров». Выпускная квалификационная работа. Ташкент, ТГЭУ, 2004 г., 95 с.

- Сайт часто задаваемых вопросов и ответов по DELPHI, документация по функциям Windows API, уроки по Delphi, Delphi 4 - интерактивно, работа с DirectX и MMX.

16. Электронные учебники виртуальной библиотеки

54. Б.А.Бегалов. Введение в базы данных. Электронное учебное пособие. Программисты: А.Бобожонов, У.Муслимов. Ташкент. ТГЭУ. 2004 г.

17. Выпускные квалификационные работы

55.Касимова Л.С. Автоматизация управления торгово–закупочной деятельностью фирмы. Выпускная квалификационная работа. Ташкент, ТГЭУ, 2004 г., 81 с.

56. Ермакова П.Н. «Автоматизация расчета проектируемой суммы от реализации товаров». Выпускная квалификационная работа. Ташкент, ТГЭУ, 2004 г., 95 с.