Abstract: *A follow on paper on the "Dynamic Packages" (#27178) paper on how to create and use dynamic packages in Delphi.*

Advanced Dynamic Packages

A follow-on to the "Dynamic Packages" whitepaper by Vino Rodrigues
vino@vinorodrigues.com

What will be covered in this paper:

Foreword

Since publishing my paper "Dynamic Packages" on community.borland.com I've had many a mail from the Delphi community asking certain how-to's. Out of all the hundreds of questions, three were asked more often than not, namely:

1. How do I call my own class methods (function and procedures) in a dynamically loaded class?
2. How do I call functions and procedures not in a class?
3. You mentioned a "registration mechanism" - can you give us an example?

I started answering those questions one for one – but later it become overwhelming. So I've decided to write this addition to that paper, where I will try to answer those questions.

Calling Custom Methods

A common question I got was:

> *"I have a form with a procedure called 'SomeInit' (where I do some initialization work), but, if I dynamically load and create the form I can't call it?"*

Now there's two ways of doing this.

Class Inheritance

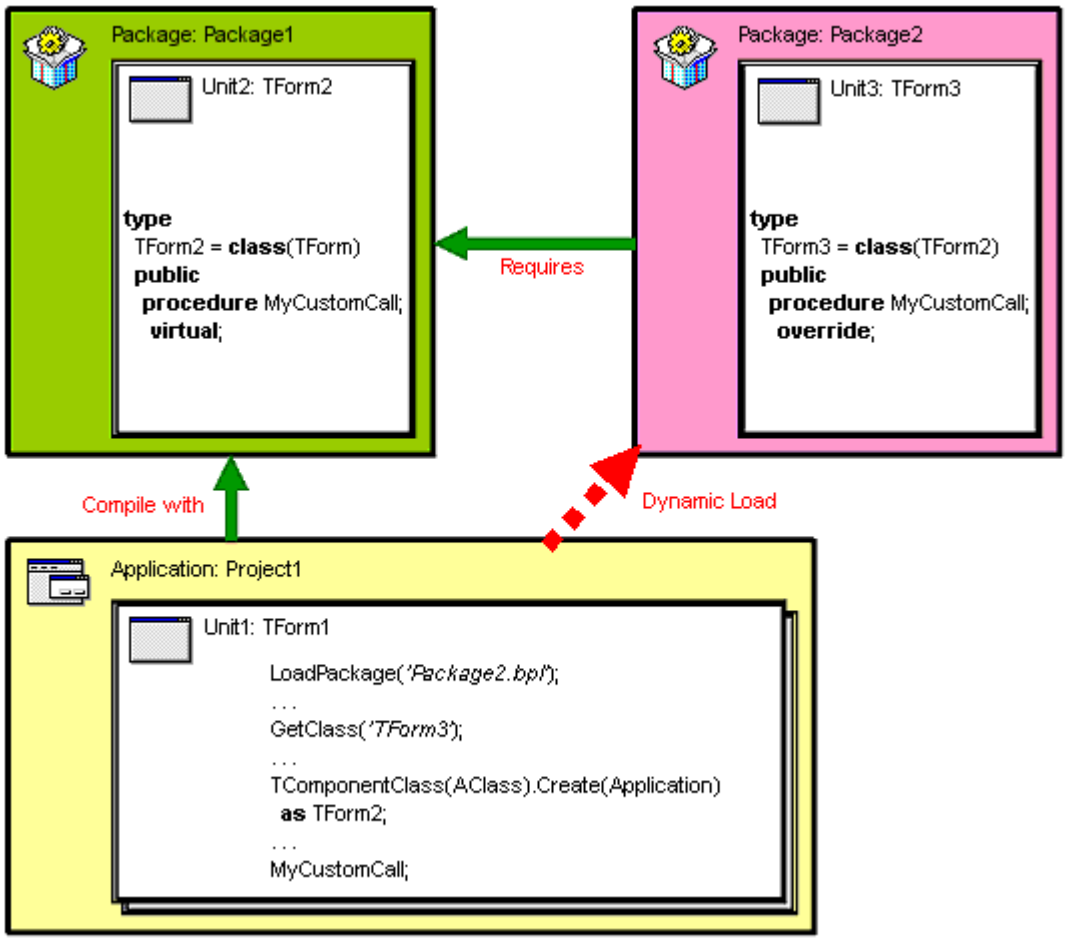
The first way to call your methods is to first define a base class that holds your method calls. This is the more traditional way of doing thing and generally works for all flavours of Delphi.
For the purpose of illustration we'll create a new form that we use as template called TForm2 in Unit2. Here we add custom methods as blank virtual methods (methods with no actual code in them) or as abstract virtual – in our example it's a procedure called "MyCustomCall". This unit and form is then compiled into a package – in this case called Package1. This package then becomes our common package – our application will "compile with" it, and the dynamically loaded package will "require" it.
Secondly we'll create a new form (TForm3 in Unit3) that inherits from the template form (TForm2) and override the custom methods. We will also remember to call 'RegisterClass(TFrom3)' in the unit's initialization section. We then compile this unit and form into a new package that we will dynamically load (Package2).
The application will then "compile with" Package1 and dynamically load Package2. To use our custom method we do something like this:

```
var
  PackageModule: HModule;
  AClass: TPersistentClass;
begin
  PackageModule := LoadPackage('Package2.bpl');
  if PackageModule <> 0 then
  begin
    AClass := GetClass('TForm3');  // create inherited class
```

```
    if AClass <> nil then
      with TComponentClass(AClass).Create(Application)
        as TForm2 do  // type as base class
      begin
        MyCustomCall;  // the custom call here
        ShowModal;
        Free;
      end;

    UnloadPackage(PackageModule);
  end;
end;
```

