Сериализация объектов стандартными средствами Delphi

Автор: Иван Пономарёв
ЛАНИТ
Источник: RSDN Magazine #6-2003

Опубликовано: *01.08.2004*
Исправлено: *13.03.2005*
Версия текста: *1.0*

Часть I. Введение

Что это такое и зачем это нужно

При создании многих приложений в Delphi возникает задача разработки системы сохранения в файл документа или проекта, с которым работает программа. Довольно часто при этом разработчики выбирают непростой путь создания собственных форматов, что, хотя иногда и бывает оправдано, но чаще приводит к возникновению больших и порою критических для проекта трудностей.

Есть две вещи, которые требуются от хорошего механизма сохранения, и которые непросто реализовать самому. Во-первых, нужно, чтобы процедуры записи и чтения информации были достаточно независимыми от реализации самого приложения. Если это не так, то любое изменение в логике программы, требующее новой структуры сохраняемого файла, должно сопровождаться синхронными изменениями в процедурах записи и чтения. Это рано или поздно приведёт к тому, что поддерживать такой проект станет слишком сложно. Во-вторых, в формате записываемого файла должна быть заложена совместимость «снизу вверх», чтобы пользователь обновлённой версии программы мог без проблем открывать файлы, созданные в старой версии, а от программиста для достижения такой возможности требовалось бы минимум усилий.

К счастью, универсальное решение указанных выше проблем даёт стандартная библиотека Delphi. Реализованный в Delphi механизм сериализации (сериализация – это автоматическая выгрузка в файл, BLOB-поле таблицы или другой поток находящейся в памяти системы

взаимосвязанных объектов) достаточно развит, соответствует приведённым выше требованиям и предоставляет много вариантов быстрого создания сколь угодно сложных сохраняемых систем.

К сожалению, среди работ, которые можно видеть в книжных магазинах и российском Интернете, не попадаются исследования, специально посвящённые использованию этого механизма. Среди того, что можно найти в Интернете – ряд статей Андрея Чудина, включая опубликованную на сайте «Королевство Delphi» (www.delphikingdom.com) статью «XML-сериализация объекта Delphi», в которой автор описывает собственное решение задачи и предлагает воспользоваться созданным им компонентом. Однако в своих статьях Андрей Чудин не исследует до конца стандартный механизм сериализации.

Цель настоящей статьи – полностью разобрать механизм сериализации объектов стандартными средствами Delphi. Статья написана на основе практического опыта создания приложений, использующих сериализацию для сохранения в файл своих документов, и содержит освещение предметов, нужных для написания этих программ. Надеюсь, что по прочтении этой статьи вы сможете с лёгкостью создавать сохраняемые системы, решающие ваши задачи.

Что потребуется от вас

Поскольку сериализация сохраняет в поток состояние объектов, необходимо, чтобы бизнес-логика приложения была разработана в виде системы взаимосвязанных классов. Насколько мне известно, у многих Delphi-программистов бывает не так. В отличие от других систем разработки, Delphi не подталкивает разработчика с самого начала к тому, чтобы отделять логику от интерфейса и писать, помимо классов форм, классы для каждой из задействованных в логике приложения сущностей. Тем не менее, такой подход наиболее оправдан, и сама среда Delphi позволяет создавать проекты с очень интересной архитектурой на основе шаблонов, или паттернов проектирования, о которых в последнее время появилось много хороших статей (например, «Практика применения паттернов проектирования» в RSDN #3 за 2002 год).

Основная идея

Воспользоваться стандартным механизмом сериализации объектов в Delphi довольно просто. Надо лишь:

1. Унаследовать класс одного из сериализуемых объектов, который в дальнейшем будет называться «корневым», от TComponent.
2. Создать поток (экземпляр класса-наследника TStream) и вызвать его метод WriteComponent(Instance: TComponent), указав в качестве аргумента ссылку на корневой объект.

Собственно говоря, всё. Как известно, от TStream в Delphi наследуются разные типы потоков, предназначенные для работы с памятью, с файлом и с BLOB-полем таблицы. Все эти типы наследуют метод WriteComponent от TStream. Прочитать объект из потока можно будет с помощью метода TStream.ReadComponent.

WriteComponent сериализует объекты в двоичном формате. Однако в отладочных целях гораздо удобнее иметь дело с текстами, которые можно читать и править в Блокноте. Для перевода двоичного формата в удобочитаемый формат существует процедура ObjectBinaryToText из модуля Classes. Она принимает два параметра, которые должны содержать, соответственно, ссылку на входной поток с двоичными данными и ссылку на выходной поток, в который будет записываться текст. Процедура записи произвольной системы объектов в текстовый файл, таким образом, будет выглядеть так:

Процедура записи

```
procedure SaveToFile(RootObject: TComponent; const FileName: TFileName);
var
  FileStream: TFileStream;
  MemStream: TMemoryStream;
begin
  FileStream := TFileStream.Create(FileName, fmCreate);
  MemStream := TMemoryStream.Create;
  try
    MemStream.WriteComponent(RootObject);
    MemStream.Position := 0;
    ObjectBinaryToText(MemStream, FileStream);
```

```
  finally
    MemStream.Free;
    FileStream.Free;
  end;
end;
```

Что получится, если применить эту процедуру для записи какого-либо объекта-наследника TComponent (например, формы приложения) в текстовый файл? Если взглянуть на то, что получилось, можно увидеть нечто знакомое – а именно то, что мы видим, когда открываем в Блокноте .dfm-файл Delphi-проекта, или, щёлкнув в дизайнере форм правой кнопкой мыши на форме, выбираем команду "View as Text". Ничего удивительного: механизм сериализации Delphi был задуман, прежде всего, для того, чтобы запоминать раскладку и взаимные отношения визуальных компонентов в .dfm-файлах. Именно поэтому создание сложных настраиваемых компонентов – ещё одна область, в которой важно понимание сериализации.

