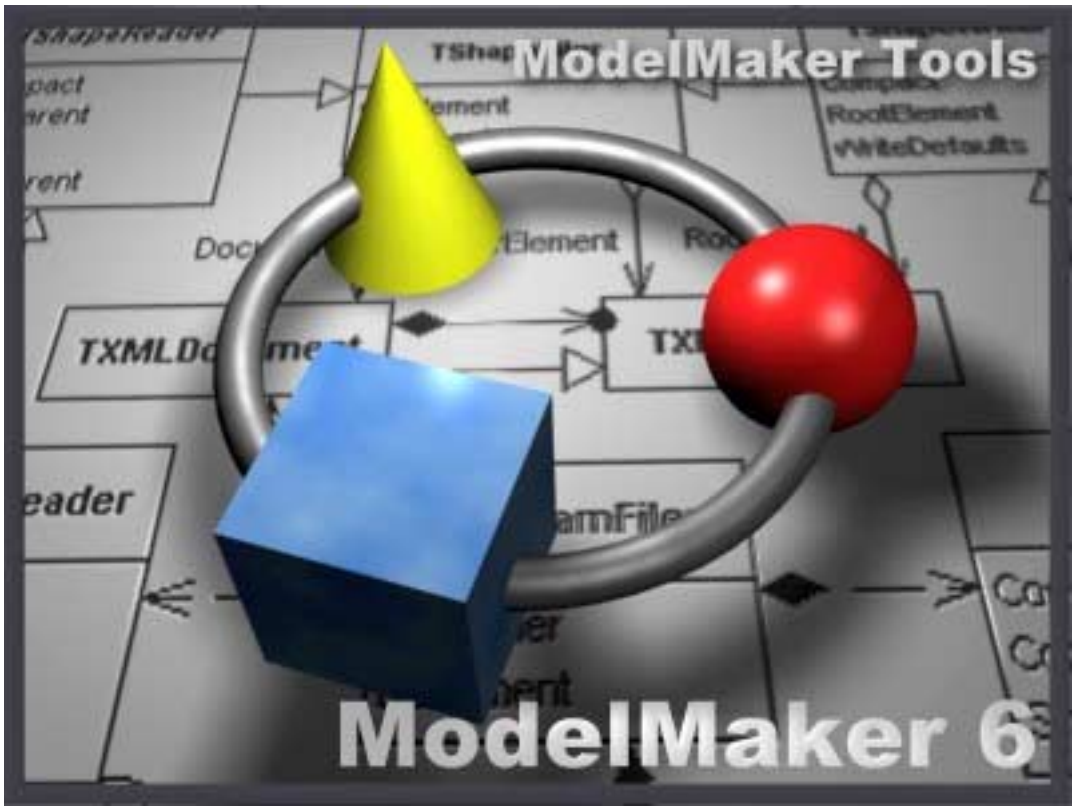


User Manual
ModelMaker 6.20

ModelMaker Tools
Stenenkruis 27 B
6862 XG Oosterbeek
The Netherlands

http:\\www.modelmakertools.com
info@modelmakertools.com

http:\\www.modelmaker.demon.nl
info@modelmaker.demon.nl

ModelMaker Tools
Stenenkruis 27 B
6862 XG Oosterbeek
The Netherlands

http:\\www.modelmakertools.com
info@modelmakertools.com

http:\\www.modelmaker.demon.nl
info@modelmaker.demon.nl

This user manual focuses on essentials and how things are done in ModelMaker. A GUI reference is available as context sensitive help file. This contains the latest GUI details. The Design Patterns manual focuses on ModelMaker's Design patterns and contains another step by step demo.

Author: G. Beuze

Contents

Introduction

ModelMaker represents a brand new way to develop classes and component packages for Borland Delphi 1-6. ModelMaker is a two-way class tree oriented productivity**,** refactoring and UML-style CASE tool specifically designed for generating native Delphi code (in fact it was made using Delphi and ModelMaker). Delphi's Object Pascal language is fully supported by ModelMaker. From the start ModelMaker was designed to be a smart and highly productive tool. It has been used to create classes for both real-time / technical and database type applications. ModelMaker has full reverse engineering capabilities.

ModelMaker supports drawing a set of UML diagrams and from that perspective it looks much like a traditional CASE tool. The key to ModelMaker's magic, speed and power however is the active modeling engine which stores and maintains all relationships between classes and their members. Renaming a class or changing its ancestor will immediately propagate to the automatically generated source code. Tasks like overriding methods, adding events, properties and access methods are reduced to selecting and clicking.

The main difference between ModelMaker and other CASE tools is that design is strictly related to and native expressed in Delphi code. This way there is a seamless transition from design to implementation currently not found in any other CASE tool. This approach makes sure your designs remain down to earth. The main difference between ModelMaker and other Delphi code generators are it's high level overview and restructuring capabilities letting you deal with complex designs.

