14. Компонентная разработка приложений. Введение в компонентные модели

Как уже было отмечено ранее, одной из основных тенденций последних 20 лет является использование инструментальных средств быстрой разработки приложений (RAD – Rapid Application Development). Все эти пакеты (Delphi, C++ Builder, MS Visual Studio, Java NetBeans и т.д.) основаны на использовании так называемых **компонентов** – объектов со специальными свойствами и специальным поведением. Такие объекты-компоненты имеют следующие особенности:

- каждый объект-компонент решает свою логически законченную задачу
- компоненты взаимодействуют друг с другом по четко определенным правилам
- компоненты могут встраиваться в состав RAD-инструментов и многократно использоваться в рамках технологии визуальной разработки приложений

Ясно, что понятие «объект» более широкое, чем понятие «компонент»: любой компонент является объектом, но не любой объект является компонентным. Наличие большого числа стандартных компонентов является основой современных средств разработки и позволяет существенно уменьшить объем «ручного» кодирования и в итоге сократить сроки разработки.

Одним из первых примеров эффективности данного подхода является компонентная разработка оконного пользовательского интерфейса программы. Для реализации этого интерфейса разработаны десятки стандартных компонентов, от очень простых до достаточно сложных. Компоненты могут выбираться из библиотеки, визуально настраиваться и при необходимости – допрограммироваться.

Например, стандартный компонент «Управляющая кнопка» (Button по-ихнему:)) имеет следующий предопределенный функционал:

- умеет отображать себя на форме в заданном месте, с заданными размерами и заданным стилем
- умеет реагировать на выбор пользователем с помощью мыши, клавиатуры, а в современных реализациях – просто пальцем

Разработчику остается лишь написать код, который будет выполняться в ответ на выбор кнопки – все остальное уже реализовано!

Еще пример – стандартный компонент «Список строк» (ListBox):

- способен хранить включенные в него строки (контейнер, однако:))
- умеет добавлять и удалять строки
- умеет отображать свои строки на форме
- умеет сортировать строки
- умеет реагировать на выбор строк пользователем

Другими часто используемыми компонентами являются поля ввода, таблицы, переключатели, деревья, текстовые надписи и др.

Конечно, разработкой интерфейса использование компонентов не ограничивается. Еще одним чрезвычайно важным и мощным применением компонентного подхода является разработка приложений, использующих базы данных. Подключение к базам разных типов, доступ к хранящимся в них данным, отображение этих данных – все реализовано в стандартных компонентах. Это позволяет создавать простые приложения вообще без написания программного кода.

Ясно, что компоненты-объекты являются экземплярами соответствующих классов, в которых описана вся функциональность таких объектов. Компонентные классы являются частным случаем обычных классов и описываются с помощью специальных правил. Набор таких правил иногда называют **компонентной моделью**. Не претендуя на полноту изложения, выделим лишь два основных момента, характерных для всех основных компонентных моделей:

- **обязательное** закрытие внутренних данных объекта-компонента и организация **контролируемого доступа** к ним с помощью специальных элементов – так называемых **свойств**
- наделение компонентных объектов **способностью реагирования на события**

Эти общие моменты в разных компонентных моделях реализованы по-разному. Далее для сравнения приводится краткое описание трех основных компонентных моделей. Начинается описание с компонентной модели пакета Delphi как одной из первых (отметим, что аналогичная модель реализована в пакете C++ Builder), а затем уже описываются более поздние модели JavaBeans и .NET Framework.

В компонентной модели Delphi доступ к внутренним закрытым полям данных реализуется с помощью специальных элементов – свойств (**property**). Общая схема использования свойств:

- объявить внутреннее закрытое поле данных;
- объявить и реализовать один или два метода доступа (обычно – защищенные);
- объявить с помощью директивы **property** открытое свойство с указанием (с помощью директив **read** и **write**) используемых методов доступа.