Теперь, чтобы перевести dfm-формат в двоичный поток, годный для чтения с помощью метода ReadComponent, надо использовать процедуру ObjectTextToBinary. Прочитать объекты из текстового файла можно с помощью такой процедуры:

Процедура чтения

```
procedure LoadFromFile(RootObject: TComponent; const FileName: TFileName);
var
  FileStream: TFileStream;
  MemStream: TMemoryStream;
begin
  FileStream := TFileStream.Create(FileName, 0);
  MemStream := TMemoryStream.Create;
  try
    ObjectTextToBinary(FileStream, MemStream);
    MemStream.Position := 0;
    MemStream.ReadComponent(RootObject);
  finally
    MemStream.Free;
    FileStream.Free;
  end;
end;
```

Итак, процедуры записи и чтения произвольных систем объектов готовы, и, в общем-то, на этом месте разговор можно было бы завершить. Но дело ещё и в том, что не каждый класс-наследник TComponent будет «по умолчанию» отдавать потоку свою информацию. Чтобы добиться сохранения того, что нам нужно, класс необходимо «калибровать» с помощью доработок, о которых, собственно, и пойдёт речь ниже. Такая «калибровка» – это итеративный процесс. Поэтому важно иметь надёжный метод тестирования сериализации. Помимо традиционной отладки, таким методом могут быть модульные тесты, которое вообще очень эффективны при разработке систем классов бизнес-логики. О ставшей стандартом де-факто системе модульного тестирования DUnit можно узнать на сайте http://dunit.sf.net.

Ещё раз о TComponent

Прежде чем продолжить разговор о сериализации, необходимо сообщить ещё некоторые сведения о классе TComponent. Этот класс, хотя и довольно близок в иерархии VCL к простому объекту (между ним и TObject находится лишь TPersistent), уже обладает столь большой функциональностью, что хватило бы на отдельную статью. Мы рассмотрим лишь самую важную для нас часть: то, что необходимо знать для использования механизма сериализации.

> **ПРИМЕЧАНИЕ**
> Для удобства в дальнейшем мы будем называть «компонентами» любые объекты, являющиеся экземплярами TComponent или классов, наследуемых от TComponent.

Я хотел бы коснуться двух связанных с классом TComponent концепций: владения и уникальных

имён. Если вам всё это уже хорошо известно, можете пропустить следующие два подраздела.

Владение: свойство Owner

Особенности компонентов начинаются с конструктора, который требует указать в качестве аргумента другой компонент:

```
constructor Create(AOwner: TComponent); virtual;
```

Этот другой компонент называется владельцем исходного и в дальнейшем может быть доступен через read-only свойство Owner: TComponent. Впрочем, nil в качестве владельца компонента тоже допустим. Каждый компонент может получить доступ к объектам, которыми он владеет, через свойства:

```
property Components[Index: Integer]: TComponent;
property ComponentCount: Integer;
```

Эти свойства содержат соответственно массив компонентов и его размер. И хотя Owner – это read-only свойство, можно сменить владельца компонента. Для этого надо вызвать public-метод компонента, который нужно сделать владельцем:

```
procedure InsertComponent(AComponent: TComponent);
```

Впрочем, не любой компонент может быть владельцем любого другого. В частности, не пытайтесь с помощью InsertComponent сконструировать «замкнутый круг» (A = B.Owner, B = A.Owner) – вы получите Stack Overflow. Таким образом, компоненты в оперативной памяти могут объединяться лишь в «деревья» – структуры, не содержащие циклов и петель.

Зачем нужно владение? Дело в том, что оно предоставляет удобный механизм автоматического высвобождения памяти. Деструктор TComponent устроен таким образом, что он вызывает деструкторы всех компонентов, которыми данный компонент владеет, а те – вызывают деструкторы компонентов, которыми владеют они, и так далее. Сделав A владельцем B, мы можем более не задумываться о вызове B.Free в надлежащем месте кода, поскольку когда разрушится A, разрушится и B. Именно таким образом при вызове деструктора формы высвобождается память, занятая всеми компонентами, находящимися на данной форме.

Класс TComponent предоставляет protected-метод Notification, который можно переопределить в наследнике. Это дает возможность предпринять какие-то действия в момент, когда один компонент становится владельцем другого или перестает быть таковым. Кроме того, с помощью public-метода FreeNotification(AComponent:TComponent) можно «подписать» компонент на «извещение о разрушении» другого компонента. При прямом вызове деструктора некоторого компонента у его владельца, а также у всех тех, которые «подписаны на извещение» о его разрушении, вызывается метод Notification с соответствующими параметрами. Всё это нужно для сохранения целостности возможных перекрёстных ссылок между компонентами.