A unique feature, currently not found in any development environment for Delphi, is the support for design patterns. A number of patterns are implemented as 'ready to use' active agents. A ModelMaker Pattern will not only insert Delphi style code fragments to implement a specific pattern, but it also stays 'alive' to update this code to reflect any changes made to the design

As a result, ModelMaker lets you:

- Speed up development
- Produce designs and code of unequaled quality.
- Think of designing code instead of typing code.
- Design without compromising.
- Refine and experiment with your designs until they are just right.
- Create and maintain magnitudes larger models in magnitudes less time.
- Document you designs in UML style diagrams.
- Document your units in help files by clicking a single button.

- In short: *save time and money, making better software.*

Installation

For installation details, refer to the readme.txt which is part of all ModelMaker distribution archives. We suggest you read this file before installing. The readme.txt also contains the latest information available on precautions related to upgrading.

ModelMaker requires Windows 95/98/ME/2000 or Windows NT 4.0 Both Borland Delphi and ModelMaker use a lot of resources. This might lead to problems under resource limited systems as Win95/98/ME.

ModelMaker is designed to work on a high resolution monitor (800x600 or better).

Contacting ModelMaker Tools

We at find it important to support you in your use of ModelMaker all the ways we can. You can find ModelMaker on the Internet at http:\\www.modelmakertools.com or alternatively http:\\www.modelmaker.demon.nl At this web site:

- We'll update you on the most recent news concerning ModelMaker and it's development.
- We'll have the latest demo versions available.
- You can consult the Tips, FAQ pages.
- You'll find links to the web-based ModelMaker newsgroups.
- You can leave hints or requests for future versions of ModelMaker.
- You can report bugs.

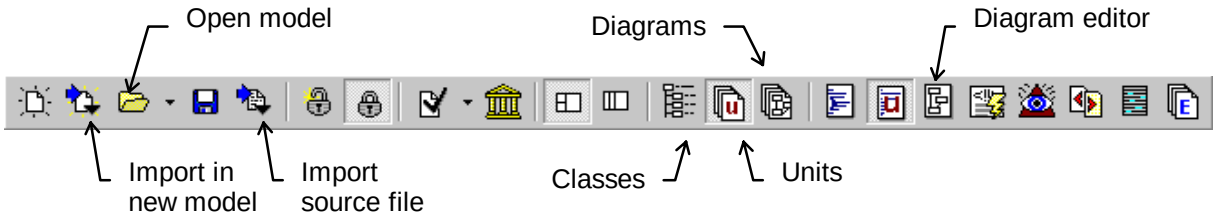
All 'how to do this' are best asked in the ModelMaker newsgroups. For other questions, the address to contact us is: info@modelmakertools.com or info@modelmaker.demon.nl.

Getting started

Getting a first impression

Here are some examples to get you started without reading lots of text. The development model will be explained in detail in the next chapters.

Loading an example model



To get a first impression of what ModelMaker is capable of, load the model ..\ModelMaker\6.0\Demos\MMToolsApi.mpb. It contains the interfaces making up ModelMaker's open tools API. In diagrams the relations between the interfaces defining this API are visualized. The Classes view shows the inheritance relations of this unit.

Then, to see how ModelMaker treats classes and units, use the 'Import source file in new Model' command from the toolbar. This will create a new model and import a Delphi unit (such as a form unit or a VCL unit). The model is named after the unit: Importing unit1.pas will create model unit1.mpb. Note the use of source aliases in the popup menu associated with the tool button. Source aliases are used to locate the source file and will be explained in chapter Source Aliases, page 52

Visualizing existing code

Visualizing existing code is a also good way too of getting started with ModelMaker. To visualize code:

1. Import the units containing the classes to visualize. Use the 'import source file" tool button in the main toolbar or drag drop source files on the 'unit's view' (View|Units)
2. Create or select a new class diagram in the 'diagrams' view (View|Diagrams)
3. In the Diagram editor (View|Diagram Editor) select the visualization wizard from the Wizard popup-up sub-menu.
4. Use this wizard to select the classes and interfaces to visualize and the kind of relations to visualize (inheritance, uses, supports etc.)
5. Completing the wizard gives you an instant diagram of the code just imported.

You might want to move around classes or interfaces (Drag move) or select different display options for classes or interfaces (Double click on the symbol) or the diagram as a whole

(Double click in empty space). Try to turn on and off member display, select interface style etc.

Creating code with ModelMaker, overview

A ModelMaker model contains a Code Model and Diagrams. The Code Model contains the classes, class members (properties, methods), units etc. that map to the corresponding entities in Delphi's Object Pascal. Diagrams are used to visualize aspects of the code model or entities that do not exists in the code model at all such as use cases. This 'Getting started' example will focus on the code model and demonstrate creating a new unit containing a new component class.

To create code for a new (component) class (or interface) in ModelMaker you will at a minimum need to,

1. Create a new model in which you want to put related classes, if you don't want to add the new class to the current model.
2. Add a new class to the model defining it's class name and ancestor.
3. Add (or override) properties, methods and events to the class's interface.
4. Implement the new methods.
5. Add the new class to a (new) unit.
6. Generate the unit to actually create or update a source file on disk.
7. In Delphi, debug the unit, and if it contains components, add it to the VCL.
8. While debugging, keep editing your code in ModelMaker, switching between Delphi and ModelMaker using ModelMaker's integration experts

An alternative way is to import existing files into a new model to either maintain these units in ModelMaker or to derive new classes from. This will be explained in detail in the Import demo.