Схематично это реализуется следующим программным кодом:

```
TSomeClass = class
  private  FSomeField : <тип поля>;              // закрытое поле данных
  protected  function  GetSomeField : <тип поля>;     // методы доступа
  protected  procedure  SetSomeField (aField : <тип поля>);
  public  property  SomeField : <тип поля>        // свойство как таковое
                    read  GetSomeField   write  SetSomeField;
end;
```

Простейшая реализация методов доступа:

```
function  TSomeClass.GetSomeField : <тип>;
begin
  result := FSomeField;
end;
procedure  TSomeClass.SetSomeField (aField : <тип>);
begin
  if  aField <> FSomeField  then  FSomeField := aField;
end;
```

Синтаксически свойства в программе используются следующим образом:

- как обычно, объявляется и создается объектная переменная

  ```
  var  SomeObject : TSomeClass;
  SomeObject :=  TSomeClass.Create;
  ```

- свойству присваивается некоторое значение в соответствии с его типом

  ```
  SomeObject.SomeField := <значение>;
  ```

- значение свойства присваивается некоторой переменной

  ```
  <переменная> := SomeObject.SomeField;
  ```

При этом последние два присваивания переводятся компилятором в вызовы соответствующих методов доступа: в первом случае для присваивания значения вызывается Set-метод, во втором для получения значения свойства – Get-метод.

В простейшем случае допускается отсутствие методов доступа с указанием в описании свойства имени соответствующего закрытого поля:

```
TSomeClass = class
  private  FSomeField : <тип поля>;
  public  property  SomeField : <тип поля>
                    read  FSomeField   write  FSomeField;
end;
```

Вместо директивы **public** при описании свойств часто используется директива **published,** которая определяет так называемые «**публикуемые**» свойства. Особенность таких свойств состоит в том, что в процессе создания

приложения они видны в окне Инспектора Объектов (Object Inspector) и могут изменяться разработчиком.

Второй важный (и более сложный) аспект создания компонентных классов связан с тем, что объекты-компоненты должны поддерживать механизм обработки событий, на котором основаны все современные универсальные операционные системы. Для этого с каждым объектом связывается набор характерных для него событий и предоставляется возможность для каждого такого события реализовать необходимый программный код обработки.

Технически для этого в состав компонентного класса вводятся специальные закрытые поля, в которых могут храниться **адреса подпрограмм**, отвечающих за обработку соответствующих событий. Такие поля принципиально отличаются от обычных полей и поэтому объявляются с помощью специального так называемого **процедурного типа.** Каждый процедурный тип имеет свое имя, описывается специальным образом и фактически определяет группу подпрограмм с **одинаковым набором параметров.** Например:

type TSomeProcType = **procedure**(параметры) **of object;**

Переменной типа TSomeProcType в качестве значения можно присвоить имя любого метода, у которого **набор, тип и порядок параметров соответствует описанному**. Кратко можно сказать, что в объектной модели Delphi **событие** является **свойством процедурного типа.**

В базовой библиотеке пакета Delphi объявлено несколько стандартных процедурных типов, используемых для создания обработчиков событий. Простейшим типом является тип **TNotifyEvent,** который вводит класс методов-обработчиков, имеющих только один параметр **Sender** типа **TObject,** через который передается указатель на объект-источник события:

TNotifyEvent = **procedure** (Sender : TObject) **of object;**

Этот тип используется для обработки так называемых **уведомляющих** событий, обычно возникающих при выборе управляющих кнопок формы, пунктов меню и т.д. Более сложные процедурные типы предусмотрены для обработки событий от мыши и клавиатуры:

```
TKeyEvent = procedure (Sender : TObject;  var  Key : word;
                                    Shift : TShiftState)  of  object;
TMouseEvent = procedure (Sender : TObject;  Button : TMouseButton;
                                    Shift : TShiftState;  x,y : integer)  of  object;
```

Как правило, стандартных процедурных типов и событий вполне хватает для решения большинства задач. При необходимости легко можно ввести и нестандартные события. Для этого надо:

- объявить новый процедурный тип:

  TNewEventProc = **procedure** (параметры) **of object;**
- ввести новый класс с объявлением в нем свойств-событий с использованием упрощенной схемы описания свойств (без объявления методов доступа):

```
TNewClass = class (КлассРодитель)
    private  FOnNewEvent : TNewEventProc; // поле процедурного типа
    published  property  OnNewEvent : TNewEventProc
                                read  FOnNewEvent  write  FOnNewEvent;
```

- объявить и реализовать обработчик события OnNewEvent в полном соответствии с процедурным типом TNewEventProc

```
procedure  NewEventHandler ( параметры как у типа TNewEventProc )
```

- ввести переменную-объект класса TNewClass (например – NewObj) и присвоить свойству-событию имя метода-обработчика этого события (например – имя NewEventHandler)

```
NewObj. OnNewEvent := NewEventHandler;
```

Еще один важный момент в реализации компонентной модели связан с ее поддержкой в рамках стандартной библиотеки классов. Для использования

компонентов как строительных блоков при создании приложений, в библиотеке VCL предусмотрено большое число классов. Несколько из них, расположенных на верхних уровнях иерархии, обеспечивают наиболее общие свойства всех компонентов. К этим классам прежде всего относятся классы **TPersistent** и **TComponent.**

Класс **TPersistent** является непосредственным потомком базового класса **TObject**. Важность данного класса определяется тем, что он вводит механизм **потоковости** или **сериализуемости (serializable),** т.е. все его потомки (в том числе абсолютно любые компоненты) могут **сохранять** свои свойства во внешних файлах-потоках и **восстанавливать** свойства из файлов. Для этого в классе вводится метод **Assign,** с помощью которого поля одного объекта могут быть присвоены полям другого объекта.