В завершение надо заметить, что, несмотря на то, что концепция «владения» поддерживает многоуровневую иерархию, на практике почти никогда не создаются системы с большим числом уровней. Чаще всего один компонент владеет многими другими, которые, в свою очередь, не владеют ничем. Так, форма владеет всеми объектами-наследниками TComponent, на ней расположенными, хотя на первый взгляд может показаться, что это не так. Например: хотя любой TAction «входит» в TActionList, если вы проверите значение свойства Action.Owner во время выполнения программы, то обнаружите, что это – форма (которая и является владельцем и TAction, и TActionList). Причины этого будут указаны ниже.

Уникальные имена: свойство Name

Другим важнейшим свойством всех компонентов является Name:

```
property Name: TComponentName;
```

где тип TComponentName – псевдоним обычной строки. У этого свойства есть две особенности. Во-первых, строка, которой приравнивается Name, должна содержать *только* большие и малые буквы латинского алфавита, цифры и знаки подчёркивания, и не должна начинаться с цифры. Попытка присвоить этому свойству что-то другое приведёт к ошибке во время выполнения. И, во-вторых, у нескольких компонентов с одинаковыми именами *никогда* не может быть общего владельца, если только их имена – не пустые строки. Ошибка возникнет при любой попытке задать какому-либо компоненту имя, неуникальное среди компонентов, имеющих того же владельца, или поменять владельца компонента таким образом, что в новой «компании» имя компонента будет неуникальным.

Если вспомнить, что имени компонента, положенного в визуальном редакторе на форму, соответствует имя переменной в классе формы, то эти ограничения станут вполне очевидны. Однако не вполне очевидно, что то, что верно в design-time, верно и в run-time, когда наследники класса TComponent могут не являться компонентами в обычном смысле этого слова, и нет никакой необходимости создавать класс, в котором имена переменных совпадали бы со свойствами Name создаваемых компонентов.

В design-time IDE само создаёт уникальное имя для всякого вновь добавляемого на форму компонента. Если же создается система, в которой компоненты будут создаваться в run-time, то об уникальных именах придется позаботиться разработчику. Получать уникальные идентификаторы можно, например, на основании счётчика, но есть и другие способы – например, на основе GUID'ов, выдаваемых функцией CreateGUID из модуля SysUtils.

В пределах общего владельца свойство Name служит уникальным идентификатором объекта. Поэтому по значению этого свойства можно получить ссылку на этот объект. Для этого существует метод

```
function FindComponent(const AName: string): TComponent;
```

возвращающий компонент по его имени (если же компонент с указанным именем не найден, возвращается nil). Как можно увидеть из реализации метода FindComponent, для поиска используется метод последовательного перебора, так что при владении большим количеством компонентов рассчитывать на хорошую скорость не приходится.

Свойство Name компонента столь важно, что при сериализации, как можно видеть в dfm-файлах, оно не просто перечисляется среди других свойств между словами object и end – оно стоит в заголовке объекта, перед его типом. Например, для следующего экземпляра TButton свойство Name = 'Button1':

```
object Button1: TButton
  Left = 136
  Top = 80
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

Name стоит на особой позиции, и это – не случайно.

Часть II. Основы

Основное правило сериализации

Теперь перечислим, что же на самом деле сохраняется в поток при вызове метода TStream.WriteComponent. То, что сейчас будет оглашено, я бы назвал основным правилом сериализации, т. к. именно это всё время надо иметь в виду при написании систем сериализуемых классов. Следует запомнить следующее: *при сериализации объекта в поток сохраняется и может быть восстановлена та и только та информация, которая либо содержится в read/write published-свойствах объекта, либо способ записи и чтения которой определён в псевдосвойствах.*

О псевдосвойствах, или "fake properties", как о них говорится в справочной системе VCL, речь

пойдёт ниже, а сейчас хотелось бы подробно остановиться на первом варианте.

В требовании «read/write published свойства» каждое слово является значимым. Если мы обходимся без псевдосвойств, вся информация, однозначно определяющая состояние объекта, *должна* быть вынесена в эти свойства, поскольку при сериализации НЕ сохраняется:

1. Ничего из того, что объявлено за пределами published секции.
2. Значения каких-либо полей (переменных) объекта, даже если эти переменные вынесены в published секцию.
3. Read-only свойства.

Но и это ещё не всё. Даже если вся определяющая состояние объекта информация вынесена в read/write published свойства, это само по себе не гарантирует успешной сериализации, поскольку нужно позаботиться о том, чтобы эти свойства имели правильный тип.

Поэтому в следующем разделе мы последовательно пройдёмся по сериализуемым типам свойств.

Сериализуемые типы

Примитивные типы

Самый простой случай сериализуемого свойства – это свойство, имеющее примитивный тип. К этим типам относятся числа, строки, перечисления (enumerations), либо то, что сводится к вышеуказанному (например, даты). В dfm-формате их значения следуют через знак равенства от имени свойства. Например, в файле формы мы можем видеть строки, подобные этим:

```
object Form1: TForm1
  BorderStyle = bsDialog
  Caption = 'Form1'
  ClientHeight = 119
. . .
end
```

Часто среди возможных значений свойства есть такое, которое этим свойством принимается в большинстве случаев. Чтобы сократить объём записываемой на диск информации – и, как следствие, сэкономить время и ресурсы при чтении – Delphi даёт нам возможность настроить каждое Integer, Boolean или enumerated свойство таким образом, чтобы оно не записывалось в поток, когда его значение совпадает с заданным по умолчанию. Это делается с помощью ключевого свойства default:

```
type TMyEnumeration = (meOne, meTwo, meThree);
property MyProp: TMyEnumeration read FMyProp write MyProp default meOne;
```

Для приведённого примера свойство MyProp не будет записываться в поток в том случае, если для сериализуемого объекта его значение будет равно meOne. Когда таких свойств много, использование default может существенно сократить размер файла и сделать его гораздо изящнее на вид (в dfm-формате).

Используя слово default, важно не допустить одной ошибки. Нужно чётко осознать, что *слово default не имеет никакого отношения к инициализации экземпляров класса.* Если, предположим, для значения по умолчанию вы выбрали нечто, отличное от нуля или первого элемента enumerated-типа, то в конструкторе класса нужно сделать так, чтобы эти значения присваивались свойствам объекта на этапе инициализации. Иначе получится неприятная вещь: при сериализации объекта значение свойства не будет сохранено в поток. При десериализации оно, соответственно, не прочитается и будет иметь значение, полученное при инициализации (или нулевое значение).

Итак, мы увидели, как сериализуются примитивные типы. В большинстве случаев нас устроят и они, но что делать, если мы хотим сохранить, например, ссылку на другой объект? Хотя ссылка – это 32-битное значение, сохранять его в поток не имеет смысла, поскольку после восстановления объектов из потока, они, конечно, займут совершенно иные адреса в памяти. Кроме того, выше мы говорили только о корневом объекте и не рассматривали варианты, при которых он может «захватить» в процесс сериализации другие объекты.

Наследники TPersistent

Первый такой случай имеет место, когда свойство объекта содержит ссылку на объект, унаследованный от TPersistent, но не TComponent, входящий в композицию с сериализуемым.

> **ПРИМЕЧАНИЕ**
> Если я говорю «объект B входит в композицию с A», я имею в виду, что в A имеется ссылка на B, и что B создаётся и разрушается вместе с A.

Как и в случае с корневым объектом, при сериализации объектов-наследиков TPersistent в поток сохраняется и может быть восстановлена та и только та информация, которая либо содержится в read/write published-свойствах объекта, либо способ записи и чтения которой определён в псевдосвойствах.

Рассмотрим пример. Если мы исследуем реализацию класса TControl = class(TComponent), то увидим, что его свойство Font ссылается на переменную FFont:TFont. Как известно, TFont = class (TPersistent), а в конструкторе TControl мы имеем FFont := TFont.Create; в деструкторе же – FFont.Free.

Если посмотреть на текст, скажем, сериализованной формы (TForm унаследован от TControl), можно заметить, что свойству Font не присваивается значение нового объекта TFont. Вместо этого происходит инициализация уже имеющегося экземпляра:

```
object Form1: TForm1
. . .
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
. . .
end
```

Charset, Color, Height, Name – всё это read/write published-свойства класса TFont.

У этого способа, хоть он и даёт нам возможность «вовлечь» в сериализацию другой объект, есть серьёзное ограничение: так можно сериализовать лишь объекты, находящиеся с данным в отношении «один к одному». Если же корневой объект ссылается на переменное число объектов, этот способ непригоден.

Коллекции

В этом случае нам помогают коллекции. Как известно, в VCL есть два класса: TCollection и TCollectionItem (оба – наследники TPersistent), предназначенные служить в качестве базовых для наших собственных коллекций и их элементов. Заводя свою коллекцию, мы должны, как минимум, переопределить конструктор TCollection (так, чтобы он получал правильную ссылку на класс элементов коллекции), метод TCollection.Add и свойство TCollection.Items, чтобы они возвращали нам объекты нужного типа. Коллекции Delphi по сути своей предназначены для того, чтобы реализовывать свойства, хранящие какие-либо списки. А если свойство даёт нам ссылку на коллекцию, которая входит в композицию с сериализуемым объектом, то каждый из элементов коллекции будет сериализован, как наследник TPersistent, т. е. будут сохранены его read/write published- и псевдосвойства.

Примером published-свойства типа коллекции является свойство Columns класса TListView. Вот как может выглядеть фрагмент сериализованной коллекции:

```
object ListView1: TListView
. . .
  Columns = <
    item
      Caption = 'Column 1'
      ImageIndex = 2
      MinWidth = 10
    end
    item
```

