Руководство программиста

Март 2007

Оглавление

Глава 1

Введение

1 Введение

Данное руководство является логическим продолжением руководства разработчика и содержит информацию адресованную Delphi программистам. В руководстве рассматриваются вопросы создания служебных пакетов для ИСР CoolManager. Система CoolManager имеет открытую архитектуру, что позволяет подключать пакеты созданные в Delphi к интегрированной среде разработки (ИСР). По своей структуре ИСР CoolManager очень похожа на IDE Delphi, поэтому в ИСР можно загружать стандартные VCL библиотеки. Процесс создания визуальных форм аналогичен тому, как это делается в Delphi. Но есть и отличия, например, при загрузке VCL пакета, в скриптовом языке доступны только компоненты регистрируемые посредством функции RegisterComponents, так же доступны только published свойства этих компонент. Для того, чтобы получить доступ к объявлениям сделанным в разделе public компонент, использовать классы, типы, процедуры и функции объявленные в VCL пакетах, следует написать дополнительный код. Конфигурация, созданная в ИСР, может включать описание метаданных прикладной задачи. Создание, редактирование и представление метаданных аналогично тому, как это делается в системе 1С. ИСР содержит конфигуратор метаданных, предназначенный для редактирования и просмотра метаобъектов конфигурации, в конфигураторе метаданные представлены в виде иерархического списка. В отличие от 1С, конфигуратор не содержит предопределенного набора метаклассов, все классы метаданных создаются в Delphi, оформляются в виде bpl пакетов и подключаются к ИСР CoolManager. По мере необходимости, Delphi программист может разрабатывать новые метаклассы для реализации специфических объектов предметной области прикладной системы.

Исходя из вышесказанного, можно выделить две основные задачи решаемые посредством создания служебных bpl пакетов для ИСР:

- Расширение возможностей встроенного скриптового языка ИСР
- Создание метаклассов для конфигуратора ИСР

На самом деле обе задачи тесно взаимосвязаны. Когда создается новый класс метаданных, требуется сделать его свойства и методы доступными в скриптовом языке.

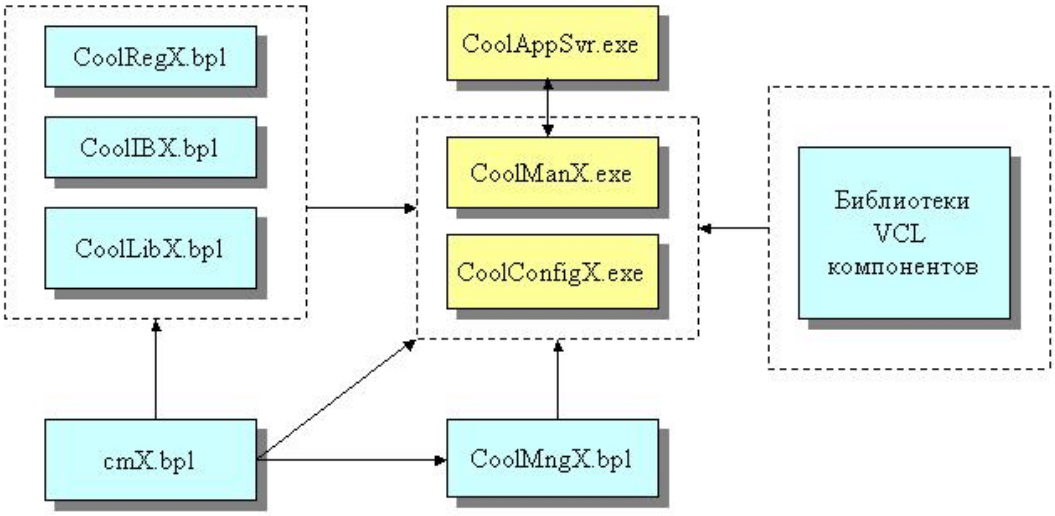
1.1 Приложения и пакеты системы

Ниже приводится перечень приложений и пакетов входящих в стандартную поставку системы CoolManager. Все скомпилированные приложения и bpl пакеты находятся в каталоге \BIN.

Объект	Описание
CoolConfig**X**.exe	Интегрированная среда разработки приложений.
CoolAppSvr.exe	Сервер приложений. Назначение и использование описано в руководстве разработчика.
CoolMan**X**.exe	Программа просмотрщик. Используется совместно с сервером приложений. Назначение и использование описано в руководстве разработчика.
cm**X**.bpl	Интерфейсный пакет для ИСР CoolManager. Описывает базовые типы и классы ИСР CoolManager. Исходные тексты пакета расположены в каталоге \source\cm
CoolMng**X**.bpl	Содержит ядро системы CoolManager. Исходные тексты не предоставляются.
CoolReg**X**.bpl	Пакет содержит объекты для подключения к скриптовому языку компонентов пакета CoolComp и некоторых свойств и методов библиотеки VCL. Исходные тексты пакета расположены в каталоге source\StdReg.
CoolIB**X**.bpl	Пакет содержит классы для доступа к InterBase серверу баз данных. Исходные тексты пакета расположены в каталоге source\ib.
CoolLib**X**.bpl	Библиотека компонентов для создания систем автоматизации учета.

Примечание: Под буквой **X**, в именах файлов, подразумевается цифра обозначающая версию Delphi.