Our application schematically will look something to the effect of:



This is a relatively simple way to achieve a common way of calling our dynamically loaded forms – it sort of forces you to stick to a certain naming and calling convention in your methods – but alas, that is a good thing. After all, the component model is just that – a common way to do similar things.

Interfaces

The second way to call your methods is to use interfaces to define common method calls, and then creating classes that then support those interfaces. With interfaces we have far more variation in what our classes can do since a class (form) can implement more that one interface. This is now the preferred way of doing things, but only the latest flavours of Delphi support it.
For the purpose of illustration we'll create a unit called Unit2. Here we add a new interface (in our example we'll call it 'IMyCustomCalls') where we write the declarations of our methods – in our example it's a procedure called "MyCustomCall". We should also give the interface a GUID (use Shift+Ctrl+G to let Delphi generate one). You'll need the GUID because Delphi's internal interface manager uses this GUID as an index to a list of interfaces that it can type-cast. If you exclude the GUID you will not be able to call the 'Supports' function nor do a 'as' type-cast. It should look something like this:

```
type
  IMyCustomCalls = interface
    ['{D2945EC7-17C4-446F-9DF0-012069D07CC6}']
    procedure MyCustomCall;
  end;
```

This unit is then compiled into a package – in this case called 'Package1'. This package then becomes our common package – our application will "compile with" it, and the dynamically loaded package will "require" it.
Secondly we'll create a new form (TForm3 in Unit3) that implements our new interface and write in our custom methods. Regardless of packages or not we need to call 'RegisterClass(TFrom3)' in the unit's initialization section because the interface mechanisms of Delphi require it . We then compile this unit and form into a new package that we will dynamically load (Package2).
The application will then "compile with" Package1 and dynamically load Package2. To use our custom method we do something like this:

```
var
  PackageModule: HModule;
  AClass: TPersistentClass;
  AForm: TForm;
  IForm: IMyCustomCalls;
begin
  PackageModule := LoadPackage('Package2.bpl');
  if PackageModule <> 0 then
  begin
    AClass := GetClass('TForm3');

    if AClass <> nil then
    begin
      AForm := TComponentClass(AClass).Create(Application)
        as TForm;  // create as base class, like normal

      if Supports(AForm, IMyCustomCalls) then
        IForm := AForm as IMyCustomCalls;

      if Assigned(IForm) then IForm.MyCustomCall;

      AForm.ShowModal;

      if Assigned(IForm) then IForm := nil;

      AForm.Free;
    end;
    UnloadPackage(PackageModule);
  end;
end;
```