```
      . . .
    end
    item
      . . .
    end>
. . .
end
```

Итак, если есть необходимость в связи «один-ко-многим», мы поступаем следующим образом: создаём свою коллекцию, делаем композицию этой коллекции с нужным классом и выносим её в published-свойство. А что, если нужна связь «многие-ко-многим?» Или «многие-к-одному» – если понадобится, скажем, сериализовать систему из трёх объектов, A, B и C, и при этом как A, так и B должны иметь ссылки на C?

Другие компоненты

Классический пример подобной системы – система Action'ов на форме. Один и тот же Action, несущий в своих свойствах информацию о себе, может быть назначен одновременно нескольким control-ам на форме (например, кнопке и пункту меню). В подобных ситуациях объект, на который могут ссылаться один, несколько или ни одного объекта, должен быть компонентом.

В отличие от двух предыдущих случаев, когда содержимое сериализуемого объекта прописывалось «внутрь» свойства, здесь компонент, на который ссылаются другие объекты, будет сериализован отдельно, независимо от всех. О том, как это устроить, мы узнаем чуть ниже, здесь же просто рассмотрим, как будет выглядеть результат в текстовом формате:

```
object Form1: TForm1
. . .
  object Button1: TButton
. . .
    Action = Action1
  end
  object Button2: TButton
. . .
    Action = Action2
  end
  object ActionList1: TActionList
. . .
    object Action1: TAction
      Caption = 'Action1'
    end
    object Action2: TAction
      Caption = 'Action2'
    end
  end
end
```

Вот где нам пригодилась уникальность значений свойства Name! Вспомним, что владельцем всех этих Button'ов и Action'ов является Form1. Уникальность имён в пределах общего владельца позволяет системе сериализации использовать их в качестве идентификаторов. После загрузки происходит полное восстановление ссылок по именам, и мы вновь получаем систему объектов с перекрёстными ссылками.

Есть одна тонкость, которую надо учитывать, используя этот сериализуемый тип. Корректное восстановление ссылок на основании имён произойдёт лишь если корневой объект является общим владельцем для компонентов, которые содержат ссылки, и компонентов, на которые они ссылаются. Именно из-за этого выше упоминалось о том, что на практике используется лишь самый простой вариант иерархии владения: корневой компонент владеет всем, что входит в сериализацию, остальные компоненты не владеют ничем. Механизм десериализации устроен таким образом, что вне зависимости от того, как было раньше, после загрузки из потока корневой компонент окажется владельцем всех других компонентов.

Ну и конечно, ещё в программе надо позаботиться о том, чтобы все сериализуемые компоненты имели непустые уникальные имена.

Об ответственности за сериализацию

Не пугайтесь названия этого раздела. Речь пойдёт всего лишь о том, как сделать так, чтобы компонент записывал вместе с собой в поток другие.

Было бы ошибкой предполагать, что компонент сериализует то, чем он владеет. Владение и ответственность за сериализацию – это разные отношения, и по умолчанию компонент не сериализует вместе с собой ничего. Впрочем, в VCL есть два класса-наследника TComponent, которые сериализуют именно то, чем владеют: это TForm и TDataModule, и если посмотреть их исходный код, можно увидеть, что в них переопределён protected-метод TComponent.GetChildren.

```
procedure GetChildren(Proc: TGetChildProc; Root: TComponent); dynamic;
```

где

```
type TGetChildProc = procedure (Child: TComponent) of object;
```

GetChildren – переопределяемый метод, в классе TComponent он пуст. Он автоматически вызывается в тот момент, когда система сериализации хочет опросить компонент о том, какие ещё компоненты нужно сохранять. В свою очередь, те компоненты тоже будут опрошены, и т. д. В аргумент Root передаётся ссылка на компонент, для которого когда-то был запущен метод TStream.WriteComponent, и если мы пришли к сериализации данного компонента через один или несколько «каскадов», Root<>Self.

В GetChildren от программиста требуется, чтобы он один или несколько раз вызвал Proc, указывая в качестве аргумента те компоненты, которые нужно сериализовать вместе с данным. Например, в классах TForm и TDataModule в GetChildren организуется цикл по элементам массива TComponent.Components:

```
var
  I: Integer;
  OwnedComponent: TComponent;
begin
  inherited GetChildren(Proc, Root);
  if Root = Self then
    for I := 0 to ComponentCount - 1 do
    begin
      OwnedComponent := Components[I];
      if not OwnedComponent.HasParent then Proc(OwnedComponent);
    end;
end;
```

В методе же TCustomActionList.GetChildren имеется цикл по Action'ам, которые входят в данный ActionList.

Как выглядит результат работы GetChildren в dfm-формате, мы уже видели: объект сохраняется как бы «внутри» того, который отвечает за его сериализацию. Так, на представленном выше фрагменте dfm-файла, Form1 отвечает за сериализацию Button1, Button2 и ActionList1, а ActionList1 – за сериализацию Action1 и Action2.

Во время же загрузки из потока компонент может узнать о том, кто отвечал за его сериализацию, и предпринять соответствующие действия. Для этого надо переопределить метод

