Глава 7

Программирование с использованием графического интерфейса пользователя

В этой главе...

- Структурный стиль программирования
- Объектно-ориентированный стиль программирования
- Повышение скорости работы
- Разделение графической и логической составляющих программы
- Сообщения
- Программа для шифрования
- Резюме
- Контрольные вопросы

До сих пор программы, описанные в книге, были относительно небольшими и почти полностью располагались в одном блоке операторов. Однако при разработке большой программы поместить весь ее исходный код в один блок невозможно. В этой главе рассматриваются терминология и методы программирования больших и достаточно сложных задач.

Структурный стиль программирования

Под структурным программированием понимается метод программирования, обеспечивающий создание программы на основе отдельных элементов, которые логически объединены и связаны со структурой решаемой задачи. В основе метода структурного программирования лежит подход, при котором исходная задача разделяется на несколько достаточно крупных задач, каждая из которых реализуется, как отдельная программа. В свою очередь, подзадача делится на задачи более низкого уровня, также реализуемые, как отдельные программы. Таким образом, общая структура программы отражает структуру задачи, для решения которой она предназначена.

Например, задача работы с базой данных может быть разделена на две подзадачи: задачу добавления информации в базу данных и задачу обработки информации из базы данных. Задача добавления информации может быть также разделена, например, на три подзадачи: получение информации, проверка непротиворечивости и непосредственное добавление информации в базу.

Метод структурного программирования предполагает использование технологии разработки программы, получившей название “сверху вниз”. Его суть состоит в том, что сначала разрабатывается часть программы, реализующая наиболее общий алгоритм, а решение частных задач откладывается на более поздний срок. Процедуры, предназначенные для решения отложенных задач, заменяются процедурами-заглушками.

В языке Delphi структурный стиль программирования реализуется как модульное программирование, или программирование с помощью компонентов. Например, в обычную стереосистему входят приемник, магнитофон, проигрыватель CD, усилитель и динамики. Производители выпускают эти компоненты отдельно, а потребители могут объединять их в разных сочетаниях. Для потребителей такие стереосистемы привлекательны тем, что их легко модернизировать: достаточно купить и установить новый компонент. Поскольку технические характеристики непрерывно совершенствуются, заменять компоненты можно один за другим по мере необходимости, что значительно дешевле замены всей стереосистемы.

Компоненты стереосистемы похожи на модули в Delphi. Термин *модульное программирование* означает разработку программ, состоящих из отдельных модулей исходного кода. Структурирование программы — это ее разбивка на отдельные подпрограммы, в которых содержатся управляющие структуры и коды, необходимые для решения определенной частной задачи. В каждом модуле может находиться произвольное количество подпрограмм. Программы, написанные на Delphi, структурированы по своей природе, поскольку сам язык строго структурирован. Больше того, все приложения, разработанные в среде Delphi, являются модульными программами, потому что среда разработки создает отдельные модули исходного кода.

Модульное программирование предоставляет ряд существенных преимуществ. Во-первых, легко логически организовать программу, разделяя сложные задачи на простые подзадачи. Такое разделение называется *пошаговым решением* и является составной частью методологии разработки “сверху вниз”.

Во-вторых, модульное программирование упрощает отладку и делает код более понятным. В-третьих, как и в случае со стереосистемой, программу можно модернизировать путем замены отдельных модулей, не затрагивая другие модули.

И в-четвертых, концепция модульного программирования позволяет использовать разработанные коды повторно. Повторное использование кодов — важнейший принцип компьютерного программирования. Вы уже неоднократно с ним сталкивались, ведь кнопки, надписи, области просмотра — это не что иное, как разработанные ранее коды. Программист уже разработал подпрограмму, решающую некоторую задачу, поэтому нет необходимости разрабатывать ее заново, если в другом приложении решается та же задача. Можно просто использовать код подпрограммы повторно. Но здесь особую важность приобретает качество документирования. Код подпрограммы должен содержать комментарии с полным описанием ее входных и выходных данных.

Объектно-ориентированный стиль программирования

Исторически сложилось так, что программирование возникло и развивалось как процедурное программирование. При процедурном программировании предполагается, что основой программы является алгоритм — процедура обработки данных. Объектно-ориентированное программирование (ООП) — это методика разработки программ, в основе которой лежит понятие объекта как некоторой структуры, соответствующей объекту реального мира и его поведению. Задача, решаемая с использованием методики ООП, описывается в терминах объектов и операций над ними. Программа при таком подходе представляет собой набор объектов и связей между ними.

Работа с объектами требует некоторой перестройки способа мышления программиста. Методология ООП построена таким образом, чтобы структуры данных были более близки к объектам реального мира и не являлись абстрактными математическими понятиями.

Например, компьютер можно рассматривать как объект, содержащий отдельные платы и блоки, которые также можно рассматривать как объекты. Все это может быть описано в терминах ООП. Отдельную плату также можно рассматривать как комбинацию отдельных объектов, поэтому иногда трудно решить, сколько объектов следует создавать. Здесь необходим компромисс между сложностью отдельных объектов и их количеством, и достижение этого компромисса является немаловажной задачей (в некотором роде — даже искусством).

Наиболее общее правило гласит: всякий раз, когда одни структуры данных присутствуют внутри других структур данных, причем имеют динамический характер, следует применять средства ООП.

Объекты — это динамически размещаемые структуры. Тип объекта называется *классом*. При создании нового типа объекта фактически создается класс. При создании новой программной конструкции на основе некоторого класса создается *экземпляр класса*, или объект. Каждый объект содержит уникальную копию каждого поля, определенного в его базовом классе. Все объекты одного класса имеют одни и те же методы. Для создания и удаления объектов используются специальные методы, называемые *конструкторами* и *деструкторами*.

Класс — это тип объекта, а объект — это экземпляр (переменная) класса.

Переменная, обозначающая объект, фактически является указателем, ссылающимся на данные объекта в памяти. Следовательно, на один и тот же объект могут ссылаться несколько объектных переменных. Поскольку объектные переменные являются указателями, они могут содержать значение `nil`, указывающее, что объектная переменная не ссылается ни на какой объект. Однако, в отличие от указателей, объектная переменная для доступа к объекту не требует разыменования. Например, оператор

```
Edit1.Text := 'NewData';
```

присваивает значение `'NewData'` свойству Text поля ввода `Edit1` (элементы управления являются объектами). Оператор разыменования при этом не используется.

Операционная система Windows управляется событиями. Например, после щелчка на кнопке она генерирует событие, которое сопровождается рассылкой соответствующих сообщений. Компоненты и элементы управления в Delphi фактически являются объектами, методы которых активизируются после получения сообщений операционной системы.

Три фундаментальные составляющие ООП — это *инкапсуляция*, *наследование* и *полиморфизм*.

Инкапсуляция означает объединение всех данных об объекте и характеристик его поведения в одном блоке. Таким образом, объект содержит свойства и методы, использующие эти свойства, при этом предоставляются средства для сокрытия данных, т.е. инкапсулированные данные могут оставаться недоступными для пользователя и внешней программы.

Наследование позволяет расширять классы и способствует созданию родительско-дочерних отношений между объектами. Например, в приложении базы данных, в которой хранится информация о служащих компании, могут быть определены классы

Employee (Служащий) и Manager (Менеджер). Класс Employee содержит информацию о служащих — фамилии, номера карточек социального страхования и т.д. Каждый менеджер также является служащим, поэтому для него необходимо хранить ту же информацию, что и для служащих, плюс некоторые дополнительные сведения. Следовательно, между этими классами существует логическое отношение: класс Manager образует надмножество класса Employee. Класс Manager (дочерний класс) наследует все свойства и методы класса Employee, однако, кроме наследованных, он имеет и собственные свойства и методы.