Класс **TComponent,** как следует из его имени, является базовым классом для всех компонентов. Он происходит непосредственно от класса **TPersistent** и вводит несколько наиболее общих свойств и методов, наследуемых всеми компонентами. Наиболее важными из них являются те, что отвечают за поддержку **механизма владения**: каждый компонент имеет своего владельца и сам может владеть другими компонентами. Механизм владения очень важен для создания и уничтожения объектов-компонентов с точки зрения выделения и освобождении динамической памяти. Например, форма как визуальная единица приложения владеет всеми размещенными на ней компонентами, т.е. является специализированным контейнером. При закрытии формы это позволяет легко и корректно освободить память, выделенную для всех компонентов формы.

Механизм владения поддерживается в классе **TComponent** с помощью трех основных свойств:

- свойство **Components** представляет собой массив указателей на подчиненные компоненты;
- свойство **ComponentCount** определяет текущее число подчиненных компонентов;

- свойство **Owner** содержит указатель на хозяина компонента.

Методы класса **TComponent** обеспечивают стандартные операции добавления, удаления и поиска в контейнере.

Использование компонентов при визуальном создании приложений заключается в следующем. Каждый компонент **регистрируется** в Delphi и становится доступным для выбора в палитре компонентов. Этот выбор с помещением значка компонента на форму приводит к созданию соответствующего объекта, после чего в Инспекторе Объектов становятся доступными для редактирования все опубликованные свойства компонента, включая список обработчиков событий.

Всех потомков класса **TComponent** можно разделить на две группы – **визуальные** и **невизуальные** компоненты. Визуальные компоненты имеют видимое графическое представление на форме как в процессе разработки приложения, так и при его выполнении. Базовым классом визуальных компонентов является потомок класса **TComponent** – класс **TControl.** В нем вводятся такие общие свойства, как положение компонента, его текущее состояние и стиль, а также определяются все основные обработчики событий. Основными невизуальными компонентами являются компоненты доступа к базам данных, компоненты-диалоги, компонент-таймер и другие.

Далее кратко рассмотрим компонентную модель языка Java. Компоненты многократного использования объединены в языке Java общим названием **Beans.** Идеологически технология **Beans** близка компонентной технологии Borland Delphi. Пожалуй, наибольшее отличие этих технологий касается механизма обработки событий.

Для описания компонентных классов разработаны простые правила, с помощью которых определяются свойства (в том числе и видимые разработчиком приложения), методы и выполняется обработка событий. Что касается свойств, то общее правило их объявления и использования аналогично свойствам компонентной модели Delphi: свойство – это

внутреннее закрытое поле, доступ к которому реализуется только через обращение к соответствующим методам доступа. При этом методы доступа **обязательно** именуются с помощью префиксов **set** и **get,** за которыми идет имя свойства: setName(параметр) и getName( ). Как обычно, set-метод принимает один входной параметр типа внутреннего поля, а get-метод возвращает значение поля. Фрагмент описания компонентного класса:

```
class MyBeansClass
{ private int Field1;                          // закрытые поля
  private String Field2;
  public int getField1( ) {return Field1;}  // методы доступа
  public void setField1(int aField1) { Field1 = aField1;}
  public String getField2( ) {return Field2;}  //  методы доступа
  public void setField2(String aField2) { Field2 = aField2;};
  public void SomeMethod(параметры) { реализация };  // просто метод
 };
```

Отдельно надо хотя бы кратко остановиться на механизме обработки событий. Поскольку язык Java не поддерживает указатели на функции, разработчикам языка пришлось использовать другие подходы. Как и все остальное в языке Java, обработка событий выполняется на **объектном** уровне с помощью специальных классов, в том числе и интерфейсных (это еще раз подтверждает значение интерфейсов для технологии Java!).