```
procedure SetParentComponent(Value: TComponent); dynamic;
```

Например, TAction таким образом при десериализации может зарегистрировать себя в сериализовавшем его ActionList'е.

Какой класс выбрать в качестве базового

При проектировании классов бизнес-логики часто возникает очень важный вопрос: какой класс выбрать в качестве базового для того или иного класса? Мы рассмотрели ряд классов, с которыми работает механизм сериализации, но их обилие не должно сбивать нас с толку. Думаю, что вышесказанного уже достаточно, чтобы сделать оптимальный выбор. Для удобства нарисуйте диаграмму классов и отобразите кратность связей между их объектами.

Для класса корневого объекта у нас выбора нет – это должен быть наследник TComponent.

Если требуется, чтобы на объект некоторого класса ссылались сразу несколько объектов – выбора тоже нет, этот класс должен наследоваться от TComponent. Владеть экземплярами такого класса будет, разумеется, корневой объект, а вот насчёт того, кто будет ответственным за сериализацию – стоит подумать. Класс объекта, ответственного за сериализацию, тоже должен быть наследником TComponent.

С оставшимися классами мы поступаем так: для связей типа «один к одному» выбираем TPersistent, в случае же связей типа «один ко многим» создается свойство, тип которого должен быть наследником класса TCollection, а классы элементов коллекции нужно унаследовать от TCollectionItem.

Регистрация класса в потоковой системе

Итак, предположим, что теперь нам удалось сохранить в поток всю нужную нам информацию – все объекты, все определяющие их свойства, все перекрёстные ссылки. Осталось теперь в нужный момент всё это прочесть. Оказывается, тут есть тонкий момент, требующий от программиста одного микроскопического усилия, без которого, однако, ничего работать не будет.

Перед началом десериализации каждый класс объектов, которые будут восстанавливаться из потока, должен быть зарегистрирован в потоковой системе.

Сделать это просто: для этого нужно лишь воспользоваться процедурами из модуля Classes

```
{для одного класса}
  procedure RegisterClass(AClass: TPersistentClass);
{сразу для нескольких классов}
  procedure RegisterClasses(AClasses: array of TPersistentClass);
```

Где:

```
type TPersistentClass = class of TPersistent;
```

В качестве аргумента мы передаём регистрируемый класс или массив регистрируемых классов. Регистрировать нужно действительно *все* классы, участвующие в сериализации, а не только корневой. Если этого не сделать, при десериализации возникнет исключение с сообщением “Class not found”.

Процедуры RegisterClass и RegisterClasses удобнее и правильнее всего вызывать в initialization-секции .pas-модуля. Это даст нам уверенность в том, что до любого использования классов, объявленных в модуле, они будут зарегистрированы в потоковой системе.

Классы визуальных компонентов, зарегистрированных в IDE с помощью RegisterComponents, в вызове RegisterClass не нуждаются.

Часть III. «Высший пилотаж»

О псевдосвойствах

Есть одно существенное ограничение, относящееся ко всему сказанному выше. Рассмотренные случаи позволяют сериализовать только ту информацию, которая содержится в read/write published-свойствах класса. Однако всякому программисту известно, что информацию класса далеко не всегда допустимо выносить в такие свойства. Прежде всего, некоторые из свойств по архитектурному замыслу должны быть read-only, т. е. прямое присвоение их значений в программе не должно допускаться. Если мы вынесем read-only свойство в published-секцию, его

значение сохранится в поток, но при десериализации не восстановится. С другой стороны, некоторые свойства вообще нельзя выносить за пределы protected-секции, а значения иных нуждающихся в сериализации полей должны быть только private.

И, кроме того, существует ещё один случай, для которого описанные выше приёмы сериализации бесполезны: если вместе с объектом должна сохраняться информация, представленная во вполне определённом формате – например, картинка или звук.

Здесь нас выручат псевдосвойства, которые предоставляют более общие механизмы сериализации, чем те, что доступны при использовании published-свойств, и дают фактически неограниченную свободу при создании систем, обладающих возможностью сохранять в потоке своё состояние.

Определение псевдосвойств

Псевдосвойство – это то, что выглядит в dfm-формате как сериализованное published-свойство, но на самом деле таковым не является. Например, вот как выглядит сериализованная композиция TImage–TPicture, где объект типа TPicture доступен через свойство Picture:

```
object Image1: TImage
. . .
  Picture.Data = {
    0A544A504547496D616765. . .
    E0FF000BFFD9}
end
```

Как и полагается, TPicture унаследован от TPersistent, однако не стоит искать Data среди его published-свойств.

Чтобы сделать информацию сериализуемой, не вынося её в published-свойства рассмотренных типов, можно воспользоваться псевдосвойствами, которые задаются в предназначенном для переопределения protected-методе класса TPersistent

```
procedure DefineProperties(Filer: TFiler); virtual;
```

Как и в случае с рассмотренным выше методом GetChildren, предполагается, что программист, переопределив данный метод, выполнит в нём заданные действия. А именно, для каждого будущего псевдосвойства вызовет у объекта Filer один из двух методов:

```
procedure DefineProperty(const Name: string;
  ReadData: TReaderProc;
  WriteData: TWriterProc;
  HasData: Boolean); virtual; abstract;
procedure DefineBinaryProperty(const Name: string;
  ReadData, WriteData: TStreamProc; HasData: Boolean); virtual; abstract;
```

где