Взаимосвязи между приложениями и пакетами библиотеки изображены на рисунке:



Ядром системы являются приложения CoolConfig**X** (интегрированная среда разработкм) и CoolMan**X** (программа просмотрщик), далее два этих приложения будем называть ядром. Ядро использует стандартные VCL библиотеки, например, CoolCompX является стандартным VCL пакетом, который может быть установлен и в IDE Delphi. Пакет содержит набор полезных компонент, аналогов которым нет в стандартной библиотеке VCL. CoolMng**X** - это главная библиотека ядра, содержит классы и объекты используемые как на этапе проектирования, так и на этапе выполнения приложения. Отдельную группу составляют служебные пакеты, эти пакеты разрабатываются специально для ИСР и расширяют её возможности. Пакет CoolReg**X** содержит код регистрации некоторых стандартных классов, типов и процедур Delphi используемых ИСР CoolManager. CoolIB**X** содержит классы для доступа к InterBase серверу. CoolReg**X** и CoolIB**X** поставляются в исходных текстах. Пакет CoolLibX содержит классы и компоненты для создания систем автоматизации учета. Ключевую роль играет пакет cmX, он содержит базовые классы, переменные, типы и интерфейсы используемые ядром системы. Через объявленные в нем переменные и интерфейсы служебные пакеты могут получить доступ к ядру ИСР.

1.2 Пакет cmX

Пакет cmX является шлюзом, который дает возможность служебным пакетам взаимодействовать с ядром системы. Пакет поставляется в исходных текстах для того, что бы программист создающий служебные пакеты мог детально ознакомиться с его содержимым.

Внимание! Не в коем случае не следует вносить изменения в интерфейсную часть модулей включенных в пакет, это приведет к неработоспособности системы, т.к. приложения ядра так же используют пакет cmX.

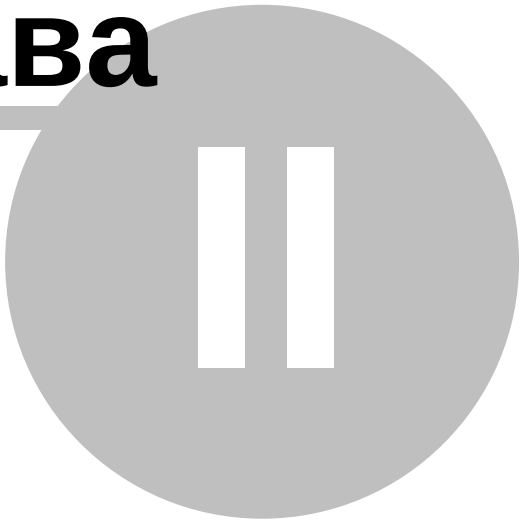
Список модулей пакета cmX.bpl

Модуль	Описание
cmScriptReg.pas	Содержит интерфейс IcmScriptRegister для подключения новых объектов к скриптовому языку ИСР.
cmEventMng.pas	Содержит интерфейс менеджера событий IEventManager, используется для регистрации в ИСР обработчиков событий.

cmEvents.pas	Библиотека для регистрации в ИСР стандартных обработчиков событий библиотеки VCL.
cmConfigClass.pas	Содержит классы и процедуры для разработки и регистрации в ИСР классов метаданных.
cmDataBase.pas	Содержит абстрактные классы для организации доступа к базам данных.
cmStringEditorForm.pas	Содержит редактор свойств наследованных от TStrings.

В последующих главах будут подробно рассмотрены все модули пакета и приведены примеры их использования.

Глава



Доступ к скриптовому языку

2 Доступ к скриптовому языку

В качестве скриптового языка системы Cool Manager используется FastScript www.fast-report.com. В комплект поставки входит пакет с реализацией скриптового языка - fsX.bpl, пакет расположен в каталоге \BIN и поставляется без исходных текстов. Для подключения объектов к FastScript, в пакете cmX.bpl содержится модуль cmScriptReg.pas. В модуле объявлен интерфейс IcmScriptRegister с описанием методов доступа к главному модулю скрипта и переменная IcmRegister: IcmScriptRegister. IcmScriptRegister инициализируется при запуске ядра системы, используется служебными пакетами для получения ссылки на интерфейс IcmScriptRegister. В данной главе объясняется использование интерфейса IcmScriptRegister.