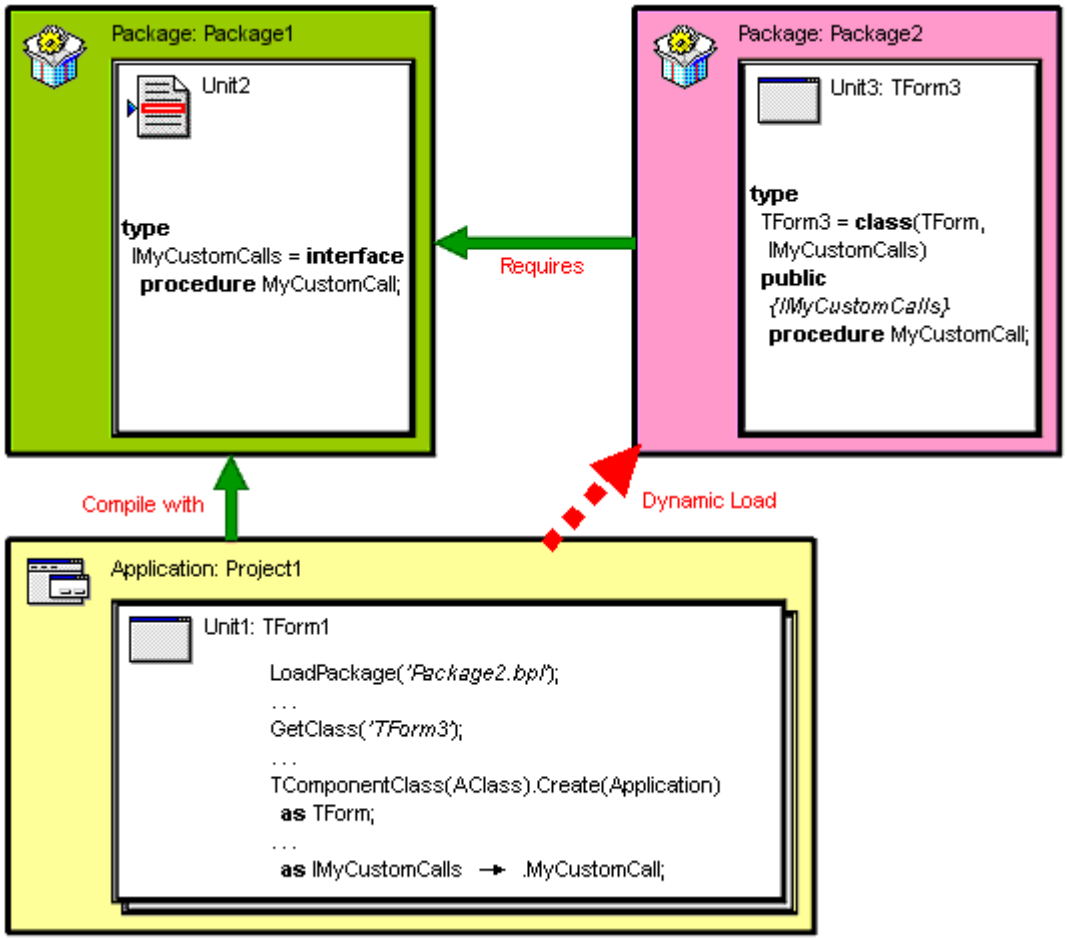
Just a few notes:

i. We must create a new interface reference (in this case the var 'IForm', otherwise '_Release' never gets called enough times, and the reference object will stay in memory and cause an access violation after you call Free on the object reference (AForm).

ii. This is not the correct way of using interface references – we should have created the form object directly on the interface reference 'IForm' (not the object reference 'AForm') – but then we wouldn't be able to call the form's standard functions like 'ShowModal'. So the work-around is to create an object reference and the copy it into a interface reference (IForm := AForm as IMyCustomCalls;)

iii. Because we created a copy of the reference we must remember to destroy this copy before we destroy the actual object. We can do this by setting the interface reference to nil (IForm := nil;), this will force a '_Release' to be called and the reference is then voided.