Полиморфизм (дословно — "способность проявляться во многих формах") означает, что программа может обрабатывать объект по-разному в зависимости от его класса. Это достигается путем переопределения методов родительского класса в его дочерних классах. Пусть, например, класс Shape (Фигура) является базовым для классов Circle (Круг), Rectangle (Прямоугольник) и Triangle (Треугольник). Принцип полиморфизма позволяет определить для всех этих классов метод Area, вычисляющий площадь фигуры. Для объекта каждого дочернего класса площадь вычисляется по-разному, однако, поскольку метод Area переопределяется (т.е. применяется метод соответствующего дочернего класса), программист может вызывать метод Area, не уточняя вид фигуры. При этом до момента вызова метода в программе вид фигуры вообще может быть неизвестен, и ее характеристики будут определены только в момент вызова.

Реализация принципов ООП в Delphi

В Delphi методом является процедура или функция, определенная в классе. Рассмотрим некоторые понятия. *Полем* называется переменная, входящая в состав объекта. *Свойство* — это интерфейс, посредством которого программа получает доступ к значению поля, связанного со свойством. Со свойством также связаны *спецификаторы доступа* к переменной, определяющие, что происходит при записи значения в поле и при его чтении для соответствующего свойства. Однако поле может и не быть связанным со свойством. В этом случае доступ к нему выполняется, как к обычной переменной.

Классы и объекты — это современные средства для представления компьютерных данных. Впервые классы появились в популярных языках программирования в середине 80-х годов. Поле — это переменная, входящая в состав объекта. Свойство — это интерфейс для доступа к полю. Метод — это подпрограмма, определенная в классе.

Надеюсь, вы обратили внимание, что в момент присвоения значения, например, свойству Text, входящему в состав поля редактирования, вид надписи на экране изменяется. Но почему это происходит, ведь операция присвоения состоит только в изменении содержимого ячейки памяти? Однако для свойств это не так. Свойства тем и отличаются от полей объектов (и от других переменных), что при чтении или записи в поле свойства компьютер выполняет дополнительные действия, определенные спецификаторами доступа.

Преобразуем в класс написанные ранее процедуры для шифрования и расшифровки файлов и рассмотрим несколько моментов, связанных с созданием классов. Сначала создадим модуль CriptClass, для чего выберем команду меню File⇨New⇨Other⇨Delphi Files⇨Unit и в созданный модуль введем код, приведенный в листинге 7.1. Затем сохраним модуль под именем CriptClass.pas.

Листинг 7.1. Класс для шифрования и расшифровки файлов

```
unit UCript;
{ Класс для шифрования и расшифровки файлов.
  Автор: Галисеев Г.В., Киев }
interface
uses
```

```
  SysUtils, Dialogs, Classes;
type
  TCript = class
  private
    FCharCounter,
    FCyrcleCounter: Byte;
    FSum: Integer;
    FPassword: string;
  protected
    function RandomGenerator: Integer;
    function Coder(bt: Byte): Byte;
    function Decoder(bt: Byte): Byte;
  public
    constructor Create;
    procedure CriptFile(const inFileName:string);
    procedure set_Password(const Value: &string);
    property Password: string read FPassword write set_Password;
  end;

implementation
uses FCript;
{ TCript }
constructor TCript.Create;
begin
  inherited Create;
  FCharCounter := 1;
  FSum := 0;
end;

procedure TCript.CriptFile(const inFileName: string);
var
  outFileName: string;
  inFile,
  outFile: TFileStream;
  bt: Byte;
  oldExt: string;
  AnsiExt: AnsiString;
  i: Longint;
  lengthExt: Byte;
begin
  outFileName := inFileName;
  oldExt := ExtractFileExt(inFileName);
  if  oldExt<>'.xxx'  then
  begin     { Зашифровать }
    outFileName := StringReplace(outFileName,oldExt, '.xxx',
[rfIgnoreCase]);
    outFile := TFileStream.Create(outFileName, fmCreate);
    lengthExt := Length(oldExt);
    outFile.WriteBuffer(lengthExt);
    AnsiExt := oldExt;
    for i:=1 to lengthExt do
      outFile.WriteBuffer(AnsiExt[i]);
    inFile := TFileStream.Create(inFileName, fmOpenRead);
    for i:=1 to inFile.Size do begin
      inFile.ReadBuffer(bt);
      bt := coder(bt);
      outFile.WriteBuffer(bt);
    end;
```

```
  end else
  begin      { Расшифровать }
    inFile := TFileStream.Create(inFileName, fmOpenRead);
    inFile.ReadBuffer(lengthExt);
    SetLength(AnsiExt,lengthExt);
    for i:=1 to lengthExt do
      inFile.ReadBuffer(AnsiExt[i]);
    oldExt := AnsiExt;
    outFileName := StringReplace(outFileName,'.xxx', oldExt,
[rfIgnoreCase]);
    if not FileExists(outFileName) then begin
      outFile := TFileStream.Create(outFileName, fmCreate);
      for i:=1 to inFile.Size-lengthExt-1 do begin
        inFile.ReadBuffer(bt);
        bt := decoder(bt);
        outFile.WriteBuffer(bt);
      end;
    end;
  end;
  inFile.Free;
  outFile.Free;
end;

function TCript.Coder(Bt: Byte): Byte;
var
  Temp: Integer;
begin
  Temp := Bt - RandomGenerator;
  if Temp<0 then Result := Byte(Temp + 256)
  else Result := Byte(Temp);
end;

function TCript.Decoder(Bt:Byte): Byte;
var
  Temp: Integer;
begin
  Temp := Bt + RandomGenerator;
  if Temp>255 then Result := Byte(Temp - 256)
  else Result := Byte(Temp);
end;

function TCript.RandomGenerator: Integer;
var
  j: Integer;
begin
  FCyrcleCounter := FCharCounter;
  for j:=0 to Length(FPassword)-2 do
  begin
    Inc(FSum,Integer(FPassword[FCyrcleCounter]));
    if FCyrcleCounter=Length(FPassword) then
      FCyrcleCounter := 0;
    Inc(FCyrcleCounter);
  end;
  Inc(FCharCounter);
  if FCharCounter=Length(FPassword) then FCharCounter := 1;
  if FSum>255 then Result := FSum mod 255
  else Result := FSum;
end;
```

```
procedure TCript.set_Password(const Value: &string);
begin
  FPassword := Value;
end;
end.
```

В листинге использованы те же процедуры, что и в главе 4, в которой приведено их подробное описание, но в данном случае эти процедуры являются методами класса `TCript`. Методы `RandomGenerator`, `Coder` и `Decoder` объявлены в защищенном (`protected`) интерфейсе, т.е. их нельзя вызвать из внешней программы, но они могут быть переопределены в производных классах, где их область видимости может быть расширена. Возможность переопределения позволяет разрабатывать улучшенные методы и использовать их в дальнейшем, не затрагивая исходный класс и, соответственно, не внося в него никаких дополнительных ошибок. Для разработки нового метода просто создается производный класс, в котором вводится только один новый метод, и отладка будет выполняться только для производного класса, что значительно проще и надежнее, чем вносить изменения в исходный класс и вновь его тестировать.

Обратите внимание, что в модуле используется константа `FEXT`, которая подставляется в тех местах программы, где необходимо использовать определенное значение. Всегда делайте только так, а не подставляйте непосредственно значения, поскольку при необходимости внести изменения вы должны будете найти все места, где поставлены значения, и заменить их. Могу вас уверить, что если таких мест несколько, то одно из них вы обязательно пропустите и потом будете долго разбираться, почему программа работает не так, как задумано. Если же использовать именованную константу, то достаточно будет установить для нее новое значение, и программа будет эффективно работать с измененными данными.