Interface раздел модуля cmScriptReg.pas

```
 TcmVarType = (cmtInt, cmtBool, cmtFloat, cmtChar, cmtString, cmtClass,cmtArray,
              cmtVariant, cmtEnum, cmtConstructor);
//.........Интерфейс для доступа к скриптовому языку.........................
 //классовый метод для вызова внешней процедуры или функции
 TcmCallMethodEvent = function(Instance: TObject; ClassType: TClass;
            const MethodName: String; var Params: Variant): Variant of object;
 //установка значения свойства
 TcmGetValueEvent = function(Instance: TObject; ClassType: TClass;
            const PropName: String): Variant of object;
 //чтение значения свойства
 TcmSetValueEvent = procedure(Instance: TObject; ClassType: TClass;
            const PropName: String; Value: Variant) of object;

 //интерфейс для регистрации объектов скриптового языка
 IcmScriptRegister = interface
 ['{4F70A4B2-C78C-48FC-BDD5-262CD98BAC10}']
  //добавить константу. Пример:
  procedure AddConst(const Name, Typ: String; const Value: Variant);
  // Добавить тип. Пример: AddType('TDateTime', fvtFloat) }
  procedure AddType(const TypeName: String; ParentType: TcmVarType);
  //добавить перечисление
  //  AddEnum('TFontPitch', 'fpDefault, fpFixed, fpVariable')
  //все элементы перечисления нумеруются 0,1,2,3,4.....
  procedure AddEnum(const Typ, Names: String);  //добавить множество
  //  AddEnumSet('TFontStyles', 'fsBold, fsItalic, fsUnderline')
  //элементы множества нумеруются следующим образом 1,2,4,8,.. }
  procedure AddEnumSet(const Typ, Names: String);
  //добавить процедуру или функцию
  procedure AddMethod(const Syntax: String; CallEvent: TcmCallMethodEvent;
            const Category: String = ''; const Description: String = '');
  //Добавить объект. Пример:
  // AddObject('Memo1', Memo1) }
  procedure AddObject(const Name: String; Obj: TObject);
  //Добавить переменную. Пример:
  //  AddVariable('n', 'Variant', 0) }
  procedure AddVariable(const Name, Typ: String; const Value: Variant);
  //.......................Регистрация классов.............................
  //Добавить класс
  function AddClass(AClass: TClass; const Ancestor: String): TObject;
  //добавить конструктор. Пример:
  //    AddConstructor(Obj, 'constructor Create(AOwner: TComponent)', MyCallEvent) }
  procedure AddConstructor(AClass: TObject; Syntax: String; CallEvent: TcmCallMethodEvent);
  // Добавить свойство. Пример:
  //    AddProperty(Obj, 'Font', 'TFont', MyGetEvent, MySetEvent) }
  procedure AddProperty(AClass: TObject; const Name, Typ: String;
                        GetEvent: TcmGetValueEvent; SetEvent: TcmSetValueEvent = nil);
  //Добавить свойство по умолчанию. Пример:
  //   AddDefaultProperty(Obj, 'Cell', 'Integer,Integer', 'String', MyCallEvent)
  // регистрация свойства: property Cell[Index1, Index2: Integer]: String
  procedure AddDefaultProperty(AClass: TObject; const Name, Params, Typ: String;
                               CallEvent: TcmCallMethodEvent; AReadOnly: Boolean = False);
  //Добавить индексированное свойство.
  procedure AddIndexProperty(AClass: TObject; const Name, Params, Typ: String;
```

```
      CallEvent: TcmCallMethodEvent; AReadOnly: Boolean = False);
  //Добавить метод. Пример:
  //  AddClassMethod(Obj, 'function IsVisible: Boolean', MyCallEvent) }
  procedure AddClassMethod(AClass: TObject; const Syntax: String; CallEvent: TcmCallMethodEvent);
  //Добавить форму
  procedure AddForm(Form: TComponent);
  //Установка владельца регистрируемых объектов
  procedure SetAddedBy(Obj: TObject);
  function GetAddedBy: TObject;
  property AddedBy: TObject read GetAddedBy write SetAddedBy;
  procedure RemoveAddedBy(Obj: TObject);
 end;

var
 //Ссылка на интерфейс, посредством которого производится регистрация объектов скрипта.
 IcmRegister: IcmScriptRegister;
 //ссылка на интерфейс доступа к объектам дизайнера форм
 IcmDesigner: IcmDesignerConfig;
```

2.1 Добавление процедуры

Для добавления процедуры/функции в скрипт выполните следующие действия:

- Создайте обработчик - функцию CallMethod.
- Вызовите метод IcmRegister.AddMethod. Первый параметр - это синтаксис функции (обратите внимание - синтаксис, независимо от используемого вами языка, должен быть паскалевским!), второй - ссылка на обработчик TcmCallMethodEvent.

```
{ собственно, функция }
procedure DelphiFunc(s: String; i: Integer);
begin
ShowMessage(s + ', ' + IntToStr(i));
end;

{ обработчик TcmCallMethodEvent }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
DelphiFunc(Params[0], Params[1]);
end;

procedure RegisterProc;
begin
{ делаем процедуру DelphiFunc доступной в скрипте }
IcmRegister.AddMethod('procedure DelphiFunc(s: String; i: Integer)', CallMethod);
end;
```

Если вы желаете добавить несколько методов, вы можете сделать это, используя один дескриптор метода:

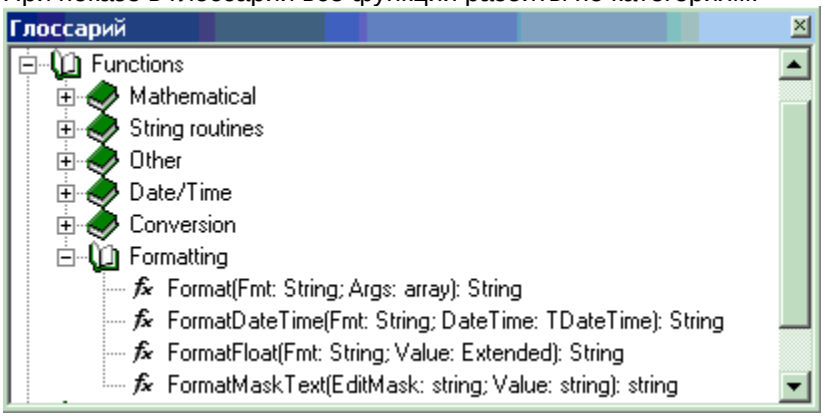
```
IcmRegister.AddMethod('procedure DelphiFunc(s: String; i: Integer)', CallMethod);
IcmRegister.AddMethod('procedure DelphiFunc2(s: String)', CallMethod);

{ дескриптор метода }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
 { определение вызываемого метода }
 if MethodName = 'DELPHIFUNC' then DelphiFunc(Params[0], Params[1])
 else if MethodName = 'DELPHIFUNC2' then DelphiFunc2(Params[0]);
end;
```

Обратите внимание на два необязательных параметра в объявлении метода AddMethod:

```
  procedure AddMethod(const Syntax: String; CallEvent: TcmCallMethodEvent;
                      const Category: String = ''; const Description: String = '');
```

- Kategory - имя категории, к оторой следует отнести процедуру или функцию.
- Description - краткое описание процедуры или функции.

При показе в глоссарии все функции разбиты по категориям:



Вы можете отнести функцию к одной из уже объявленных категорий, либо создать новую. Краткое писание функции Description используется ядром библиотеки FastReport и не оказывает на ИСР CoolManager никакого влияния.