Our application schematically will look something to the effect of:



As we can see, this is a far more flexible way of doing thing – giving us two major benefits.

1. We can utilize more than one interface – thus giving us unlimited flexibility in the functionality of our dynamically loaded classes.
2. We can test a loaded class (with 'Supports') to see if it supports a interface before we actually use that interfaces functionality – this gives us the ability to write generic execution code and we don't have to relay on forcing any particular class or it's descendants to be loaded.

Calling Standard Functions and Procedures

Another very common question I got was:

> *"Loading and calling functions in a class is cool, but how do I call a standalone function in a package?"*

Firstly: *WHAT, ON EARTH, ARE YOU STILL USING PROCEDURE ORIENTED PROGRAMMING FOR!?*
But let's be fair – there is still a use for this type of thing. And, no surprises, there are two ways of doing this.

Using the Package as a DLL

A package is a DLL. Maybe a bit glorified, but it is still just a DLL. And as such, the LoadPackage function is just a glorified LoadLibrary function.
So to use a function or procedure in a dynamically loaded package we just need to 'export' the function in the unit that will compile into the DLL.
All we need to do is declare our function in interface section of a unit and add "export" to the end of it. Then add "exports OutFunctionName" just below the function declaration.

```
// Exporting the traditional way (DLL way)
```

```
procedure ProC00; export;

exports
  Proc00;
```

NB: I've mixed the case to show that the method names are case sensitive.
We then can call it by using the standard GetProcAddress function:

```
type
  TProc00 = procedure;
var
  PackageModule: HModule;
  Proc00: TProc00;
begin
  PackageModule := LoadPackage('Package1.bpl');
  if PackageModule <> 0 then
  begin
    @Proc00 := GetProcAddress( PackageModule, 'ProC00' );
    if @Proc00 <> nil then
      Proc00;

    UnloadPackage(PackageModule);
  end;
end;
```

That's the rather simple way of calling standard functions and procedures in a packages. It does however require you to physically export those methods in the source.

> **DID YOU KNOW**: You can even export methods in EXE's and have other EXE's use them. This is because in Windows an EXE is just a DLL with a 'WndMain' exported function.

Using the Package Internal Naming Convention

When you compile a package it stores all units, initialization, finalization, classes and methods as exported functions. Another way to call a function or procedure is to tap into those exported functions directly. Fortunately these exported functions follow a very strict naming convention. By using a module viewer (like QuickView) we can look at the export table of the package to find the naming format of the function we are tying to access.
NOTE: This is not a documented way of calling methods in packages, and thus may change over different versions of Delphi. Using this technique is done at your own risk and will not be supported either by myself or Borland. This example uses Delphi 6.
In my example I've created several functions and procedures and then compiled them into a package:

```
// The "Calling conventions"
procedure ProC01; {register;}  //this is the default
procedure ProC02; pascal;
procedure ProC03; cdecl;
procedure ProC04; stdcall;

// Function calls
function FunC01: Integer;
function FunC02: Byte;
function FunC03: Word;
function FunC04: Real;
function FunC05: Char;
function FunC06: String;
function FunC07: Boolean;

// Parameters
function FunC08(i: Integer): Integer;

procedure ProC05(i: Integer);
```

```
procedure ProC06(shi: Shortint);
procedure ProC07(smi: Smallint);
procedure ProC08(i: Integer);
procedure ProC09(li: Longint);
procedure ProC10(i64: Int64);
procedure ProC11(b: Byte);
procedure ProC12(w: Word);
procedure ProC13(lw: Longword);
procedure ProC14(c: Cardinal);
procedure ProC15(r: Real);
procedure ProC16(r48: Real48);
procedure ProC17(sgl: Single);
procedure ProC18(d: Double);
procedure ProC19(e: Extended);
procedure ProC20(cmp: Comp);
procedure ProC21(cur: Currency);
procedure ProC22(ch: Char);
procedure ProC23(s: String);
procedure ProC24(sc: PChar);
procedure ProC25(l: Boolean);

procedure ProC26(
  shi: Shortint; smi: Smallint; i: Integer; li: Longint;
  i64: Int64;
  b: Byte;
  w: Word; lw: Longword; c: Cardinal;
  r: Real; r48: Real48; sgl: Single; d: Double; e: Extended;
  cmp: Comp; cur: Currency;
  ch: Char;
  s: String; sc: PChar;
  l: Boolean);

procedure ProC27(Form: TForm);

// Parameter semantics
procedure ProC28(i: Integer);
procedure ProC29(var i: Integer);
procedure ProC30(const i: Integer);
procedure ProC31(out i: Integer);
```