The more advanced features of ModelMaker demonstrated in this example include:

9. Creating documentation for your component.
10. Generating a Help file.
11. Creating a class diagram to document your design.

The demo component: TIntLabel

Let's examine these steps a little closer by creating an new component class `TIntLabel` which is a `TLabel` descendant. `TIntLabel` adds a property `NumValue` of type integer which simple converts the `Caption` property to an Integer. We'll store this class in a new file `INTLABEL.PAS` and register it on page '*MM Demo*' in the VCL. We'll also create the help file `INTLABEL.HLP` and integrate it with Delphi's on line help.

This demo project is also shipped with ModelMaker. You can load the `GETSTART.mpb` in the `[installdir]\DEMOS` folder. The source file `INTLABEL.PAS` is also in this folder.

The ModelMaker Class Creation Wizard

If you start up ModelMaker the first time you'll see the Class Creation Wizard. This wizard makes it easy to create new classes and add them to a (new) unit. This wizard can be found at the main menu 'Tools|Create Class wizard'.

The wizard is great for adding classes, but for demonstrating the ModelMaker development model it's more instructive to create a new class manually. Therefore, if you started ModelMaker and the wizard is automatically started, abort the wizard by clicking 'Cancel'. You might also want to uncheck the option 'Show at start up' which will stop the wizard from appearing each time you run ModelMaker.

Creating a new project

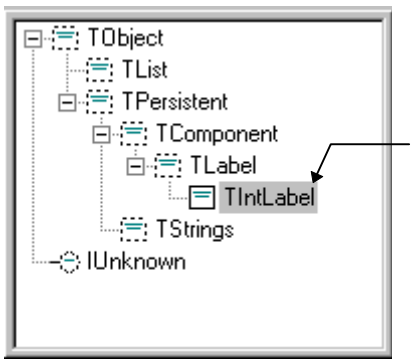
There are three ways to create a new project (or model):

- Select 'File|New', you'll get a clean project just containing the default ancestors `TObject` and `IUnknown`.
- Select 'File|New from default', you'll get a new project loaded from the default template.
- Select 'File|New from template', you'll select a template other than the default to create a new project.

In this case select 'File|New from default' to create a project which at least contains the `TComponent` class, if you didn't modify the default project shipped with ModelMaker `[installdir]\BIN\DEFAULT.mpb`.

Creating new classes

We'll use the Classes view to create a new class. In this view you add a new class as a descendant to another class in the model. The Classes view is depicted here.



Class tree containing place holder TLabel and real class TIntLabel

The ancestor class must *always* be part of the model since ModelMaker needs it to correctly generate the class declaration. In our case this implicates that before adding the `TIntLabel` class, it's ancestor `TLabel` must exist in the model. This raises a problem. If you started with the same default model, or with a template model that did not contain the class

TLabel, you will have to add the TLabel class first. But, in order to correctly add a class TLabel to your model, you now need to add it's ancestor first, and before that, etc.... help!

Since you do not intend to create code for TLabel, but only use it as an ancestor, there is no need to have the correct ancestor for TLabel. In our example the ancestor for TLabel could be anything, for example TObject. A better fitting ancestor is of course TComponent. Classes like TLabel in our example are called 'placeholder' classes as opposed to 'real' classes such as TIntLabel. Other examples of placeholders are TObject, Delphi's default class ancestor and TComponent.

What we have to do now, is add two classes TLabel ('placeholder') and then add TIntLabel ('real' class).

To do so:
1. Make the Classes view visible by selecting 'View|Classes' (or press F3)
2. Select TComponent by clicking it.
3. Press the "Ins" key or select add 'Add descendant' from the popup menu.
4. Enter TLabel as class name. You might want double click the class and in the class editor dialog check the option 'placeholder' to make it clear that TLabel is just a substitute for the real TLabel (which is in unit StdCtrls).
5. Now add the class TIntLabel using the TLabel as it's ancestor the same way. Of course you don't check 'placeholder' here.