2.2 Добавление функции

Добавление функции аналогично добавлению процедуры.

```
IcmRegister.AddMethod('function DelphiFunc2(s: String): Boolean', CallMethod);

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
 Result := DelphiFunc(Params[0]);
end;
```

2.3 Добавление функции с параметрами по умолчанию

Вам не нужно заботиться о параметрах по умолчанию - FastScript подставит их автоматически. Параметры var должны обрабатываться вами.

```
IcmRegister.AddMethod('function DelphiFunc(var s: String; i: Integer =0): Boolean', CallMethod);

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
var
 s: String;
begin
 s := Params[0];
 Result := DelphiFunc(s, Params[1]);
 Params[0] := s;
end;
```

2.4 Добавление функции с параметром class

Поскольку все параметры представляются как массив типа Variant, вам надо преобразовать их в объекты.

```
IcmRegister.AddMethod('procedure HideButton(Button: TButton)', CallMethod);

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
 TButton(Integer(Params[0])).Hide;
```

```
end;
```

2.5 Добавление функции возвращающей значение типа class

Поскольку значения, возвращаемые дескриптором метода, это массив типа Variant, вам надо преобразовать результаты типа TObject к Variant.

```
IcmRegister.AddMethod('function MainForm: TForm', CallMethod);

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
 Result := Integer(Form1);
end;
```

2.6 Добавление константы

Для добавления в скрипт константы вызовите метод IcmRegister.AddConst. Первый параметр - это наименование константы, второй - тип (должен быть одним из стандартных типов), третий - значение.

```
{ добавление в скрипт константы }
IcmRegister.AddConst('pi', 'Extended', 3.14159);
```

2.7 Добавление переменной

Для добавления в скрипт переменной вызовите метод IcmRegister.AddVariable. Этот метод подобен методу AddConst, за исключением того, что вы можете изменить значение переменной в скрипте. Обратите внимание, что переменная Delphi не изменится после выполнения скрипта.

```
IcmRegister.AddVariable('i', 'Integer', i);
```

2.8 Добавление объекта в скрипт

Для добавления объекта в скрипт вызовите метод IcmRegister.AddObject. Первый параметр это имя объекта, второй - собственно, объект.

```
IcmRegister.AddObject('Button1', Button1);
```

Если добавляемый объект имеет незарегистрированный класс, то предварительно надо зарегистрировать его:

```
IcmRegister.AddClass(TForm1, 'TForm');
IcmRegister.AddObject('Form1', Form1);
```

Вы также можете использовать метод IcmRegister.AddForm для добавления формы или модуля данных вместе со всеми дочерними компонентами:

```
IcmRegister.AddForm(Form1);
```

В этом случае регистрировать класс формы с помощью AddClass не требуется.

Теперь вы можете обращаться к элементам формы из скрипта:

```
Form1.Button1.Caption := '...'
```

2.9 Добавление описания типа

Для добавления собственного типа в скрипт вызовите метод IcmRegister.AddType. Первый параметр это наименование типа, второй - один из следующих поддерживаемых типов:

```
TcmVarType = (cmtInt, cmtBool, cmtFloat, cmtChar, cmtString, cmtClass,
              cmtArray, cmtVariant, cmtEnum, cmtConstructor);
IcmRegister.AddType('TCursor', cmtInt);
```

2.10 Добавление перечисления

Для добавления перечислимого типа в скрипт вызовите метод IcmRegister.AddEnum. Первый параметр это имя типа, второй - значения, разделенные запятыми.

```
IcmRegister.AddEnum('TPrinterOrientation', 'poPortrait, poLandscape');
```

2.11 Добавление множества

Для добавления множественного в скрипт вызовите метод IcmREgister.AddEnumSet. Первый параметр это наименование типа, второй - значения, разделенные запятыми.

```
IcmRegister.AddEnumSet('TFontStyles', 'fsBold, fsItalic, fsUnderline,fsStrikeOut');
```

2.12 Добавление класса

Для добавления в скрипт класса вызовите метод IcmRegister.AddClass. Первый параметр это наименование класса, второй - имя базового класса. Например:

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

var Obj: TObject;

Obj:=IcmRegister.AddClass(TMyClass, 'TObject');
```

Это сделает все published свойства данного класса доступными. Метод возвращает ссылку на класс, эта ссылка используется для регистрации свойств и методов класса.

2.13 Реализация public свойств и методов класса

Метод AddClass автоматически добавляет все published свойства класса. Public свойства и методы требуют дополнительной работы. Следующий пример демонстрирует как добавить p ublic-метод в класс. Вам потребуется создать дескриптор метода (функция типа TcmCallMethodEvent).

```
var Obj: TObject;

begin
...
{ добавим новый класс, унаследованный от TObject }
Obj:=IcmRegister.AddClass(TList, 'TObject') ;
{ добавим public методы }
IcmRegister.AddClassMethod(Obj,'function Add(Item: TObject): Integer', CallMethod);
IcmRegister.AddClassMethod(Obj,'procedure Clear', CallMethod);
...
end;

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
```

```
 Result := 0;
 if MethodName = 'ADD' then
   { преобразуем параметр типа Variant к типу Pointer и вызовем с ним метод Add }
   TList(Instance).Add(Pointer(Integer(Params[0])))
 else if MethodName = 'CLEAR' then TList(Instance).Clear
end;
```

Для реализации свойства вам потребуется создать дескриптор метода и два дескриптора свойства типов TcmGetValueEvent и TcmSetValueEvent:

```
TcmGetValueEvent = function(Instance: TObject; ClassType: TClass;
                            const PropName: String): Variant of object;
TcmSetValueEvent = procedure(Instance: TObject; ClassType: TClass;
                            const PropName: String; Value: Variant) of object;
```

Индексируемые (indexed) свойства и свойства по умолчанию (default) описываются дескриптором метода, обычные свойства описываются дескрипторами свойств Get/Set.