Viewing this packages export table will show:

Export Table

Name:	*Package1.bpl*
Characteristics:	00000000
Time Date Stamp:	00000000
Version:	0.00
Base:	00000001
Number of Functions:	00000032
Number of Names:	00000032

Ordinal	Entry Point	Name
0031	000018dc	@GetPackageInfoTable
002e	000018ac	@Package1@@GetPackageInfoTable$qqrv
002d	000018a4	@Package1@@PackageLoad$qqrv
002c	0000189c	@Package1@@PackageUnload$qqrv
002b	00001894	@Package1@initialization$qqrv
0027	0000188c	@UNIT2@PROC02$QV

```
0001    00001884    @Unit2@Finalization$qqrv
0024    00001844    @Unit2@FunC01$qqrv
0023    0000183c    @Unit2@FunC02$qqrv
0022    00001834    @Unit2@FunC03$qqrv
0021    00001804    @Unit2@FunC04$qqrv
0020    000017fc    @Unit2@FunC05$qqrv
001f    000017f0    @Unit2@FunC06$qqrv
001e    000017e4    @Unit2@FunC07$qqrv
001d    000017d8    @Unit2@FunC08$qqri
0028    000017c0    @Unit2@ProC01$qqrv
0026    000017b4    @Unit2@ProC03$qv
0025    000017a8    @Unit2@ProC04$qqsv
001c    000017a0    @Unit2@ProC05$qqri
001b    00001798    @Unit2@ProC06$qqrzc
001a    00001790    @Unit2@ProC07$qqrs
0019    00001788    @Unit2@ProC08$qqri
0018    0000177c    @Unit2@ProC09$qqri
0017    00001774    @Unit2@ProC10$qqrj
0016    0000176c    @Unit2@ProC11$qqruc
0015    00001764    @Unit2@ProC12$qqrus
0014    0000175c    @Unit2@ProC13$qqrui
0013    00001754    @Unit2@ProC14$qqrui
0012    0000174c    @Unit2@ProC15$qqrd
0011    00001744    @Unit2@ProC16$qqr6Real48
0010    0000171c    @Unit2@ProC17$qqrf
000f    00001714    @Unit2@ProC18$qqrd
000e    000016fc    @Unit2@ProC19$qqrg
000d    000016f4    @Unit2@ProC20$qqr11System@Comp
000c    000016ec    @Unit2@ProC21$qqr15System@Currency
000b    000016e4    @Unit2@ProC22$qqrc
000a    000016dc    @Unit2@ProC23$qqr17System@AnsiString
0009    000016d4    @Unit2@ProC24$qqrpc
0008    000016cc    @Unit2@ProC25$qqro
0007    000016c4    @Unit2@ProC26$qqrzcsiijucusuiuid
          6Real48fdg11System@Comp15System@Currencyc
        17System@AnsiStringpco
0006    000016bc    @Unit2@ProC27$qqrp11Forms@TForm
0005    000016bc    @Unit2@ProC28$qqri
0004    00001a74    @Unit2@ProC29$qqrri
0003    00001a68    @Unit2@ProC30$qqrxi
0002    00001a5c    @Unit2@ProC31$qqrri
0000    00001a54    @Unit2@initialization$qqrv
002f    00001a68    Finalize
0030    00001a5c    Initialize
```