Процедуры `RandomGenerator`, `Coder` и `Decoder` не изменены, но теперь они оформлены, как методы класса, т.е. к их именам добавлено имя класса и в классе объявлены процедуры. Как уже упоминалось, при создании класса необходимо в самом классе объявить все необходимые методы, щелкнуть правой кнопкой мыши на классе и в появившемся контекстном меню выбрать пункт Complete class at cursor (Закончить реализацию класса). Все необходимые заготовки для методов будут автоматически созданы и размещены в разделе реализации. Необходимо заполнить их в соответствии с заданным алгоритмом (в нашем случае — просто скопировать из ранее разработанных процедур).

Также обратите внимание на идентификаторы. Одной из особенностей профессионально написанной программы является продуманный выбор всех идентификаторов, что значительно облегчает понимание программы и не требует дополнительных описаний. Это называется *неявным документированием*, и необходимо выработать в себе привычку присваивать четкие и недвусмысленные имена всем составляющим программы, даже если вы пишете программу для себя. При программировании на Delphi рекомендуется по возможности избегать сокращений и использовать слова полностью, разделяя их прописными буквами. Но опять же, не следует этим увлекаться, так как слишком длинные идентификаторы могут только затруднить понимание программы. Хороший стиль программирования придет со временем, если постоянно обращать на это внимание и стремиться к совершенству.

Повышение скорости работы

Вероятно, вы слышали об уникальных программистах, которые работают на порядок продуктивнее остальных. Получается, что один программист работает за десятерых. Возможно ли это? Ясно, что они не могут набирать текст в десять раз быстрее, чем рядовые программисты. Достигается это в результате оптимизации затрат и выра-

ботки правильной стратегии. Даже суперпрограммист не сможет кодировать в десять раз быстрее среднего программиста. Он применяет стратегию, позволяющую накапливать затраченные усилия. Одна из составляющих такой стратегии — написание *конвергентного кода*, или, говоря более доступным языком, кода, который можно использовать многократно. Синонимами слова "конвергенция" можно считать слова "сближение" и "приобретение общих свойств". Следовательно, необходимо уметь выделять существенные фрагменты программ, которые можно использовать в различных ситуациях. Десятикратному повышению скорости создания кода способствует использование именно такого кода.

Способы, с помощью которых можно заставить себя следовать этой стратегии и писать конвергентные программы, основываются на двух правилах. Во-первых, постоянно проверяйте свой код. Как говорится, гениальность — это 1% таланта + 99% труда. Значит, стремиться к совершенству нужно всегда. И во-вторых, выработайте в себе привычку оформлять алгоритм в виде процедуры, как только заметите повторение нескольких строк.

Способность выделять повторяющиеся фрагменты приходит с практикой. И если в процессе работы вы постоянно находите такие фрагменты, то этим вы облегчаете себе работу. И наоборот, если вы все улучшения планируете делать после написания программы, вы только потеряете время. Стараясь усовершенствовать большой блок программы за короткое время, можно только усугубить положение и дестабилизировать уже работающий код.

Использовать код неоднократно можно тремя способами: 1) создать производные классы, включая компоненты, для повторного использования кода; 2) определить шаблон компонента — нововведение в Delphi для применения новых возможностей в одном или нескольких компонентах; 3) создать шаблон формы, который позволит многократно применять целые формы или фреймы со всем их содержимым.

В листинге 7.1 нет никаких оригинальных алгоритмов, просто небольшие фрагменты кода для открытия и создания файлов объединены в методы, что позволяет для открытия файла вместо кода использовать всего одну команду. Поскольку в некоторых программах работа с файлами происходит довольно интенсивно, экономия времени от использования класса `TCrtFile` при написании таких программ будет довольно существенной.

В следующем разделе продолжим эту тему и разработаем графический интерфейс пользователя, который можно неоднократно применять во многих ситуациях.

Разделение графической и логической составляющих программы

В первых главах этой книги использовались приложения типа Console Application, в которых не было графического интерфейса пользователя и ввод-вывод выполнялся в стандартное окно DOS. При этом возникали значительные неудобства при вводе и выводе данных (например, приходилось полностью вводить имя и путь файла, что трудно сделать без ошибки) или при выводе, когда необходимо было упорядочить выводимую информацию определенным образом.

Такие программы, даже если они отлично выполняют алгоритмическую часть, но неудобны в работе, обычно не завоевывают симпатии пользователей. Например, программа для сжатия данных ZIP — далеко не самая лучшая по своим возможностям. Существует не менее сотни устойчиво работающих алгоритмов сжатия, и ZIP даже не в первой десятке по скорости и степени сжатия. Тем не менее все пользуются только ZIP и не обращают внимания на другие алгоритмы, потому что ZIP удобен в использовании, имеет хорошо продуманный графический интерфейс и отвечает ос-

новным потребностям пользователей. Так что если вы разработаете алгоритм, сжимающий даже в два раза лучше, чем ZIP, но не имеющий таких возможностей, как ZIP, на него вряд ли обратят внимание широкие массы пользователей.

Чтобы программа имела коммерческий успех, она должна быть удобной в использовании. Поэтому задача создания современного графического интерфейса довольно актуальна, и необходимо научиться разрабатывать графические интерфейсы. Delphi предоставляет для этого наиболее совершенные средства, и научиться ими пользоваться совсем несложно. Для начала разработаем простой универсальный графический интерфейс, который можно использовать во многих ситуациях.

Поставим следующую задачу: разработать шаблон графического интерфейса, который можно использовать для ввода-вывода информации и который позволяет выполнять поиск файлов для открытия и сохранения. Его можно использовать как компонент в любой программе. Когда я поставил для себя эту задачу, у меня получился такой графический интерфейс (рис. 7.1).

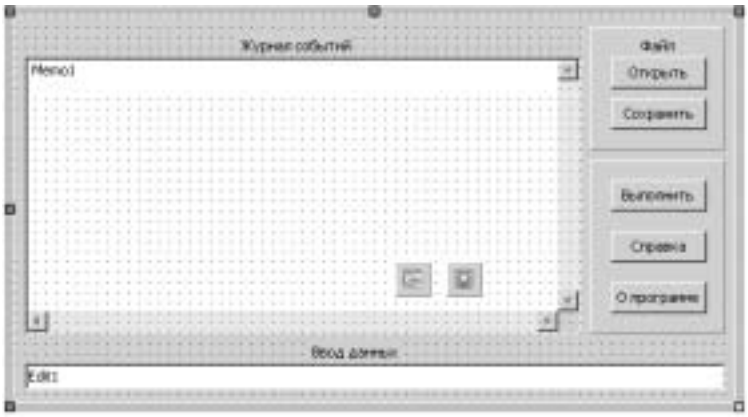

Рис. 7.1. Простой графический интерфейс пользователя

Ниже приведен соответствующий ему листинг 7.2.

Листинг 7.2. Графический интерфейс пользователя

```
unit FMyGUI;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, Borland.Vcl.StdCtrls, Borland.Vcl.ExtCtrls,
  System.ComponentModel;
type
  TMyGUI = class(TFrame)
    Memo1: TMemo;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Panel1: TPanel;
    Button1: TButton;
    Button2: TButton;
    Panel2: TPanel;
    Button3: TButton;
    Button4: TButton;
```

```
    Button5: TButton;
    Label3: TLabel;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
    FInFileName,
    FOutFileName,
    FInString: string;
  public
    procedure set_InString(const Value: &string);
  public
    { Public declarations }
    Exec,
    Help,
    About,
    isString: Boolean;
    property InFileName: string read FInFileName;
    property OutFileName: string read FOutFileName;
    property InString: string read FInString write set_InString;
  end;

implementation
{$R *.nfm}
procedure TMyGUI.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    FInFileName := OpenDialog1.FileName;
    Memo1.Lines.Append('Входной файл:   '+InFileName);
  end;
end;

procedure TMyGUI.Button2Click(Sender: TObject);
begin
  if SaveDialog1.Execute then begin
    FOutFileName := OpenDialog1.FileName;
    Memo1.Lines.Append('Выходной файл:  '+OutFileName);
  end;
end;

procedure TMyGUI.Button3Click(Sender: TObject);
begin
  Exec := True;
end;

procedure TMyGUI.Button4Click(Sender: TObject);
begin
  Help := True;
end;

procedure TMyGUI.Button5Click(Sender: TObject);
begin
  About := True;
end;
```

```
procedure TMyGUI.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  if not isString then begin
    FInString := FInString + Key;
    if Key=#13 then isString := True;
  end;
end;

procedure TMyGUI.set_InString(const Value: &string);
begin
  FInString := Value;
end;
end.
```