Любое событие генерируется объектом-источником при изменении его внутреннего состояния. Все основные события **классифицированы** и им поставлены в соответствие некоторые классы. Родоначальником иерархии событий является класс **EventObject,** от которого порождены следующие основные классы событий:

- **ActionEvent –** обработка нажатий кнопок или выбор пунктов меню.
- **ComponentEvent –** общие события при манипуляциях с компонентом.
- **ContainerEvent –** события при добавлении/удалении компонента из контейнера.

- **KeyEvent –** события ввода с клавиатуры.
- **MouseEvent –** «мышиные» события.
- **WindowEvent –** оконные события.
- **ItemEvent –** выбор элементов в списках.
- **TextEvent –** любые изменения в текстовых полях.

Обработку того или иного события выполняет специальный **объект-слушатель (listener).** Этот объект является экземпляром класса, реализующего некоторый интерфейс, характерный для данного типа события. Для каждого класса событий существует свой **интерфейс слушателя,** определяющий один или несколько специфических методов обработки данного события. В качестве примера приведем несколько наиболее важных интерфейсов слушателя и определяемых ими методов:

- **KeyListener**: keyPressed, keyReleased, keyTyped
- **MouseListener**: mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased
- **MouseMotionListener**: mouseMoved, mouseDragged
- **ItemListener**: itemStateChanged
- **WindowListener**: windowOpened, windowClosing, windowActivated
- **ActionListener**: actionPerformed

Для обработки события надо создать объект-слушатель и **зарегистрировать** его в компоненте, который генерирует данное событие. Для регистрации слушателей используются специальные методы **add***Listener**() и **remove***Listener**(), где звездочки *** заменяют тип события. Например: addActionListener, addKeyListener, addMouseListener, addItemListener и т.д. Компонент может зарегистрировать **несколько** слушателей-обработчиков разных событий. Например, если ранее описанный класс MyBeansClass должен обрабатывать события от мыши и клавиатуры, он должен зарегистрировать соответствующих слушателей, для чего в описание класса надо добавить следующие методы:

public void addKeyListener(KeyListener kl) { . . . };

public void removeKeyListener(KeyListener kl) { . . . };

public void addMouseListener(MouseListener ml) { . . . };

public void removeMouseListener(MouseListener ml) { . . . };

Информация о всех свойствах компонентного класса и зарегистрированных событиях сохраняется в файле и с помощью механизма **отражения** доступна в процессе визуального создания программы. Здесь также используется такое важное свойство Bean-компонентов, как **сохраняемость (persistence),** т.е. способность компонентов записывать себя во внешние файлы с восстановлением своей структуры в более позднее время. Это свойство обеспечивается механизмом **сериализации,** который является стандартным для всех объектов Java, реализующих интерфейс **Serializable.** Этот механизм превращает объект в поток байтов, помещаемый во внешний файл. Очень важно, что механизм сериализации позволяет сохранять и восстанавливать объектную структуру **любой сложности,** т.е. со всеми внутренними объектными свойствами.

В заключение рассмотрим компонентную модель платформы .NET. Эта платформа в основном предназначена для разработки приложений для систем семейства Windows. Основу этой технологии составляет библиотека классов **Framework Class Library**, которая в настоящее время включает в себя несколько тысяч классов, с помощью которых можно создавать практически любые типы Windows-приложений.

Одной из особенностей процесса разработки .NET-приложений является **языковая независимость** (относительная, конечно): исходный текст можно написать на любом языке, поддерживающем данную технологию (C#, J#, VB, C++), после чего соответствующий компилятор переводит этот текст в универсальное **промежуточное представление**. Это представление описывается с помощью специального языка **IL (Intermediate Language),** немного похожего на язык ассемблера. После этого отдельные IL-модули могут объединяться в так называемую **сборку (assembly)**, которая

представляет собой единицу распространения программы. Интересно, что сборка помимо кода на IL-языке содержит **метаданные** с полным описанием используемых в программе типов и классов. При запуске программы на выполнение IL-код переводится на машинный язык конкретного процессора и выполняется. Этими процессами управляет специальное средство .NET-технологии – так называемая **Common Language Runtime** (CLR, общеязыковая среда выполнения).

Компонентная модель платформы .NET аналогична рассмотренным ранее моделям языков Java и Delphi. При описании класса вводятся закрытые поля данных и открытые свойства для определения методов доступа к этим полям. Описание свойства должно включать объявление одного или двух методов доступа с обязательными именами **get** и **set**.