```
var Obj: TObject;
begin
...
Obj:=TcmRegister.AddClass(TStrings, 'TPersistent');
{ property CommaText: String }
IcmRegister.AddProperty(Obj, 'CommaText', 'string', GetProp, SetProp);
{ property Count: Integer только для чтения, поэтому второй парметр равен nil }
IcmRegister.AddProperty(Obj, 'Count', 'Integer', GetProp, nil);
{ index property Objects[Index: Integer]: TObject }
IcmRegister.AddIndexProperty(Obj,'Objects', 'Integer', 'TObject', CallMethod);
{ default property Strings[Index: Integer]: String }
IcmRegister.AddDefaultProperty(Obj,'Strings', 'Integer', 'string', CallMethod);
...
end;

{ обработчик }
function CallMethod(Instance: TObject; ClassType: TClass;
                    const MethodName: String; var Params: Variant): Variant;
begin
 Result := 0;
 if MethodName = 'OBJECTS.GET' then
       Result := Integer(TStrings(Instance).Objects[Params[0]])
 else if MethodName = 'OBJECTS.SET' then
      TStrings(Instance).Objects[Params[0]] := TObject(Integer(Params[1]))
 else if MethodName = 'STRINGS.GET' then
      Result := TStrings(Instance).Strings[Params[0]]
 else if MethodName = 'STRINGS.SET' then
      TStrings(Instance).Strings[Params[0]] := Params[1]
end;

{ обработчик }
function GetProp(Instance: TObject; ClassType: TClass; const PropName: String): Variant;
begin
 Result := 0;
 if PropName = 'COMMATEXT' then Result := TStrings(Instance).CommaText
 else if PropName = 'COUNT' then Result := TStrings(Instance).Count
end;

{ обработчик }
procedure SetProp(Instance: TObject; ClassType: TClass; const PropName: String; Value: Variant);
begin
 if PropName = 'COMMATEXT' then TStrings(Instance).CommaText := Value
end;
```

2.14 Удаление описаний из скрипта

При выгрузке служебного пакета из ИСР требуется удалить все описания добавленные данным пакетом в скрипт. Для решения этой задачи интерфейс IcmScriptRegister содержит следующие объявления:

```
property AddedBy: TObject read GetAddedBy write SetAddedBy;
procedure RemoveAddedBy(Obj: TObject);
```

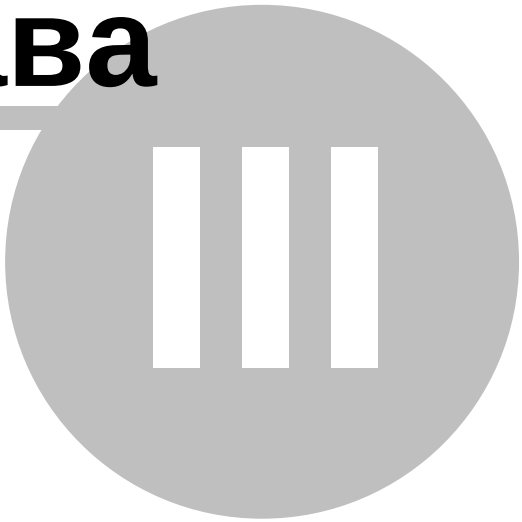
Свойство AddedBy содержит ссылку на владельца добавляемого описания, процедура RemoveAddedBy удаляет все описания сделанные указанным владельцем. Для демонстрации работы со свойством AddedBy приведем выдержку из кода модуля cmStdReg.pas включенного в пакет CoolRegX.bpl:

```
type

 TcmReg = class
  constructor Create;
  destructor Destroy; override;
...
 end;

var cmReg: TcmReg;

implementation

constructor TcmReg.Create;
 var Obj: TObject;
begin
 IcmRegister.AddedBy:=Self;
 with IcmRegister do
  begin
   //TCommonDialog
   Obj:=AddClass(TCommonDialog, 'TComponent');
   AddClassMethod(Obj,'function Execute: Boolean',ExecMethod);
   //TComponent
   Obj:=AddClass(TComponent, 'TPersistent');
   AddProperty(Obj,'ComponentCount','integer',GetProp,nil);
   AddIndexProperty(Obj,'Components','integer','TComponent',ExecMethod,true);
  end;
 IcmRegister.AddedBy:=nil;
end;

destructor TcmReg.Destroy;
begin
 if Assigned(IcmRegister) then IcmRegister.RemoveAddedBy(Self);
 inherited;
end;

....

initialization
 cmReg:=TcmReg.Create;

finalization
 cmReg.Free;

end.
```

В модуле объявлен класс TcmReg для регистрации стандартных классов VCL. Конструктор Create устанавливает в свойстве AddedBy ссылку на самого себя, затем производится регистрация классов и методов, после чего свойство AddedBy обнуляется. В деструкторе вызывается метод RemoveAddedBy, которому так же передается ссылка на текущий объект, все объявления связанные с этой ссылкой будут удалены из скрипта.

В качестве примеров работы с интерфейсом IcmScriptRegister, можете посмотреть исходные тексты следующих модулей:
\source\StdReg\cmStdReg.pas
\source\StdReg\cmCoolCompReg.pas
\source\ib\cmIBReg.pas

Глава



Работа с обработчиками событий

3 Работа с обработчиками событий

В ИСР CoolManager скриптовые процедуры могут использоваться в качестве обработчиков событий форм и расположенных на них компонентов, а так же объектов метаданных конфигурации. Для того, что бы скриптовую процедуру можно было использовать в качестве обработчика события, список параметров процедуры должен совпадать со списком параметров метода обработчика. Для регистрации новых обработчиков событий, в пакете cmX.bpl определен модуль cmEventMng.pas.

Модуль содержит объявление интерфейса IEventManager, используемого для доступа к менеджеру событий:

```
  // Интерфейс менеджера событий.
  IEventManager = interface
    ['{577C8021-F94E-44AA-8738-89404012446E}']
    //поиск обработчика
    function FindHandler(Obj: TObject; pInfo: PPropInfo): TObject;
    // Устанавливает обработчик для свойства pInfo объекта Obj
    // связывает его со скриптовой процедурой ProcName.
    function SetHandler(Obj: TObject; pInfo: PPropInfo; ProcName: string): Boolean;
    // Возвращает имя скриптовой процедуры связанной сос свойством pInfo объекта Obj.
    function GetHandler(Obj: TObject; pInfo: PPropInfo): string;
    // Возвращает true если скриптовая процедура (MethodName) существует.
    function MethodExists(pInfo: PPropInfo; const MethodName: string): Boolean;
    // Возвращает список процедур скрипта совместимых с типом pInfo.
    procedure GetProcNames(pInfo: PTypeInfo; Proc: TGetStrProc);
    // Выполняет скриптовую процедуру.
    function Run(const ProcedureName: String; Pars: array of const): Variant;
    // Возвращает значение параметра по номеру. Используется для чтения
    //значений var параметров обработчика события после вызова метода Run.
    function GetParam(const ProcedureName: String; i: integer): variant;
  end;
```

Класс TEventHandler, от этого класса должны наследоваться все создаваемые вами классы регистрации событий.

```
  TEventHandler = class
  protected
    FManager: IEventManager;
    FSource: TObject;
    FProcName: string;
    FPropInfo: PPropInfo;
    //фозвращает информацию о типе события
    class function GetTypeInfo: PTypeInfo; virtual;
  public
    destructor Destroy; override;
    { Менеджер обработчиков событий }
    property Manager: IEventManager read FManager write FManager;
    // Имя скриптовой процедуры
    property ProcName: string read FProcName write FProcName;
    // Ссылка на объект для которого создан обработчик события
    property Sender: TObject read FSource write FSource;
    // Ссылка на информацию о типе события
    property PropInfo: PPropInfo read FPropInfo write FPropInfo;
  published
    // Здесь должна быть определена процедура с именем " ProcHandler  "
  end;
```

А так же три процедуры:

```
// Регистрация класса обработчика событий.
procedure RegisterEventHandler(HandlerClass: TEventHandlerClass);
// Отмена регистрации класс обработчика события.
procedure UnRegisterEventHandler(HandlerClass: TEventHandlerClass);
// Поиск обработчика события для aTypeInfo.
```

```
function FindHandlerClass(aTypeInfo: PTypeInfo): TEventHandlerClass;
```

3.1 Регистрация класса обработчика события

Для создания класса регистрации обработчика события, следует выполнить следующую последовательность операций:

- Создать класс наследованный от TEventHandler
- Перкрыть метод: `class function GetTypeInfo: PTypeInfo; override;`
- В разделе Public класса создать метод с именем ProcHandler
- Зарегистрировать класс при помощи процедуры: `procedure RegisterEventHandler(HandlerClass: TEventHandlerClass);`

Ниже приведен код для создания класса регистрации обработчика TNotifyEvent:

```
type

  // TNotifyEvent
  TNotifyEventHandler = class(TEventHandler)
  protected
    class function GetTypeInfo: PTypeInfo; override;
  published
    procedure ProcHandler(Sender: TObject);
  end;

implementation

{ TNotifyEventHandler }
class function TNotifyEventHandler.GetTypeInfo: PTypeInfo;
begin
  Result := System.TypeInfo(TNotifyEvent);
end;

procedure TNotifyEventHandler.ProcHandler(Sender: TObject);
begin
  FManager.Run(FProcName, [integer(Sender)]);
end;

initialization
  RegisterEventHandler(TNotifyEventHandler);

end.
```

Пакет cmX.bpl содержит модуль cmEvents.pas, в модуле определены классы регистрации для многих обработчиков событий определенных в VCL Delphi. Можете использовать этот модуль в качестве примера создания классов регистрации.

Глава IV

Создание метаклассов

4 Создание метаклассов

Все метаклассы наследуются от класса TcmUnitComponent, объявленного в модуле cmConfigClass:

```
 TcmUnitComponent = class(TComponent)
........
  published
   property Description: string read fDescription write fDescription;
   property SourceUnit: string read fSourceUnit write fSourceUnit;
 end;
```

TcmUnitComponent наследован от TComponent, это позволяет использовать Object Inspector для редактирования свойств объектов метаданных. Принципиально, разработка метаклассов ничем не отличается от создания невизуальных компонент в Delphi, просто существует несколько несложных правил которые следует выполнять для того, что бы созданный класс мог быть использован в качестве метакласса:

- Класс должен быть наследован от TcmUnitComponent
- Зарегистрировать созданный метакласс с помощью процедуры RegisterConfigClass
- Зарегистрировать картинку для метакласса с помощью процедуры RegisterConfigBitMap

4.1 Пример создания метакласса

В качестве примера создания метакласса рассмотрим модуль cmIBComponent, модуль включен в пакет CoolIB**X**.bpl, исходные тексты модулей пакета расположены в каталоге \source\ib. В модуле объявляются два метакласса:

```
TcmIBBaseParam = class(TcmUnitComponent)
 private
  fParams: TStrings;
  fDataBaseName: string;
 public
  constructor Create(Owner: TComponent); override;
  destructor Destroy; override;
 published
  property Params: TStrings read fParams write fParams;
  property DataBaseName: string read fDataBaseName write fDataBaseName;
end;