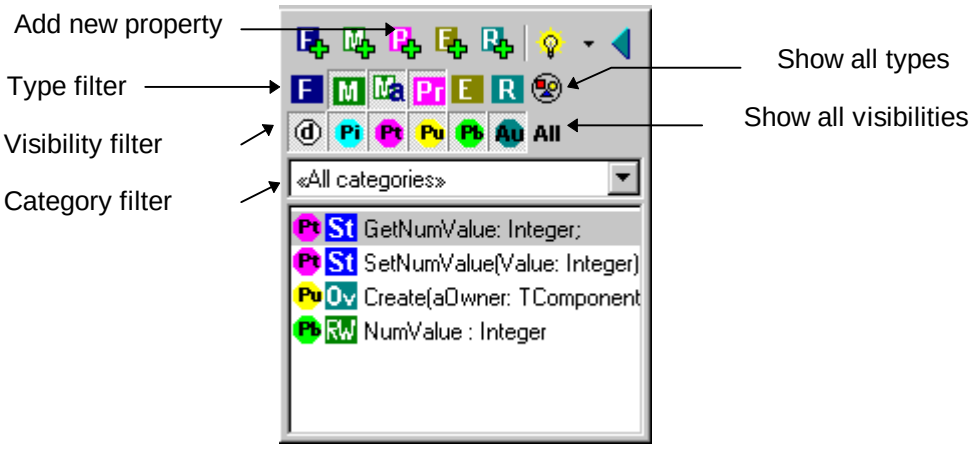
In the Classes view you'll see a tree or list based overview of all classes (and interfaces) in the model. Use the popup menu to toggle between tree and list style.

Adding properties and methods to a class

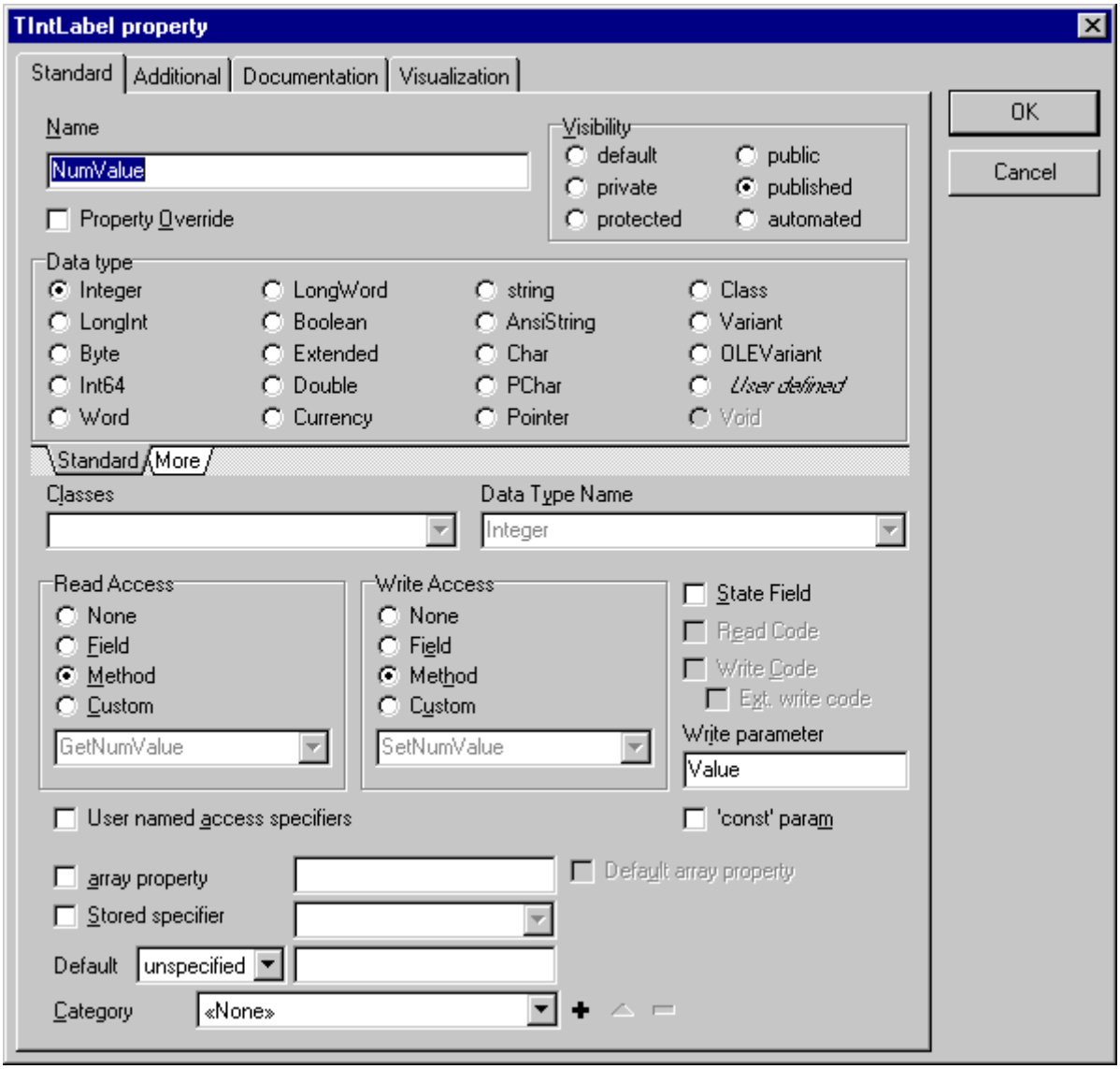
In our example we now need to add a new property and a read and write access method to the interface of the class TIntLabel, to get something like:

```
type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;
```

To do this we'll use the Class Members view - the bottom left window in the main window. Class Members are the fields, methods, properties (and event type properties) that make up a class's interface. The Class Members view is depicted here.