Идея, использованная при написании этой программы, очень проста. После щелчка на каждой кнопке интерфейса создается событие, в результате которого соответствующая переменная принимает значение `True`. Напомню, что для создания процедуры обработки события необходимо использовать инспектор объектов, в котором для соответствующего компонента следует выбрать вкладку Events (События) и дважды щелкнуть на поле OnClick (Во время щелчка). В редакторе кодов автоматически будет создана заготовка процедуры обработки события, и курсор будет установлен в том месте, где нужно ввести код. Необходимые переменные, `Exec`, `Help` и `About`, перечислены в открытом интерфейсе класса. Обратите внимание, что имена этих переменных начинаются с прописной буквы. Хотя компилятор Delphi не реагирует на регистр клавиатуры и для него не важно, прописная буква или строчная, это может служить дополнительным информативным признаком. Например, в некоторых языках программирования имена всех внутренних (закрытых) переменных принято начинать со строчной буквы, а открытые переменные, т.е. переменные, к которым можно обратиться из-за пределов класса, — с прописной буквы. Это помогает ориентироваться в программе.

Функции открытия и сохранения файлов, а также обработчик нажатия клавиш реализованы в шаблоне, полученные значения сохраняются в закрытых переменных `FInFileName`, `FOutFileName` и `FInString` типа `string`, и для доступа к ним реализованы соответствующие свойства.

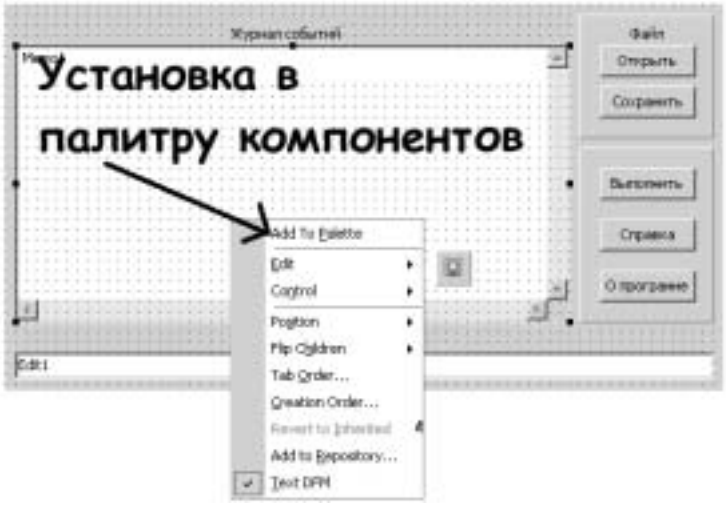

Рис. 7.2. Установка шаблона в палитру компонентов

Для создания шаблона необходимо открыть проект типа VCL Form Application, дополнительно открыть модуль VCL Frame и разместить в нем компоненты, как показано на рис. 7.1. Какие при этом использованы компоненты, хорошо видно в приведенном листинге. Необходимо установить и соответствующие свойства. Например, для компонента `Memo1` свойству ScrollBars необходимо присвоить значение `ssBoth`, чтобы были задействованы обе полосы прокрутки и можно было просматривать все выведенные значения.

После размещения всех компонентов и установки их свойств необходимо щелкнуть правой кнопкой мыши на фрейме и в появившемся контекстном меню выбрать пункт Add To Palette (рис. 7.2).

В появившемся окне Component Template Information введите необходимое имя шаблона и имя раздела, в котором будет размещен шаблон, и щелкните на кнопке OK. Если раздела с указанным именем не существует, он будет создан автоматически. В этом же окне можно выбрать и подходящую пиктограмму для нового шаблона.

Теперь шаблон установлен в палитре компонентов (на рис. 7.3 показана установка в раздел Templates), и им можно пользоваться.

Рис. 7.3. Новый раздел Templates с установленным шаблоном

Чтобы проверить вновь созданный шаблон, разработаем небольшую программу, для чего созданный шаблон разместим в форме. В листинге 7.3 приведена программа для проверки шаблона. Шаблон назван `MyGUI`, кроме него, в форме размещены надпись с названием программы и компонент `TTimer`. Это единственный компонент, который необходим для работы с шаблоном графического интерфейса. Опять же, очень простая идея: таймер периодически вызывает процедуру, в которой опрашиваются все поля графического интерфейса, и если их значение равно `True`, значит, был выполнен щелчок на соответствующей кнопке. При этом вызывается процедура для обработки данного события. Значение поля возвращается в состояние `False` до следующего щелчка. Период срабатывания таймера устанавливается в свойстве Interval для компонента `Timer1`.

Все это хорошо видно в листинге 7.3, в котором задействованы только процедуры вызова диалоговых окон для открытия и сохранения файлов, а для поля ввода `Edit1` написана небольшая программа, суммирующая числа. Последовательно вводя целые числа и нажимая клавишу <Enter>, можно циклически получать суммы двух чисел, как показано на рис. 7.4.

Следует добавить, что для компонента `Edit1` свойству TabOrder необходимо присвоить значение `0`. При этом в момент установки графического интерфейса курсор всегда будет помещаться в поле ввода `Edit1`. Последовательно нажимая клавишу <Tab>, можно активизировать следующий компонент в соответствии с тем порядком, который был задан пользователем или установлен автоматически при размещении компонентов в фрейме. Свойство TabOrder используется для задания порядка.

Листинг 7.3. Программа для проверки шаблона

```
unit FGUI;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, System.ComponentModel, SimpleGUI,
  Borland.Vcl.ExtCtrls, Borland.Vcl.StdCtrls;
type
  TForm1 = class(TForm)
```

```
    MyGUI: TFrame2;
    Label1: TLabel;
    Timer1: TTimer;
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
    procedure ProcOpen;
    procedure ProcSave;
    procedure ProcExec;
    procedure ProcHelp;
    procedure ProcAbout;
    procedure ProcKeyPress;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.nfm}
procedure TForm1.ProcAbout;
begin
end;
procedure TForm1.ProcExec;
begin
end;
procedure TForm1.ProcHelp;
begin
end;
procedure TForm1.ProcKeyPress;
const
  X: string='';
  Y: string='';
  Counter: Integer = 0;
begin
  if MyGUI.KeyChar=#13 then
  begin
    if Counter=0 then
    begin
      X := MyGUI.Edit1.Text;
      MyGUI.Memo1.Lines.Add('Первое слагаемое = '+X);
      Inc(Counter);
    end else
    begin
      Y := MyGUI.Edit1.Text;
      MyGUI.Memo1.Lines.Add('Второе слагаемое = '+Y);
      MyGUI.Memo1.Lines.Add('Сумма = '+
        IntToStr(StrToInt(Y)+StrToInt(X)));
      Counter := 0;
    end;
    MyGUI.Edit1.Clear;
  end;
end;

procedure TForm1.ProcOpen;
begin
  MyGUI.OpenDialog1.Execute;
  if MyGUI.OpenDialog1.Files.Count=0 then
```

```
    MyGUI.Memo1.Lines.Add('Файлы не выбраны')
  else
    MyGUI.Memo1.Lines.Add('Выбрано '+
      IntToStr(MyGUI.OpenDialog1.Files.Count)+' файлов.');
end;

procedure TForm1.ProcSave;
begin
  MyGUI.SaveDialog1.Execute;
  if MyGUI.SaveDialog1.FileName='' then
    MyGUI.Memo1.Lines.Add('Файл не выбран')
  else
    MyGUI.Memo1.Lines.Add('Выбран '+MyGUI.SaveDialog1.FileName);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if MyGUI.Open=True then
  begin
    MyGUI.Open:=False;
    ProcOpen;
  end;
  if MyGUI.Save=True then
  begin
    MyGUI.Save:=False;
    ProcSave;
  end;
  if MyGUI.Exec=True then
  begin
    MyGUI.Exec:=False;
    ProcExec;
  end;
  if MyGUI.Help=True then
  begin
    MyGUI.Help:=False;
    ProcHelp;
  end;
  if MyGUI.About=True then
  begin
    MyGUI.About:=False;
    ProcAbout;
  end;
  if MyGUI.KeyPress=True then
  begin
    MyGUI.KeyPress:=False;
    ProcKeyPress;
  end;
end;
end.
```