Scrutinising this table we can assume the following naming convention:

```
@[UnitName]@[MethodName]$[CallingConvention][ParameterType(
s)]
```

Where:
UnitName
The name of the unit in which the method resides. Case sensitive (except in 'pascal' calling convention, see below).
MethodName
The name of the function or procedure. Case sensitive (except in 'pascal' calling convention, see below).
Calling Convention
(Case sensitive)

"qqr"	'regsiter' – default calling convention
"Q"	'pascal' – the unit name and procedure name also become all uppercase
"q"	'cdecl'

"qqs"	'stdcall'

ParameterType(s)
Here is a list of the more common types:

"v"	the method does not have any parameters (means 'Void') - **NB**: You must include this if the method has not parameters.
"zc"	ShortInt
"s"	SmallInt
"i"	Integer
"i"	Longint
"j"	Int64
"uc"	Byte ('Unsigned Char')
"us"	Word ('Unsigned Short')
"ui"	Longword ('Unsigned Integer')
"ui"	Cardinal
"d"	Read ('Decimal')
"6Real48"	Real48 – the number 6 represents the length of the description followed by the Delphi description
"f"	Single
"d"	Double
"g"	Extended
"11System@Comp"	Comp
"15System@Currency"	Currency
"c"	Char
"17System@AnsiString"	String
"pc"	PChar
"o"	Boolean

If you want to pass any other type you'll need to name it using it's unit name and class name. For example: if you want to pass a TForm then it would be "11Forms@TForm" – where "11" is the length of the description, "Forms" the unit name followed by "@" followed by the class name "TForm".

Parameter Semantics
The way you pass parameters must also be taken into consideration. You will also need to prefix all parameters types with parameter semantics identifiers:

`X`		(No prefix)
`var` `x`	"r"	Reference
`const` `x`	"x"	
`out` `x`	"r"	

For example: if you have a "var a: Integer" as parameter then the complete parameter type will be "ri".

Example
An example of using this method of calling methods would be:

```
type
  TProc23 = procedure(s: String);
```

```
var
  PackageModule: HModule;
  Proc23: TProc23;
begin
  PackageModule := LoadPackage('Package1.bpl');
  if PackageModule <> 0 then
  begin
    @Proc23 := GetProcAddress( PackageModule,
      '@Unit2@ProC23$qqr17System@AnsiString' );
    if @Proc23 <> nil then
      Proc23('Hello World');

    UnloadPackage(PackageModule);
  end;
end;
```

Note: I chose the string example for two reasons. a) it's the longest exported name, and b) to show that, if you are using packages, you do not need to include the 'ShareMem' unit in your application which is used to load the BORLNDMM.DLL shared memory manager.

Obtaining Package Information

This topic is not one anybody has asked me about yet. But, alas, I feel it is important enough to discus, since it will help us in the next topic.
Obtaining package information involves using certain package support routines that will assist us in getting information about a package – like its description, what units it contains and what other packages it may require.

Listing Packages

Listing all the package that your application is currently using is done with one of (or both) two functions.
`EnumModules` – which enumerates the executable and all packages in an application by passing the handle of the executable and then of each package, in turn, to a user-defined callback function. This callback is of type TEnumModuleFunc which receives an instance handle and a user-defined data value for every module enumerated. EnumModules continues until the last module in the application is enumerated, or until the callback function returns False.
Or
`EnumResourceModules` – which executes the callback for all of the instance handles associated with resources of the current program. This is mostly used if you have different language modules been loaded by Delphi's Translation Tools.
Example:

```
  // TEnumModuleFunc
  function MyEnumModuleFunc(HInstance: Integer;
    Data: Pointer): Boolean;
  var
    FN: array[0..MAX_PATH] of Char;
  begin
    GetModuleFileName(HInstance, FN, SizeOf(FN)-1 );
    GS := GS + '"' + StrPas(@FN) + '"' + #13#10;
    Result := True;
  end;

var
  Dummy: Pointer;
begin
  Dummy := nil;
  GS := '';  // GS is a global typed string
  EnumModules(TEnumModuleFunc(@MyEnumModuleFunc), Dummy);
  Memo1.Lines.Text := GS;
end;
```