1. In this view all members for the currently selected class are displayed. Filters on *type* (Field, Method etc.), *visibility* (private, protected etc.) and *category* let you filter which members are displayed. Reset the filters by selecting 'Reset Filter' from the pop-up menu or by clicking the buttons 'Show all types' and 'Show all visibilities'. All filter buttons should be in a 'down' state now, except of course the 'Show all..' buttons which do nothing but (p)reset the filters. Make sure the category filter shows <all categories>. The member list is still empty because we didn't create any new Class Members yet. Note that the filter layout can be toggled using the popup menu 'filter layout or double clicking on the filter area.
2. Click the 'Add property' button.
3. The property editor dialog will appear. See picture below.
4. Enter `NumValue` as the property's name.
5. Select the visibility 'published'.
6. Make sure the property's data type is 'Integer'.
7. Select for Read Access 'Method'. This defines that the property has read access and you want to use a method to access it, rather than a field.
8. Select for Write Access 'Method' This defines that the property has write access and you want to use a method to access it, rather than a field.
9. Leave the other settings in their default values and click OK. In the property editor's picture below the correct settings are displayed.



Now have a look again in your Class Members view:
You'll not only see a property `NumValue`, but also two property access methods `GetNumValue` and `SetNumValue`. This is because properties create and update their access fields and methods automatically. Now that saves time!

The `TIntLabel` class's interface is now defined, but the methods `GetNumValue` and `SetNumValue` still need to be implemented.

Implementing methods

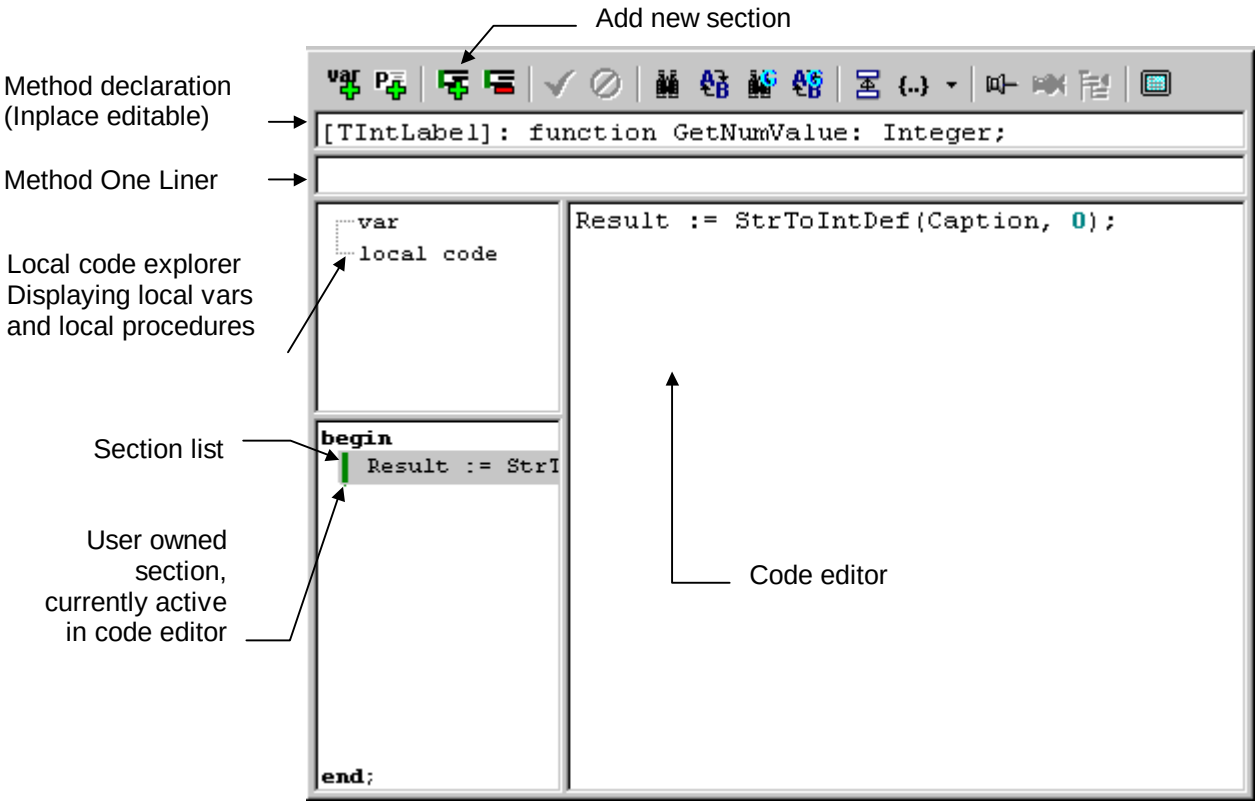
In our example we will need to add code to the implementation of the methods `GetNumValue` and `SetNumValue`. This code should be something like:

```
function TIntLabel.GetNumValue: Integer;
begin
```

```
  Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;
```