Как видите, при разработке этой программы никаких изменений в шаблон графического интерфейса вносить не пришлось. Это полностью независимый фрагмент программы, который, однажды отладив, можно неоднократно использовать и быть уверенным в том, что он не будет причиной ошибок в программе. К этому надо стремиться — разрабатывать независимые фрагменты программы для постоянного использования и на их основе создавать программные комплексы. Кстати, язык Delphi работает именно по такому принципу. Большое количество отлаженных и устойчиво работающих компонентов на все случаи жизни позволяет быстро проектировать довольно

сложные программные комплексы. Необходимо только изучить соответствующие компоненты, что при наличии хорошей документации сделать не так уж трудно.

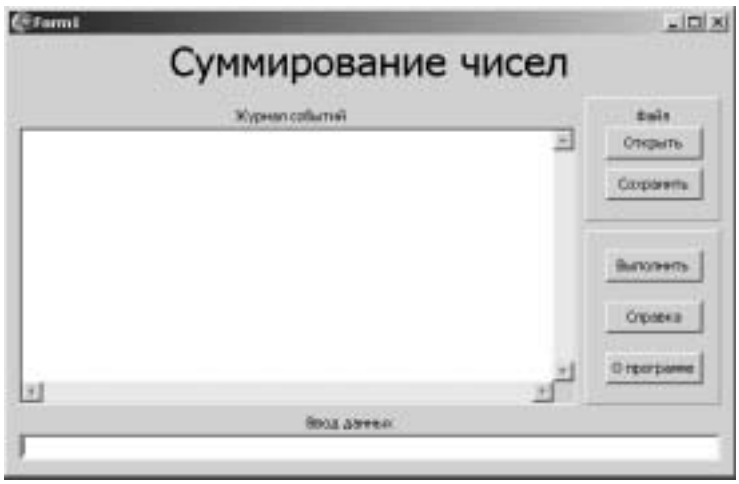

Рис. 7.4. Графический интерфейс для суммирования целых чисел

В Delphi широко развиты средства разработки интерфейса пользователя, и создать хороший интерфейс можно довольно быстро, при этом он будет отвечать всем требованиям программы. Изложенные выше идеи связи интерфейса с программой тоже могут пригодиться. В любом случае старайтесь не писать логику программы в том модуле, в котором разрабатываете интерфейс пользователя. Создавайте для этого дополнительные модули и в них отрабатывайте отдельные логические фрагменты программы, а связать интерфейс пользователя с модулями можно по-разному. Например, если известно, что вывод будет осуществляться в компонент типа `TMemo`, можно непосредственно в процедурах или функциях, из которых будет выполнен вывод, предусмотреть параметр типа `TMemo`.

Сообщения

В программе, приведенной в предыдущем разделе, для контроля состояния клавиш использовались способ периодического опроса переменных, связанных с клавишами, и вызов соответствующей процедуры, если переменная принимала значение `True`. Но у этого способа есть один недостаток — отсутствие обратной связи, т.е. нельзя изменить состояние графического интерфейса из основной программы. Значительно бо́льшую гибкость можно получить, если использовать для передачи состояния объектов *сообщения*. Например, нажав клавишу на клавиатуре, можно отослать следующее сообщение:

```
procedure TMyGUI.Edit1KeyPress(Sender: TObject; var Key:
  Char);
begin
  Key := Char(SendMessage(SENDTO,MSGKEYPRESS,Longint(Key),0));
end;
```

Для этого используется функция `SendMessage`, параметрами которой являются номер принимающего окна, номер сообщения и два числа типа `Longint`, с помощью которых можно передать дополнительную информацию. Как видно, функция имеет возвращаемое значение, которое устанавливается на приемном конце и возвращается

обратно функции. Таким образом, имеется возможность передать в графический интерфейс некоторые значения.

Процедура обработки сообщения должна быть объявлена следующим образом:

```
procedure WMMSGKEYPRESS(var Msg: TMessage); message MSGKEYPRESS;
```

Здесь директива `message` говорит о том, что это обработчик сообщения, номер которого — `MSGKEYPRESS`. В названии процедуры принято использовать идентификатор номера сообщения с подставленными вначале буквами `WM`.

Рассмотрим этот вопрос подробнее.

Что такое сообщение

Сообщение — извещение о некотором событии, отсылаемое системой Windows в адрес приложения. Любые действия пользователя — щелчок мышью, изменение размеров окна приложения, нажатие клавиши на клавиатуре — вынуждают Windows отправлять приложению сообщения о том, что произошло в системе.

Сообщение представляет собой запись, передаваемую приложению системой Windows. Эта запись содержит информацию о типе произошедшего события и дополнительную информацию, специфическую для данного сообщения. Например, для щелчка мышью запись содержит дополнительную информацию о координатах указателя мыши на момент щелчка и номер нажатой кнопки. Тип записи, используемый Delphi для представления сообщений Windows, называется `TMsg`. Он определен в модуле `Borland.VCL.Windows` следующим образом.

```
type

{ Message structure }
  tagMSG = packed record
    hwnd: HWND; ;      // Дескриптор (handle) окна-получателя
    message: UINT; ;   // Идентификатор сообщения
    wParam: WPARAM;    // 32 бит дополнительной информации
    lParam: LPARAM;    // 32 бит дополнительной информации
    time: DWORD; ;     // Время создания сообщения
    pt: TPoint; ;      // Позиция указателя мыши в момент создания
                       // сообщения
  end;
  TMsg = tagMSG;
  MSG = tagMSG;
```

Здесь `hwnd` — 32-битовый *дескриптор окна*, для которого предназначено сообщение. Окно может быть практически любым типом объекта на экране, поскольку Win32 поддерживает дескрипторы окон для большинства визуальных объектов (окон, диалоговых окон, кнопок, полей ввода и т.п.). В данном случае дескриптор можно рассматривать как порядковый номер.

`message` — это константа, соответствующая номеру сообщения. Системные константы сообщений (для Windows) определены в модуле `Borland.VCL.Windows`, а константы для пользовательских сообщений программист должен определить сам.

В поле `wParam` часто содержится константа, значение которой определяется типом сообщения.

В поле `lParam` чаще всего хранится индекс или указатель на некоторые данные в памяти. Поскольку поля `wParam`, `lParam` и `pt` имеют один и тот же размер, равный 32 бит, между ними допускается взаимное преобразование типов.

Типы сообщений

Интерфейс прикладных программ Windows 32 является интерфейсом операционной системы для взаимодействия с большинством приложений Windows. Здесь каждому сообщению Windows поставлено в соответствие определенное значение, которое заносится в поле `message` записи типа `TMsg`. В Delphi все эти константы определены в модуле `Borland.VCL.Messages` и их можно легко просмотреть с помощью любого текстового редактора или в среде разработки, для чего в разделе `uses` следует набрать имя подключаемого файла `Messages`, щелкнуть на нем правой кнопкой мыши и выбрать команду меню Open File at cursor. Обратите внимание, что имя каждой константы, представляющей тип сообщения, начинается с символов `WM` (т.е. Windows Message). В табл. 7.1 представлены некоторые наиболее распространенные сообщения Windows и их числовые коды.