```
type
  TReaderProc = procedure(Reader: TReader) of object;
  TWriterProc = procedure(Writer: TWriter) of object;
  TStreamProc = procedure(Stream: TStream) of object;
```

Аргумент Name этих методов предназначен для передачи в него имени будущего псевдосвойства. Имена псевдосвойств должны быть корректными идентификаторами с точки зрения синтаксиса Pascal и не конфликтовать с именами других псевдо- и published-свойств – впрочем, эти требования, как ни странно, не категоричны. В ReadData и WriteData передаются ссылки на методы сериализуемого класса, отвечающие за чтение и запись информации псевдосвойства (ниже мы подробнее остановимся на этих методах). Аргумент HasData позволяет реализовать то, о чём говорилось выше в связи с ключевым словом default. Для того чтобы при определённых условиях псевдосвойство не сериализовалось, нужно сделать так, чтобы к моменту сериализации

HasData получало False. В процессе же десериализации HasData никак не учитывается – если свойство будет найдено в потоке, оно будет прочитано, если нет – информация в объекте останется, как есть.

Например, в реализации класса TPicture читаем:

```
procedure TPicture.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
    . . .
  end;

begin
  Filer.DefineBinaryProperty('Data', ReadData, WriteData, DoWrite);
end;
```

ReadData и WriteData здесь – private-методы класса TPicture. Ну как, не запутались? DefineProperties – это метод класса TPesistent, его мы переопределяем. DefineBinaryProperty – это метод класса TFiler, его мы в обязательном порядке вызываем в переопределённом DefineProperties.

Абстрактный класс TFiler является базовым для двух других: TReader и TWriter. Во время вызова TStream.WriteComponent создаётся экземпляр класса TWriter. Для каждого участвующего в сериализации объекта ссылка на TWriter сначала передаётся в метод DefineProperties, а затем – в методы, предназначенные для записи информации псевдосвойств. При вызове метода TStream.ReadComponent объект типа TReader играет симметричную роль.

Процедуры записи и чтения

Пара методов, указываемая в качестве аргументов TFiler.Define[Binary]Property, должна содержать в себе функциональность, отвечающую за запись и чтение информации соответствующих псевдосвойств. Если псевдосвойство было определено через DefineBinaryProperty, эти методы в качестве параметра получат экземпляр TMemoryStream, в который метод записи может записывать всё, что угодно – лишь бы метод чтения мог это прочесть. Однако это будет происходить уже без всякой помощи механизмов сериализации.

Если же мы воспользовались методом DefineProperty, то метод записи получит ссылку на объект Writer, а метод чтения – на объект Reader, методами которых можно будет пользоваться для того, чтобы упростить себе сохранение информации. Что это нам даёт? Дело в том, что, вызвав соответствующий метод объекта Writer (Reader), мы можем записать (прочитать) некоторое значение одного из следующих типов:

Метод TWriter	Метод TReader	Сериализуемый тип
WriteBoolean	ReadBoolean	Boolean
WriteInteger	ReadInteger	LongInt (= Integer)
WriteInt64	ReadInt64	Int64
WriteSingle	ReadSingle	Single
WriteFloat	ReadFloat	Extended
WriteCurrency	ReadCurrency	Currency
WriteDate	ReadDate	TDateTime
WriteChar	ReadChar	Char
WriteString	ReadString	String
WriteWideString	ReadWideString	WideString
WriteIdent	ReadIdent	строка, содержащая идентификатор

Как видим, тут есть практически всё, кроме специального метода для записи самого «ходового» типа с плавающей точкой – Double, для которого применяют Write/ReadFloat.

По сути своей методы Write/ReadIdent очень похожи на Write/ReadString, но предназначены для записи и чтения идентификаторов. Этими методами следует пользоваться для записи/чтения, например, имен компонентов или строк, соответствующих значениям enumerated-типов.

Например, код

```
type TSuit = (sClub, sDiamond, sHeart, sSpade);

. . .

procedure TMyClass.DefineProperties(Filer: TFiler);
begin
  Filer.DefineProperty('MyFakeProperty', ReadMyFakeProperty, WriteMyFakeProperty, True);
end;

procedure TMyClass.WriteMyFakeProperty(Writer: TWriter)
begin
  Writer.WriteIdent(GetEnumName(TypeInfo(TSuit), Integer(sHeart));
end;
```

породит такую строчку в dfm-файле:

```
MyFakeProperty = sHeart
```

И если в классе TMyClass имеется поле FSuit: TSuit, то прочитать его можно будет с помощью метода:

```
procedure TMyClass.ReadMyFakeProperty(Reader: TReader)
begin
  FSuit := TSuit(GetEnumValue(TypeInfo(TSuit), Reader.ReadIdent));
end;
```

С помощью указанных методов TWriter/TReader в одной паре процедур записи/чтения мы можем отрабатывать не только одно значение, но также и массив, или даже последовательность значений различных типов. Для этого в начале метода записи необходимо вызвать метод TWriter WriteListBegin, в конце – WriteListEnd, а в середине – нужную последовательность методов, записывающих единичные значения. К примеру, вставленный в метод записи код:

```
with Writer do
begin
  WriteListBegin;
  WriteInteger(123);
  WriteString('This is a string');
  WriteIdent('AnIndentificator');
  WriteListEnd;
end;
```

породит такой фрагмент файла:

```
MyFakeProperty = (
  123
  'This is a string'
  AnIndentificator)
```

Чтобы теперь прочитать это свойство, необходимо создать процедуру чтения, вызывающую ReadListBegin, затем – правильную последовательность методов чтения, и в конце – ReadListEnd. К счастью, в читаемом псевдосвойстве можно динамически определить, «что идёт следом». Для этого можно пользоваться методами класса TReader - NextValue: TValueType и EndOfList: Boolean, которые, соответственно, указывают на тип следующего сериализованного значения и дают

знать о том, что список значений подошёл к концу.

К примеру, рассмотренное выше псевдосвойство можно прочитать так:

```
var
  IntVar: Integer;
  StringVar, IdentVar: string;
begin
  with Reader do
  begin
    ReadListBegin;
    while not EndOfList do
      case NextValue of
        vaInt8, vaInt16, vaInt32: IntVar := ReadInteger;
        vaString: StringVar := ReadString;
        vaIdent: IdentVar := ReadIdent;
      end;
    ReadListEnd;
  end;
end;
```

Fix-up своими руками

С помощью методов TWriter.WriteIdent и TReader.ReadIdent можно записывать и читать имена компонентов, входящих в сериализуемую систему объектов. Но от самих по себе имён компонентов мало толку, если по завершении загрузки они не будут замещены указателями.

Процесс замещения имён указателями в справочной системе VCL фигурирует как “fix-up” и при использовании read-write published-свойств производится автоматически. Однако задействованные при этом механизмы, к сожалению, не описаны в справочной системе и частично закрыты от разработчиков. Поэтому при использовании псевдосвойств нам придётся делать fix-up своими руками – к счастью, это нетрудно.

Идея механизма заключается в следующем. В процессе чтения системы объектов нам будут попадаться места, в которых в поле произвольного объекта необходимо записать ссылку на другой компонент. Однако всё, что мы можем получить непосредственно в момент чтения – это имя компонента, на который надо ссылаться; сам же компонент при этом может быть ещё не прочитан, и, соответственно, не существовать в памяти. И потому нам останется только лишь отложить запись ссылки в поле на более поздний срок, сохранив где-то сведения о том, что по такому-то адресу нужно записать ссылку на компонент, имеющий такое-то имя. После того, как загрузка системы объектов будет завершена, мы пройдёмся по записанным сведениям и восстановим все ссылки на основании имён прочитанных компонентов.

Таким образом, нам потребуется создать всего две процедуры: SaveName – для записи сведений об отложенном присвоении значения ссылки и ResolveNames – для разрешения имён в указатели. Эти процедуры универсальны и могут быть вынесены в модуль, предназначенный для использования в различных проектах.

Модуль Fixups.pas

```
unit Fixups;

interface

uses Classes;
type
  TCptRef = ^TComponent;

procedure SaveName(const Name: string; FieldRef: TCptRef);
procedure ResolveNames(RootObject: TComponent);

implementation

var
```

```
  FixupInfo: TStringList;

procedure SaveName(const Name: string; FieldRef: TCptRef);
begin
  FixupInfo.AddObject(Name, TObject(FieldRef));
end;

procedure ResolveNames(RootObject: TComponent);
var
  i, j: Integer;
  CurName: string;
  CurCpt: TComponent;
begin
  Assert(Assigned(RootObject), 'Корневой объект = nil');
  with RootObject do
    for i := 0 to Pred(ComponentCount) do
    begin
      CurCpt := Components[i];
      CurName := CurCpt.Name;
      for j := Pred(FixupInfo.Count) downto 0 do
        if FixupInfo[j] = CurName then
        begin
          TCptRef(FixupInfo.Objects[j])^ := CurCpt;
          FixupInfo.Delete(j);
        end;
    end;
  Assert(FixupInfo.Count = 0, 'Остаются не разрешённые имена');
  FixupInfo.Clear;
end;

initialization
  FixupInfo := TStringList.Create;
finalization
  FixupInfo.Free;
end.
```

Ради экономии места тут приведён простейший из возможных вариантов системы разрешения имён в ссылки. Первое его улучшение может быть связано с необходимостью сделать так, чтобы читаемые из потока объекты определённым образом «реагировали» на присвоение значений их полям. В этом случае в список FixupInfo надо сохранять не адреса полей, а указатели на методы присваивания. Ну и кроме того, этот модуль потребуется переделать, если возникнет желание задействовать его в многопоточной программе, где возможно параллельное использование FixupInfo при загрузке разных систем объектов.

Итак, предположим, что в сериализуемом классе TMyClass имеется поле FField:TAnotherClass (где TAnotherClass наследуется от TComponent), и мы хотим сделать это поле сериализуемым через псевдосвойства. Для этого надо написать следующие процедуры записи и чтения:

```
procedure TMyClass.WriteField(Writer: TWriter);
begin
  Writer.WriteIdent(FField.Name);
end;
procedure TMyClass.ReadField(Reader: TReader);
begin
  SaveName(Reader.ReadIdent, @FField);
end;
```

Тут возникает последний вопрос: как мы узнаем, что пора вызывать процедуру ResolveNames? Ответ прост: нужно вызвать ResolveNames в переопределённом методе Loaded корневого объекта, указывая Self в качестве аргумента. Protected-метод класса TComponent:

```
procedure Loaded; virtual;
```

автоматически вызывается после завершения загрузки компонента и всех компонентов, за сериализацию которых он отвечает. Этот метод предназначен главным образом для того, чтобы в нём производить fix-up.

Эта статья опубликована в журнале RSDN Magazine #6-2003. Информацию о журнале можно найти здесь