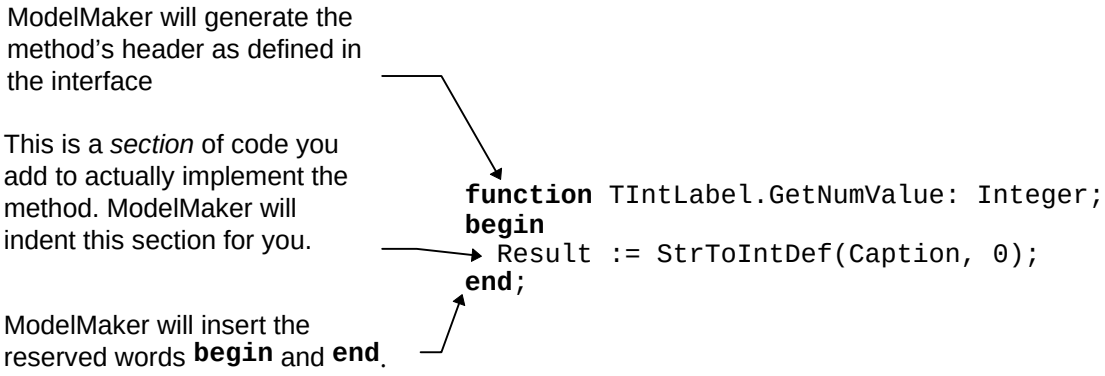
To add code to a method's implementation you use the (Method) Implementation view.

1. Select the method you want to implement in the Class Member view, in this case the method `GetNumValue`.
2. To make the Method Implementation view visible, select 'View|Implementation'.



The picture above shows the Method Implementation view. This editor is perhaps the element of ModelMaker that is the most different from other editors. To understand how this editor works you need to know a little more about how ModelMaker generates code for a method's implementation.

Let's have a closer look at the `GetNumValue` method. This is just a simple method, not containing any local variables or local procedures.



The body of a method's implementation consists of a list of local variables, local procedures and *sections* of code which implement the block between **`begin`**`..`**`end`**. A section can take up any number of lines of code. All sections together make up the actual implementation. Using sections, ModelMaker is able to identify certain lines of code within the body. This is for example used to automatically add and update a call to the inherited method, as we'll see later.

On the left side of the method code editor the complete method's code is displayed, although maybe *collapsed* if necessary. On the right we find the actual code editor. It is used to edit the section of code selected in the sections list. The same editor is also used to edit the local procedures code.

As we see in the above picture, the only thing we need to do, is add a section of code containing the statement:

```
Result := StrToIntDef(Caption, 0);
```

To do this, first create a new section. If you didn't change the code options settings in 'Options|Code options' a new section is automatically created if the method does not contain any sections yet.

1. If necessary, add a new section by clicking the 'Add section' button.
2. Enter the statement in the code editor. There's no need to indent the code with spaces since this will be done automatically by ModelMaker.
3. Click the 'save code' button. This is not really necessary, since ModelMaker will automatically save the section as soon as you select a new section or a new method.

Notice that the section is now also displayed in the section list, and is marked with a green line. This green line informs you that *you* created this section and are it's *owner*. Red lines indicate that a section is not owned by you, but, for example, is inserted by a pattern which has the only rights to update it. If a section contains more lines than the current 'Fold height' (adjustable in the Environment options tab Editors), the section will be collapsed. Collapsed sections are marked with a second purple line with a mark on the collapsing position. More about this later.

Now you should be able to implement the `SetNumValue` method the same way: select the method in the Class Members view, add a section if necessary and enter your code.

Although we have finished implementing the class `TIntLabel` for now, all code exists only in the ModelMaker model, so the next thing to do is to generate a source file.