Таблица 7.1. Некоторые сообщения Windows

Идентификатор сообщения	Значение	О чем сообщает окну
WM_ACTIVATE	$0006	Окно активизируется или дезактивизируется
WM_CHAR	$0102	От клавиши было отослано сообщение WM_KEYDOWN и WM_KEYUP
WM_CLOSE	$0010	Окно должно быть закрыто
WM_KEYDOWN	$0100	На клавиатуре была нажата клавиша
WM_KEYUP	$0101	Клавиша на клавиатуре была отпущена
WM_LBUTTONDOWN	$0201	Пользователь нажал левую кнопку мыши
WM_MOUSEMOVE	$0200	Указатель мыши переместился
WM_PAINT	$000F	Необходимо перерисовать клиентскую область окна
WM_TIMER	$0113	Произошло событие таймера
WM_QUIT	$0012	Программа должна быть завершена

Не будем подробно рассматривать сообщения Windows, так как компоненты библиотеки VCL (Visual Component Library) преобразуют большинство сообщений Win32 в события языка Delphi. Но если вы собираетесь серьезно изучать Delphi, то иметь понятие о работе системы сообщений Windows необходимо. Потребности программиста Delphi практически полностью удовлетворяются возможностями работы с событиями, предоставляемыми VCL, но при создании нестандартных приложений, особенно при разработке компонентов Delphi, потребуется непосредственная обработка сообщений Windows.

Система сообщений Delphi

Подпрограммы библиотеки VCL выполняют существенную часть обработки сообщений Windows в приложении, благодаря чему прикладному программисту не нужно беспокоиться о выборке сообщений из очереди и их передаче соответствующим процедурам окон. Кроме того, Delphi помещает информацию из записи типа `TMsg` в собственную запись типа `TMessage`, определение которой приведено ниже.

```
type
TMessage = record
  Msg: Cardinal;
  case Integer of
  0: (
```

```
    WParam: Longint;
    LParam: Longint;
    Result: Longint );
  1: (
    WParamLo: Word;
    WParamHi: Word;
    LParamLo: Word;
    LParamHi: Word;
    ResultLo: Word;
    ResultHi: Word );
end;
```

Обратите внимание, что в записи `TMessage` содержится меньше информации, чем в исходной записи `TMsg`. Это связано с тем, что Delphi берет часть обработки сообщений Windows на себя, и в запись `TMessage` помещается только та часть информации, которая необходима для дальнейшей обработки.

Важно отметить, что в записи `TMessage` содержится поле `Result` (Результат). Как уже говорилось, некоторые сообщения требуют возврата значения из процедуры окна после завершения их обработки. В Delphi такая задача решается совсем просто — достаточно поместить возвращаемое значение в поле `Result` записи `TMessage`.

Специализированные записи

В дополнение к обычной записи типа `TMessage` в Delphi определен набор специализированных записей для всех типов сообщений Windows. Они предназначены для того, чтобы предоставить программисту возможность работать со всей содержащейся в исходном сообщении Windows информацией без необходимости декодировать значения полей `wParam` и `lParam`. Определения всех типов этих записей находятся в модуле `Messages`. Ниже приведен класс `TWMMouse`, используемый для большинства типов сообщений Windows о событиях мыши.

```
type
  TWMPosition = class(TWMNoParams)
  protected
    function GetSmallPoint: TSmallPoint;
    procedure SetSmallPoint(Value: TSmallPoint);
  public
    property XPos: SmallInt read GetLParamLoSmall write
SetLParamLoSmall;
    property YPos: SmallInt read GetLParamHiSmall write
SetLParamHiSmall;
    property Pos: TSmallPoint read GetSmallPoint write SetSmallPoint;
  end;

  TWMMouse = class(TWMPosition)
    property Keys: Integer read GetWParam write SetWParam;
  end;
```

Все типы записей для конкретных сообщений о событиях мыши (например, `WM_LBUTTONDOWN` или `WM_RBUTTONUP`) определяются, как записи типа `TWMMouse`:

```
TWMRButtonUp = TWMMouse;
TWMLButtonDown = TWMMouse;
```

Специализированные записи сообщений определены практически для всех стандартных сообщений Windows.

Учитывая соглашение о присвоении имен, необходимо присваивать записи имя, соответствующее имени сообщения с префиксом `T`. К примеру, запись для сообщения `WM_SETFONT` должна содержать имя `TWMSetFont`.

Записи типа `TMessage` создаются для всех типов сообщений Windows и в любых ситуациях. Но работать с подобными записями не так удобно, как со специализированными.

Обработка сообщений

Обработка сообщений означает, что приложение соответствующим образом реагирует на получаемые от операционной системы сообщения. В стандартном приложении Windows обработка сообщений выполняется в процедурах окна. Но Delphi, частично обрабатывая сообщения, упрощает работу программиста, позволяя вместо одной процедуры для обработки всех типов сообщений создавать независимые процедуры для обработки сообщений каждого типа. Любая процедура обработки сообщений должна отвечать трем требованиям.

- Процедура должна быть методом класса.
- Процедура должна принимать по ссылке (параметр с ключевым словом `var`) один параметр типа `TMessage` или любого другого типа специализированного сообщения.
- Объявление процедуры должно содержать директиву `message`, за которой должна следовать константа, задающая тип обрабатываемого сообщения.

Ниже приведен пример объявления процедуры, обрабатывающей сообщение `WM_CHAR`:

```
procedure WMChar(var Mes: TWMChar); message WM_CHAR;
```

В соответствии с соглашением о присвоении имен обработчику сообщений следует присваивать то же имя, что и имя обрабатываемого им сообщения, но без символа подчеркивания.

В качестве примера напишем простую процедуру для обработки сообщения `WM_CHAR`, которая будет отображать ASCII-коды нажатых клавиш.

Создайте новый пустой проект. Разместите в форме надпись. Добавьте в закрытый интерфейс формы объявление функции

```
procedure WMChar(var Mes: TWMKEY); message WM_CHAR;
```

и опишите ее в разделе реализации.

Полный текст программы должен быть следующим.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
  private
    procedure WMChar(var Mes: TWMKEY); message WM_CHAR;
  end;
var
```

```
  Form1: TForm1;
implementation
{$R *.dfm}
{ TForm1 }
procedure TForm1.WMChar(var Mes: TWMKEY);
begin
  Label1.Caption := IntToStr(Mes.CharCode);
  inherited;
end;
end.
```

Теперь, нажав любую клавишу, вы получите отображение ее кода на экране. Простая, но полезная программка. Не надо копаться в справочниках, а можно сразу получить необходимое значение (рис. 7.5). Создать такую программу в Delphi — дело нескольких секунд.

Рис. 7.5. Программа для отображения кодов клавиш

Тип переменой `Mes` указан как `TWMKEY`, который соответствует типу `TWMCHAR`. Это можно увидеть в модуле `Borland.VCL.Messages.pas`. Рекомендую почаще заглядывать в исходные коды Delphi. Во-первых, можно научиться хорошему стилю программирования, а во-вторых, в прилагаемом справочнике иногда не отражены самые последние изменения (их можно увидеть только в исходных кодах).

Обратите внимание на использование в процедуре ключевого слова `inherited` (унаследованный), которое позволяет передать сообщение обработчику, принадлежащему базовому классу. В данном случае по ключевому слову `inherited` сообщение передается обработчику сообщения `WM_CHAR` класса `TForm`, в котором завершится его обработка.

В отличие от событий Delphi, обработка сообщений обязательна. Если в программе объявлены обработчики сообщений, то система Windows ожидает выполнения некоторых связанных с ней предопределенных действий. Компоненты VCL выполняют большую часть обработки своими встроенными средствами, для доступа к которым программисту достаточно лишь вызвать обработчик базового класса с помощью директивы `inherited`. В обработчике сообщения должны выполняться лишь те действия, которые нужны для приложения.

Событие OnMessage класса TApplication

Еще один метод обработки сообщений заключается в использовании события `OnMessage` класса `TApplication`. После назначения обработчика этому событию он будет вызываться всякий раз, как только сообщение будет получено из очереди и готово к обработке. Обработчик события `OnMessage` всегда вызывается до того, как система Windows получит возможность обработать сообщение. Обработчик `TApplication.OnMessage` имеет тип `TMessageEvent` и объявляется со списком параметров, приведенным ниже:

```
type
  TMessageEvent = procedure (var Msg: TMsg; var Handled:
    Boolean) of object;
```

Все параметры сообщения передаются обработчику события OnMessage в параметре Msg. Этот параметр имеет тип TMsg. Параметр Handled имеет тип Boolean и используется для указания, обработано ли сообщение.

Чтобы создать обработчик события OnMessage, можно воспользоваться компонентом TApplicationEvents, который расположен в разделе Additional палитры компонентов. Ниже приведен пример такого обработчика события.