Listing Package Information

By calling GetPackageInfo we can process the information in a package's information table. This is done by passing this call a TPackageInfoProc type, which is a pointer to a callback which gets called once for each information element returned (that is for every unit included in the package and for every package required by the package).
Example:

```
  // TPackageInfoProc
  procedure MyPackageInfoProc(const Name: string;
    NameType: TNameType; Flags: Byte; Param: Pointer);
  begin
    case NameType of
      ntContainsUnit:
        GS := GS + 'Contains "' + Name + '"'#13#10;
      ntRequiresPackage:
        GS := GS + 'Requires "' + Name + '"'#13#10;
      ntDcpBpiName:
        GS := GS + 'Named "' + Name + '"'#13#10;
    end;
  end;

var
  PackageModule: HModule;
  Param: Pointer;
  Flags: Integer;
begin
  PackageModule := GetModuleHandle('vcl60.bpl');
  if PackageModule <> 0 then
  begin
    Param := nil;
    GS := '';  // GS is a global typed string
    GetPackageInfo(PackageModule, Param, Flags,
      @MyPackageInfoProc);
    Memo2.Lines.Text := GS;
  end;
end;
```

You can also extract the package description by using the GetPackageDescription function, which will obtain the description stored with the named package. If the package does not have a description resource, GetPackageDescription returns an empty string.
Example:

```
begin
  Label1.Caption := GetPackageDescription('vcl60.bpl');
end;
```

Obtaining Knowledge of Class Names

Toward the end of my article on dynamic packages I say:

> *An application requires "knowledge" of the registered class names prior to loading the package. One way to improve this would be to create a registration mechanism to inform the application of all the class names registered by the package.*

Quite a few of the queries I got was to give an example of how this would work.

Use the registry

If we look at what we need we will find that only three things are required: 1) The package file name, b) the form or class name we need to load, and finally 3) a place to load it form (e.g. a menu item). Closer inspection tells us that all three values are strings - which lead to the simplest way: To create a registration mechanism that uses the registry (or an INI file) to list all the requires data.
Delphi itself uses the "HKCUSoftwareBorlandDelphi6.0Known Packages" to store its list of dynamically loaded packages that contain components and wizards.

Self Registration

Another way to achieve this is through a self registration mechanism. What do I mean self registration? Let me explain with a side-by-side example - I'll use Delphi's own component registration to explain a practical example.
Those of you who have created your own components for Delphi know that a component doesn't just magically appear on the component palette. You need to write a "Register" function and in it call a "RegisterComponents" function that registers all the components that you wish to add to the palette.
I've created a simple example (that you can download with this paper) that uses a similar approach. Here is the logic:

a. I create a unit ('Unit2') that handles all the registration. This unit I place in a common package ('Package1') that my applications will then 'compile with' and all my dynamically loaded packages will 'require'.

b. In this unit I have a process starting procedure ('BuildDynamicFormMenu') that my application will call only once to invoke the registration mechanism. In my case I will be using a main menu to create dynamic instances of dynamically loaded forms.

c. The procedure then reads in a list of packages to load. In my case I first find out which packages are already loaded and then used the 'FindFirst' WinAPI do create a list of .BPL's in the applications directory.

d. Next the procedure will load each package in turn and obtain a list of all it's units. With this list it then try's to find the 'Register' method ('FormRegister'), and then invokes this method.

e. The 'Register' method, invoked on the loaded package, will then in turn call back to the common unit a 'RegisterClass' method ('RegisterForm'), which in my example will then build the menu and store an action with the package name and the class name to be created.

Simple - hey?
Some of you may wonder why the "call-to-callback" methodology - why not just register the class name in the unit's 'initialization' section. You can, if you where to load the package and keep it in memory. In my case I load, register and unload the packages individually. I do this to save computer resources (memory) and I don't want to re-register every time I load the package.
Download the example source code of this paper here.