Creating a source file

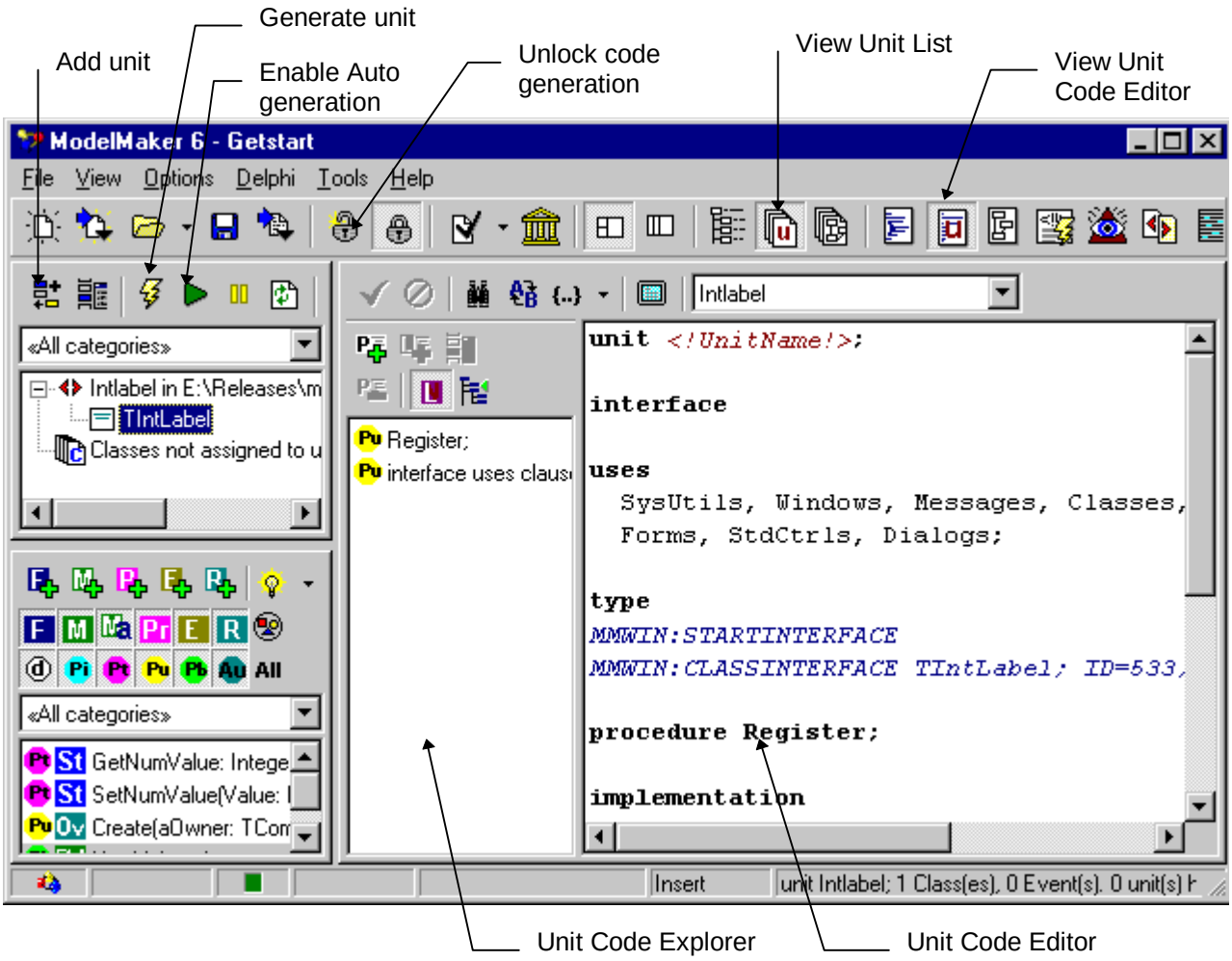
Units are the gateways to source files on disk. They provide a link between all data in a ModelMaker project such as classes, method implementations etc. and an Object Pascal style unit file which Delphi is able to compile.

Creating a Unit

In this example we need to create a unit which, after it has been generated, should look something like:

```
unit IntLabel;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TIntLabel = class (TLabel)
  protected
    function GetNumValue: Integer;
    procedure SetNumValue(Value: Integer);
  published
    property NumValue: Integer read GetNumValue write SetNumValue;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('MM Demo', [TIntLabel]);
end;

function TIntLabel.GetNumValue: Integer;
begin
  Result := StrToIntDef(Caption, 0);
end;

procedure TIntLabel.SetNumValue(Value: Integer);
begin
  Caption := IntToStr(Value);
end;

end.
```

To create a new unit we use the Unit List view which is depicted below,



1. To make the Unit list view visible, select ‘View|Units’. Repeat this for the Unit Code editor.
2. In the Unit list click the ‘Add unit’ button, a unit properties dialog will now appear.

In the Unit editor dialog you define:

1. Leave the source path alias `<no alias>` unchanged.
2. The source file name (the full path including drive and folders), to define the path you could use an source alias, but for now just click the browse button and locate the `\ModelMaker\6.0\TEST` folder (or `ModelMaker\6.0\Test` depending on the base path you installed ModelMaker in) and enter the file name `IntLabel.PAS`.
3. On the tab sheet ‘Classes’ add the class `TIntLabel` to the list on the right either by dragging or by selecting it and clicking the ‘Add selected’(▸) button or by double-clicking.
4. Change the ‘VCL page’ from `<unregistered>` to ‘`MM Demo`’ by entering this name in the string grid.
5. Click OK.

We’ll see the newly created unit now listed in the unit list on the left. In this editor unit’s code containing a text which should look something like:

```
unit <!UnitName!>;
```

```
interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
MMWIN:STARTINTERFACE
MMWIN:CLASSINTERFACE TIntLabel; ID=7;

procedure Register;

implementation

procedure Register;
begin
MMWIN:CLASSREGISTRATION TIntLabel; ID=7; Page='MM Demo';
end;

MMWIN:STARTIMPLEMENTATION
MMWIN:CLASSIMPLEMENTATION TIntLabel; ID=7;

end.
```

For now, it is enough to understand that ModelMaker uses tags (like `MMWIN:CLASSINTERFACE`) to insert the interface, VCL registration and implementation of a class in otherwise 'plain' text that you define. There is one problem however: if you look at the **`uses`** clause in the interface, you'll see that the unit `StdCtrls` which defines the ancestor class `TLabel`, is missing. That is because the default unit template we are using does not contain this unit. For changing this template refer to 'Customizing ModelMaker'. For now, we will have to add `StdCtrls` manually.