```
var
  NumMessages: Integer;
  procedure TForm1.ApplicationEvents1Message(var Msg: tagMSG;
    var Handled: Boolean);
  begin
    Inc(NumMessages);
    Handled := False;
  end;
```

Событие OnMessage обладает одним существенным ограничением — оно перехватывает только те сообщения, которые выбираются из очереди, и не перехватывает те, которые передаются непосредственно процедурам окон приложения. Однако и это ограничение можно обойти.

Событие OnMessage перехватывает все сообщения, направляемые в адрес всех окон, относящихся к данному приложению. Обработчик этого события будет самой загруженной подпрограммой приложения, поскольку таких событий очень много — до тысяч в секунду. Поэтому избегайте выполнения в данной процедуре любых продолжительных операций, иначе работа всего приложения может существенно замедлиться.

Использование собственных типов сообщений

Аналогично тому, как система Windows отсылает свои сообщения различным окнам приложения, в самом приложении может возникнуть необходимость обмена сообщениями между его собственными окнами и элементами управления. Delphi предоставляет разработчику несколько способов обмена сообщениями в пределах приложения: метод Perform, работающий независимо от API Windows, а также функции интерфейса API Win32 SendMessage и PostMessage.

Метод Perform

Этим методом обладают все потомки класса TControl, входящие в состав библиотеки VCL. Метод Perform позволяет отослать сообщение любой форме или элементу управления, заданному именем экземпляра требуемого объекта. Методу Perform передаются три параметра — собственно сообщение и соответствующие ему параметры lParam и wParam. Для данного метода сделано несколько объявлений.

```
public function Perform(Msg: Cardinal,
  AObjectMsg: TCMObjectMsg): Integer;
public function Perform(Msg: Cardinal, WParam: Integer,
  LParam: Integer): Integer;
public function Perform(Msg: Cardinal,
  WParam: Integer, var LParam: ): Integer;
public function Perform(Msg: Cardinal,
  WParam: Integer, LParam: string): Integer;
public function Perform(Msg: Cardinal, WParam: Integer,
  var LParam: string, BufLen: Cardinal,
  ResultIsLen: Boolean): Integer;
```

Чтобы отослать сообщение форме или элементу управления, применяется следующий синтаксис:

```
RetVal := ControlName.Perform(MessageID, WParam, LParam);
```

При вызове функции `Perform` управление вызывающей программе не возвратится до тех пор, пока сообщение не будет обработано. Метод `Perform` упаковывает переданные ему параметры в запись типа `TMessage`, а затем вызывает метод `Dispatch` указанного объекта, чтобы передать сообщение, минуя систему передачи сообщений API Windows.

Уведомляющие сообщения

Уведомляющие сообщения, или *уведомления* (notifications), представляют собой сообщения, отсылаемые родительскому окну в том случае, когда в одном из его дочерних элементов управления происходит нечто, заслуживающее внимания родителя. Эти сообщения рассылаются только стандартными элементами управления Windows (кнопками, списками, раскрывающимися списками, полями редактирования) и общими элементами управления Windows (деревом объектов, списком объектов и т.п.). Щелчок или двойной щелчок на элементе управления, выбор текста в поле редактирования или перемещение бегунка полосы прокрутки — вот примеры событий, отсылающих уведомляющие сообщения.

Уведомления обрабатываются с помощью соответствующих процедур обработки, принадлежащих той форме, в которой содержится данный элемент управления. В табл. 7.2 перечислены уведомляющие сообщения для стандартных элементов управления Win32.

Таблица 7.2. Уведомления для стандартных элементов управления

Уведомление	Описание
Уведомления кнопки	
BN_CLICKED	Пользователь щелкнул на кнопке
BN_DISABLE	Кнопка переведена в неактивное состояние
BN_DOUBLECLICKED	Пользователь дважды щелкнул на кнопке
BN_HILITE	Пользователь выделил кнопку
BN_PAINT	Кнопка должна быть перерисована
BN_UNHILITE	Выделение кнопки должно быть отменено
Уведомления раскрывающегося списка	
CBN_CLOSEUP	Раскрытый список был закрыт пользователем
CBN_DBLCLK	Пользователь дважды щелкнул на строке
CBN_DROPDOWN	Список был раскрыт
CBN_EDITCHANGE	Пользователь изменил текст в поле ввода
CBN_EDITUPDATE	Требуется вывести измененный текст
CBN_ERRSPACE	Элементу управления не хватает памяти
CBN_KILLFOCUS	Список потерял фокус ввода
CBN_SELCHANGE	Выбран новый элемент списка
CBN_SELENDCANCEL	Пользователь отказался от сделанного им выбора
CBN_SELENDOK	Выбор пользователя корректен
CBN_SETFOCUS	Список получил фокус ввода

Окончание табл. 7.2

Уведомление	Описание
Уведомления поля ввода	
EN_CHANGE	Требуется обновление после внесения изменений
EN_ERRSPACE	Элементу управления не хватает памяти
EN_HSCROLL	Пользователь щелкнул на горизонтальной полосе прокрутки
EN_KILLFOCUS	Поле ввода потеряло фокус ввода
EM_MAXTEXT	Введенный текст был обрезан
EN_SETFOCUS	Поле ввода получило фокус ввода
EN_UPDATE	Требуется отображение введенного текста
EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки
Уведомления списков	
LBN_DBLCLK	Пользователь дважды щелкнул на строке
LBN_ERRSPACE	Элементу управления не хватает памяти
LBN_KILLFOCUS	Список потерял фокус ввода
LBN_SELCANCEL	Отмена выделения
LBN_SELCHANGE	Изменение выделения
LBN_SETFOCUS	Список получил фокус ввода

Для того чтобы увидеть, как передаются эти уведомления, можно создать форму с несколькими кнопками и надписью для вывода информации, а затем объявить обработчик сообщений:

```
procedure BTNMessage(var Msg: TMessage); message WM_COMMAND;
```

и описать процедуру обработчика:

```
procedure TForm1.BTNMessage(var Msg: TMessage);
begin
  Label1.Caption := IntToStr(Msg.Msg)+
    ' '+IntToStr(Msg.WParam)+
    ' '+IntToStr(Msg.LParam);
end;
```

Щелкнув на кнопке, вы увидите содержимое передаваемого сообщения.

Использование сообщений внутри приложения

Можно вынудить приложение отослать сообщение самому себе. Сделать это очень просто с помощью метода `Perform`. Сообщение должно обладать идентификатором в диапазоне от `WM_USER+100` до `$7FFFF` (этот диапазон Windows резервирует для сообщений пользователя), например:

```
const
  SX_MYMESSAGE = WM_USER + 100;

begin
  SomeForm.Perform(SX_MYMESSAGE, 0, 0);
end;
```

Затем для перехвата этого сообщения создайте обычный обработчик, выполняющий необходимые действия:

```
TForm1 = class(TForm)
private
  procedure SXMyMessage(var Msg: TMessage): message SX_MYMESSAGE;
end;

procedure TForm1.SXMyMessage(var Msg: TMessage);
begin
  MessageDlg('Сообщение принято!', mtInformation,
    [mbOk], 0);
end;
```

Как видно из примера, различия в обработке собственного сообщения и стандартного сообщения Windows невелики. Они заключаются в использовании идентификаторов от `WM_USER+100` и выше, а также в присвоении каждому сообщению имени, отображающего его смысл.

Никогда не отсылайте сообщений, значение которых превосходит `$7FFF` (десятичное `32767`), если вы не уверены абсолютно точно, что получатель способен правильно его обработать. Значение для `WM_USER` содержится в файле `Messages.pas`, и обычно оно равно `$400` (десятичное `1024`).

В листинге 7.4. приведена демонстрационная программа.

Листинг 7.4. Демонстрационная программа

```
unit FTMsg;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, Borland.Vcl.StdCtrls, System.ComponentModel;
const
  SX_MYMESSAGE = WM_USER + 100;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure SXMyMessage(var Msg: TMessage); message SX_MYMESSAGE;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
{$R *.nfm}
{ TForm1 }
procedure TForm1.SXMyMessage(var Msg: TMessage);
begin
  Label1.Caption := 'Сообщение принято!';
end;
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  Form1.Perform(SX_MYMESSAGE, 0, 0);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Caption := '';
end;
end.
```

Здесь щелчок на первой кнопке приведет к созданию сообщения для компонента `Form1`, которое будет принято формой, в результате чего надпись отобразит строку "Сообщение принято!". Щелчок на второй кнопке приведет к отображению пустой строки.

Широковещательные сообщения

Любой класс, производный от класса `TWinControl`, позволяет с помощью метода `Broadcast` отослать широковещательное сообщение любому элементу управления, владельцем которого он является. Эта технология используется в случаях, когда требуется отправить одно и то же сообщение группе компонентов. Например, чтобы отослать пользовательское сообщение по имени `UM_FOO` всем элементам управления, принадлежащим объекту `Panel1`, можно воспользоваться следующим кодом.

```
var
M: TMessage;
begin
  with M do
  begin
    Message := UM_FOO;
    wParam := 0;
    lParam := 0;
    Result := 0;
  end;
  Panel1.Broadcast(M);
end;
```

Программа для шифрования

Используя разработанные ранее модули и шаблон графического интерфейса, оформим полностью программу для шифрования, чтобы ею было удобно пользоваться. Оставшаяся часть работы отнимет совсем немного времени, и вся программа поместится в листинге 7.5. Графический интерфейс показан на рис. 7.6.

Листинг 7.5. Программа для шифрования файлов

```
unit FCript;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, Borland.Vcl.StdCtrls, System.ComponentModel, FMyGUI,
  Borland.Vcl.ExtCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    MyGUI: TMyGUI;
    Timer1: TTimer;
```

```
    procedure Timer1Timer(Sender: TObject);
    procedure FormActivate(Sender: TObject);
  private
    { Private declarations }
    procedure pCript;
    procedure pPassword;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation
uses UCript;
{$R *.nfm}
var
  Cript: TCript;
  Password: string;
  isPassword: Boolean;

procedure TForm1.pCript;
begin
  if IsPassword then
    if MyGUI.InFileName<>nil then begin
      Cript := TCript.Create;
      Cript.Password := Password;
      if Cript.CriptFile(MyGUI.InFileName) then
        MyGUI.Memo1.Lines.Append('Задача выполнена.')
      else
        MyGUI.Memo1.Lines.Append(
          'Задача не выполнена. '+
          'Возможно, файл с выходным именем уже существует.');
      Cript.Free;
      MyGUI.InString := nil;
      isPassword := False;
      Password := nil;
    end else
      MyGUI.Memo1.Lines.Append('Файл не указан!')
  else begin
    MyGUI.Memo1.Lines.Append('Введи пароль!');
    MyGUI.Edit1.SetFocus;
  end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if MyGUI.Exec then begin
    MyGUI.Exec := False;
    pCript;
  end;
  if MyGUI.isString then begin
    MyGUI.isString := False;
    pPassword;
  end;
  if MyGUI.Help then begin
    MyGUI.Help := False;
    MyGUI.Memo1.Lines.Append('Кнопка не используется!');
  end;
```

```
  if MyGUI.About then begin
    MyGUI.About := False;
    MyGUI.Memo1.Lines.Append('Кнопка не используется!');
  end;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
  MyGUI.Memo1.Clear;
  MyGUI.Memo1.Lines.Append
    ('До начала шифрования/дешифрования '+
     'введите имя файла и пароль');
  MyGUI.Edit1.Clear;
  MyGUI.Edit1.SetFocus;
end;

procedure TForm1.pPassword;
begin
  Password := MyGUI.InString;
  if Length(Password)<=6 then begin
    MyGUI.Memo1.Lines.Append(
      'Пароль — не меньше 6 символов. Повтори.');
    MyGUI.InString := nil;
    MyGUI.Edit1.Clear;
  end else begin
    isPassword := True;
    MyGUI.Edit1.Clear;
    MyGUI.Memo1.Lines.Append(
      'Пароль правильный! Если входной файл указан, '+
      'щелкни на кнопке Выполнить.');
  end;
end;
end.
```

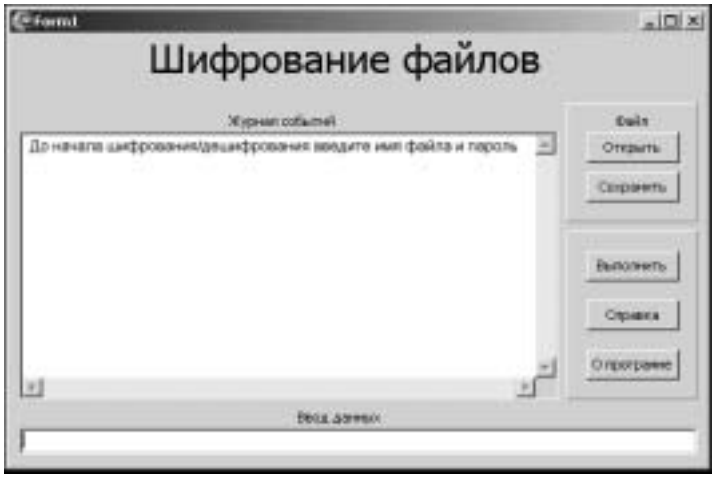

Рис. 7.6. Графический интерфейс программы для шифрования файлов

Здесь реализована функция ввода пароля, контроля длины пароля и выбора файла с выводом предупреждающих сообщений, если пароль короткий или не введено имя файла. При активизации формы задаются начальные значения (обработчик `FormActivate`).

Резюме

В данной главе рассмотрены основные методики и стили разработки программ и приведены направления, следуя которым, можно значительно повысить скорость разработки программ, а также способы создания независимых фрагментов программ, которые можно использовать неоднократно. Кроме того, рассмотрены основы технологии передачи сообщений и использование сообщений Windows.

Достаточное количество примеров и фрагментов работающих программ помогут вам освоить Delphi и внушат уверенность, что Delphi — не привилегия профессионалов и программирование на этом языке вполне доступно начинающим программистам.

Контрольные вопросы

1. Перечислите основные технологии программирования.
2. Какая технология программирования является наиболее прогрессивной?
3. Что такое класс?
4. Почему при написании больших программ необходимо использовать классы?
5. Что такое объект?
6. Сколько объектов можно создать на основе одного класса?
7. За счет чего можно повысить скорость кодирования?
8. Что такое сообщение?
9. Для чего нужны сообщения?
10. Когда создается сообщение?
11. Какие типы сообщений вы знаете?
12. Что такое обработка сообщения?
13. Напишите процедуру обработчика сообщения.