Например:

```
public class MyClass
{ private int myfield;   // закрытое поле
   public int  MyField  // соответствующее свойство
   {  get  { return  myfield; }  //  метод доступа
   {  set   {  myfield = value;  }  // метод доступа
  }
. . .  другие поля и их свойства . . .
}
```

Аналогично другим языкам, механизм свойств позволяет организовать более строгий и контролируемый доступ к значениям внутренних свойств объектов.

Как уже отмечалось выше, компонент как строительный блок приложения должен обеспечивать реакцию на определенные **события**. Обработка событий в .NET-приложениях основана на использовании указателей на функции-обработчики, но по сравнению с пакетом Delphi эти указатели реализованы более надежным способом – с помощью специальных объектов, называемых **делегатами** (**delegate**). Объект-делегат может хранить

указатель на функцию только **конкретного** заранее заданного типа, что позволяет организовать **контролируемый** вызов методов и тем самым повышает надежность приложения.

Базовым классом для реализации делегатов является класс **MulticastDelegate**, в котором вводится возможность хранения **любого** числа указателей на функции и все необходимые для этого методы. На основе этого класса создаются пользовательские дочерние классы, объекты которых и выполняют всю работу по обращению к необходимым функциям. Поскольку все вызовы таких функций выполняются **динамически** при работе приложения, то без использования среды выполнения CLR обойтись невозможно.

Класс-делегат описывается **неявно** с помощью служебных слов, например (в языке C#) – с помощью директивы **delegate**. Синтаксис описания делегатов отличается от привычного описания классов, поскольку часть работы возложена на компилятор. При описании класса-делегата указывается **прототип метода**, на который будут ссылаться объекты-экземпляры этого класса. Например, следующий делегат предназначен для вызова **любых** методов, которые имеют **ровно один параметр строкового типа** и **не** возвращают результат (**void**):

```
delegate void MyDelegat (string st);
```

После этого надо создать объект-делегат с помощью неявного конструктора, передав ему в качестве параметра имя метода, на который должен ссылаться объект-делегат:

```
MyDelegat myDel = new MyDelegat (myObj.MyMetod);
```

Здесь myObj – объект класса, в котором реализован пользовательский метод с именем MyMetod. Для нашего примера этот метод **обязательно** должен иметь ровно один параметр строкового типа! Если попытаться передать конструктору метод с другой сигнатурой, компилятор выдаст сообщение об ошибке.

Пусть для наглядности этот метод MyMetod просто лишь выводит на экран текстовую строку, передаваемую ему через входной параметр st. Тогда для вызова этого метода с помощью делегата достаточно оформить следующую простую конструкцию:

myDel (“ Hello from delegate”);

Конечно, в данном примере использование делегатов кажется надуманным, поскольку метод MyMetod можно вызвать непосредственно. Однако для реализации **обратных вызовов**, возникающих при выполнении программы этот механизм весьма удобен. Именно по этой причине он и используется при обработке событий. При этом важным моментом в механизме делегатов становится возможность объединения объектов-делегатов в списки. Для добавления и удаления элементов в эти списки можно использовать методы **Combine** и **Remove** базового класса **MulticastDelegate**. Для удобства использования этих методов в языке C# переопределены операторы += и -= . Вместо них компилятор подставляет вызовы методов **Combine** и **Remove**. Например, если вместе с методом MyMetod объявить другой метод MyMetod2 с такими же параметрами (в данном случае – с одним параметром строкового типа), то его можно добавить в цепочку делегатов с помощью следующей конструкции:

myDel += myObj.MyMetod2;

После этого указанное выше обращение к делегату приведет к вызову уже **двух** разных методов.

Обработка событий встраивается в компоненты следующим образом. Прежде всего, в классе компонента объявляется **закрытое поле-делегат,** через которое должна устанавливаться связь с объектом-делегатом. Потом в классе вводится специальное **открытое свойство** с двумя методами, которые пользователи компонента могут использовать для добавления и удаления событий в списке вызова делегата. Поскольку эти операции являются достаточно стандартными, в язык C# введена специальная директива **event**, с

помощью которой в классе компонента и вводятся необходимые составляющие:

```
class MyComponent: ParentComponent
{ public event MyDelegat MyEvent;
  . . . . . };
```

После этого пользователи компонента должны создать его экземпляры и добавить обработчики события MyEvent следующим образом:

```
MyComponent myComp = new MyComponent ( );
myComp += new MyDelegat (myComp.MyMetod);
```

В заключение отметим, что для создания полноценных компонентов необходимо придерживаться ряда простых рекомендаций по оформлению делегатов и методов-обработчиков событий.