TcmIBBase = class(TcmUnitComponent)
 private
  fDataBase: TcmIBDataBase;
  fParams: TcmIBBaseParam;
  procedure SetConnected(Data: boolean);
  function GetConnected: boolean;
  procedure SetParams;
  procedure SetParamsProp(Data: TcmIBBaseParam);
 public
  constructor Create(Owner: TComponent); override;
  destructor Destroy; override;
  function GetDataBase: TcmDataBase;
 published
  property Params: TcmIBBaseParam read fParams write SetParamsProp;
  property Connected: boolean read GetConnected write SetConnected;
end;
```

Класс TcmIBBase используется для установки соединения с InterBase базой данных, класс TcmIBBaseParam содержит параметры установки соединения с базой. TcmIBBase имеет свойство Params, ссылающиеся на объект класса TcmIBBaseParam. Оба класса наследованы от TcmUnitComponent, следовательно, создаются метаклассы для конфигуратора ИСР

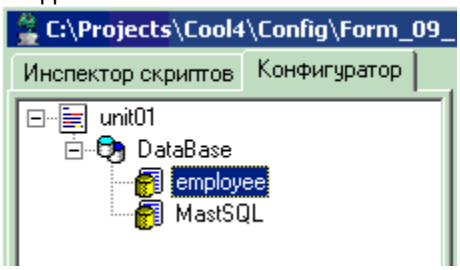
CoolManager. Нас интересует раздел инициализации модуля:

```
initialization
//регистрируем классы
RegisterClass(TcmIBBaseParam);
RegisterClass(TcmIBBase);
//редактор свойств для ссылки на параметры базы данных
RegisterPropertyEditor(TypeInfo(TcmIBBaseParam),TcmIBBase,'',TcmIBBaseProperty);
RegisterPropertyEditor(TypeInfo(TStrings), TcmIBBaseParam, '', TcmStringListProperty);
//регистрируем классы для конфигуратора
RegisterConfigClass('', TcmIBBase, false);
RegisterConfigClass('TcmIBBase', TcmIBBaseParam, false);
//регистрируем картинки компонент для конфигуратора
RegisterConfigBitMap(HInstance, 'CMIBBASE', clWhite, TcmIBBase);
RegisterConfigBitMap(HInstance, 'CMIBBASEPARAM', clWhite, TcmIBBaseParam);

finalization
//отменяем регистрацию классов для конфигуратора
UnRegisterConfigClass(TcmIBBase);
UnRegisterConfigClass(TcmIBBaseParam);

end.
```

В разделе инициализации выполняются следующие действия:

- Классы регистрируются при помощи стандартной процедуры RegisterClass.
- Регистрируются редакторы свойств при помощи процедуры RegisterPropertyEditor. Для свойства Params класса TcmIBBase регистрируется редактор TcmIBBaseProperty, редактор объявлен в этом же модуле. Редактор наследован от TcmComponentEditor и главное его отличие от родителя состоит в том, что при формировании выпадающего списка, поиск подходящих компонент ведется только среди подчиненных метаобъектов, а не во всей конфигурации.
- Регистрируются метаклассы конфигурации с помощью процедуры RegisterConfigClass, процедуре передаются три параметра: имя метакласса-владельца, указатель на класс регистрируемого компонента и флаг уникальности метакласса в списке (будет описан в следующем разделе). Для TcmIBBase владелец не указан, следовательно, метаобъекты этого класса будут создаваться как подчиненные модулю конфигурации (TcmUnitComponent), метаобъекты класса TcmIBBaseParam будут создаваться как подчиненные объектам класса TcmIBBase.
- Далее регистрируются картинки для отображения перед именем метаобъекта в конфигураторе. Сами картинки расположены в файле ресурсов cmIBBase.res, имеют размер 16x16 и подключаются к модулю с помощью дерективы {$R cmIBBase.res}, расположенной сразу же после обявления implementation секции модуля.

В разделе finalization вызываются процедуры UnRegisterConfigClass для отмены регистрации метаклассов конфигуратора. Вот собственно и весь код, который потребовалось написать для того, что бы подключить метаклассы к ИСР CoolManager. Использование метаклассов объявленных в пакете CoolIB описывается в руководстве разработчика, в конфигураторе, созданные на их основе метаобъекты имеют следующий вид:

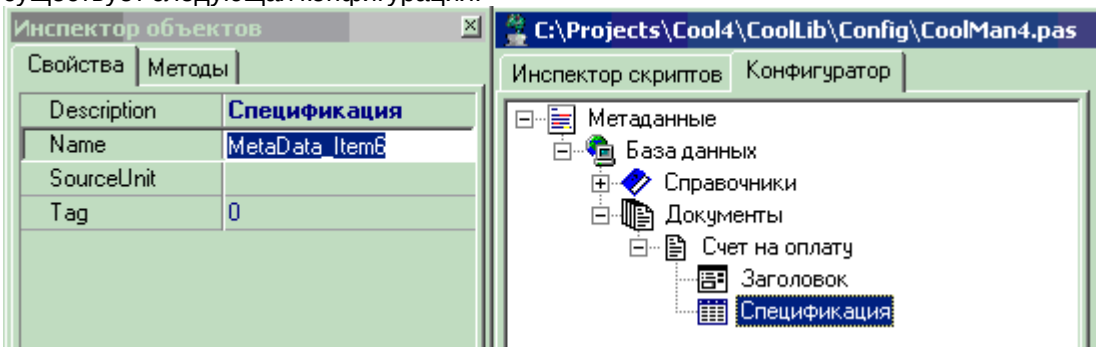


4.2 Отражение метаобъекта в конфигураторе.

В предыдущем разделе была описана процедура RegisterConfigClass, используемая для регистрации метакласса:

```
procedure RegisterConfigClass(ParentName: string; AClass: TComponentClass; AUnique: boolean);
```

если задать третий параметр процедуры равным true, то в списке подчиненных объектов можно будет создать только один объект данного класса. Поясним на примере. Предположим существует следующая конфигурация:



Конфигурация описывает структуру метаобъектов прикладной задачи учета, задается описание справочников и документов. Существуют следующие определения метаклассов:

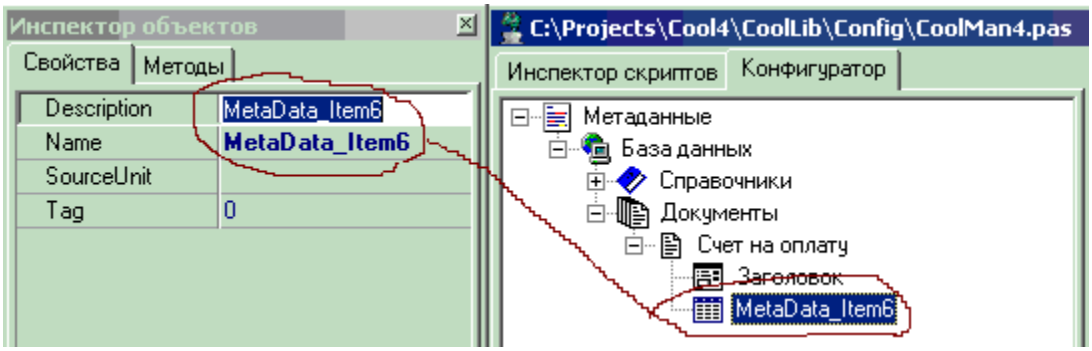
```
//документ
TcmDoc = class(TcmUnitComponent)
 .......
end;

//заголовок документа
TcmDocHead = class(TcmUnitComponent)
 .......
end;

//спецификация документа
TcmDocSpec = class(TcmUnitComponent)
 .......
end;
```

Метакласс документа имеет два подчиненных метакласса - заголовок и спецификацию, для каждого документа должен быть создан только один объект заголовка и один объект спецификации. Тогда, процедуры регистрации метаклассов должны иметь следующий вид:

```
RegisterConfigClass('TcmDocList', TcmDoc, false);
RegisterConfigClass('TcmDoc', TcmDocHead, true);
RegisterConfigClass('TcmDoc', TcmDocSpec, true);
```

обратите внимание на то, что при регистрации метаклассов TcmDocHead и TcmDocSpec третьим параметром передается значение true. Это обозначает, что метаобъект TcmDoc может иметь только по одному экземпляру подчиненных метаобъектов класса TcmDocHead и TcmDocSpec. Если документ имеет подчиненный объект класса TcmDocSpec, второй объект данного класса создать уже будет нельзя.

При создании нового объекта в конфигурации, свойству Description, по умолчанию, присваивается сгенерированное имя объекта:



Это не всегда удобно, например, если мы создаем объект класса TcmDocSpec, то желательно, что бы свойству Description сразу же присваивалось значение 'Спецификация'. Для этого следует перекрыть виртуальный метод CreateDescription:

```
type
TcmDocSpec = class(TcmUnitComponent)
  public
   procedure CreateDescription; override;
 end;

implementation

{ TcmDocSpec }

procedure TcmDocSpec.CreateDescription;
begin
 Description:='Спецификация';
end;
```

Кроме этого, метод CreateDescription может быть перекрыт для выполнения дополнительных операций при вставке нового объекта в конфигурацию.

4.3 Редакторы свойств метаклассов

Так как метаклассы являются стандартными Delphi комопнентами, программисту следует уделять серьезное внимание проблеме создания удобных и многофункциональных редакторов свойств и компонент для метаклассов. Т.к. в конечном счете, эффективность работы разработчика прикладной задачи определяется тем, насколько мощным и удобным инструментарием он располагает.
В модуле cmConfigClass объявлен редактор свойств TcmComponentProperty:

```
  TcmComponentProperty = class(TPropertyEditor, IReferenceProperty)
  protected
    function GetComponentReference: TComponent;
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure SetValue(const Value: string); override;
  end;
```

Редактор используется для редактирования свойств компонентного типа вместо стандартного TComponentProperty. TComponentProperty формирует выпадающий список для выбора компонента. При формировании списка, производится поиск подходящих объектов среди всех компонент расположенных на форме-владельце, а так же формах перечисленных в операторе uses модуля. TcmComponentProperty подобен TComponentProperty, только поиск подходящих компонент производится среди метаобъектов конфигурации.

4.4 Доступ к метаобъектам из модулей пакета

При создании служебных пакетов можно получить доступ к метаобъектам конфигурации при помощи интерфейса ICoolConfig:

```
ICoolConfig = interface
 ['{275EC69B-1BF9-4EA3-A43D-10726AF710C5}']
 function GetComponentByName(Name: string): TComponent;
 procedure GetComponentList(AClass: TClass; List: TList);
 function IsComponent(Comp: TComponent): boolean;
 function GetUnitCount: integer;
 function GetUnitByIndex(Index: integer): TComponent;
end;
```

`function GetComponentByName(Name: string): TComponent;` Возвращает метаобъект конфигурации по имени компонента.
`procedure GetComponentList(AClass: TClass; List: TList);` В переменной List возвращает список указателей на метообъекты принадлежащие классу AClass.
`function IsComponent(Comp: TComponent): boolean;` Возвращает true, если в конфигурации найден указанный метаобъект.
`function GetUnitCount: integer;` Возвращает число модулей конфигурации.
`function GetUnitByIndex(Index: integer): TComponent;` Возвращает ссылку на модуль конфигурации по порядковому номеру.

Интерфейс ICoolConfig описан в модуле cmConfigClass, там же содержится описание переменной для доступа к конфигурации:

```
var IConfig: ICoolConfig;
```

Переменная IConfig содержит ссылку на интерфейс конфигурации загруженной в ИСР.

4.5 Заключение

Ну вот и все, что мы планировали рассказать о создании служебных пакетов для ИСР CoolManager. Главной задачей, которая ставилась при проектировании системы CoolManager, была максимальная открытость среды разработки и предельно простой код, который требуется написать для создания метаклассов и регистрации новых элементов скриптового языка. Насколько это удалось, судить тем программистам, которые будут создавать служебные пакеты для ИСР CoolManager.
Большая часть пакетов ИСР поставляется в исходных текстах, и может быть использована в качестве примеров к изложенному в руководстве материалу. ИСР постоянно совершенствуется и развивается, по мере внесения изменений, редактируется система помощи и сопроводительная документация прилагаемая к системе.