To do this,

1. In the unit code editor add `StdCtrls` to the uses clause.
2. Click the 'Save code' button in the toolbar above the editor.

Generating the source file

To generate the source file and create or update a file on disk,

1. Make sure that code generation is not locked. Locking is explained later, for now make sure the button 'unlock code generation' in the ModelMaker toolbar is pressed down.
2. Click the 'Generate current unit' button in the unit list view.
3. Start Delphi (if it was not running already).
4. Either manually switch to Delphi or - much more instructive - click the 'Locate in Delphi' button in the main tool bar - or simply press Ctrl+F11. This will open the unit and locate the entity currently selected in ModelMaker.

The generated source file should look pretty much the same as we wanted it to be. (Differences may occur if you modified your file `DEFUNIT.PAS` in ModelMaker's `\BIN` folder.)

We're ready to debug the `TIntLabel` now, and install it in our VCL.
Before doing this it's a good idea to save our model. This is very much like in other windows applications, so it won't be explained here. We could use the `[installdir]\TEST` folder to

save this project. Practice shows that it is convenient to name your model after the main source file you create with it, or the main set of components. In this case `INTLABEL.mpb` seems an obvious name.

Adding the component to the VCL

Debugging your component

A good practice, is to debug your new component before adding it to the VCL. Do this for example by adding the source file to the current Delphi project (e.g. by using Delphi's project manager) and re-compile the project. This should at least filter out all syntax errors. Either use 'Compile' or 'Syntax Check' since Delphi does not always correctly manage the file's date/time and modified status if you just 'Run' the project.

Compiling errors

If you didn't make any mistakes, the unit should compile all right. If it doesn't: change the code in the appropriate place in ModelMaker. That is:

- For missing units in the unit's uses clause: the unit code editor.
- For any code not part of the class: the unit editor.
- For errors in the class's name or ancestor name: the Class view.
- For errors in the class's interface declaration: the Class Members view.
- For errors in a method's implementation: the Method Implementation view.

Switch to ModelMaker and fix the code. Use the integration expert's menu 'Jump to ModelMaker' to jump straight from Delphi's code editor to the corresponding position in ModelMaker. Finally, in the Unit list view click the button 'Generate' again and re-compile the Delphi project. If you are having trouble with this: look ahead where we are adding new behavior and editing code is explained.

Adding the component to the VCL

After debugging your new file, add it to the Delphi's VCL.
In *Delphi 1.0*: select menu 'Options|Install Components', select 'Add' and browse to find the unit `INTLABEL.PAS` in folder `..\ModelMaker\6.0\TEST\`.
In *Delphi 2.0*: Select menu 'Component|Install', select 'Add' and browse to find the unit `INTLABEL.PAS`. Refer to your Delphi User guide for more information about installing components.
In *Delphi 3 and higher*: you must install the new component in a package. Select a new package called MMtest in the ..ModelMaker\6.0\Test folder. Please refer to your user guide for installing packages.

After recompiling the VCL the new `TIntLabel` component should be on the palette page where you registered it: *MM Demo*. To test it, add a `TIntLabel` component to a (new) form.

You can use the Object Inspector now to set the '`NumValue`' property and see the caption change. But the component can be improved!

Improving the component in ModelMaker

If you watch carefully, you'll see that a `TIntLabel` when dropped on a form, initially has the caption '`IntLabel1`' rather than '0'. The `NumValue` however is 0. This is conflicting and not very nice. To improve the component we'll have to override the constructor Create like this: (refer to Delphi's on-line help for the `TControl.ControlStyle` property)

```
constructor TIntLabel.Create(AOwner: TComponent);
begin
 inherited Create(AOwner);
  { Don't let the Object Inspector set a caption }
  ControlStyle := ControlStyle - [csSetCaption];
  { Instead pre-set the Caption ourselves }
  NumValue := 0;
end;
```

To do this we need to return to ModelMaker.

Keep editing your code in ModelMaker

We could of course change `TIntLabel`'s code in Delphi, but then the ModelMaker model and the modified source file would be out of sync. The next time we would (re-)generate the file from within ModelMaker, the changes made in the Delphi Editor will be lost. Of course, if we do not intend to maintain our code any longer in ModelMaker that's fine, but we won't benefit the advantages ModelMaker offers during maintenance and documentation. And although it may seem a burden at first, after getting used to it, the benefits of keeping the master code in ModelMaker are much higher than the costs. So resist the itch in your fingers and return to ModelMaker now.

Overriding the constructor Create

To override the constructor `Create`, we could add a method in the Class Members view clicking 'Add method', name it 'Create' adjust it's other attributes such as parameters '`AOwner: TComponent`', method kind 'constructor', etc. but overriding methods can be done far more easy. The only thing is: in order to override a method (or property) the method to be overridden must exist in the model. If you used the default project template as was shipped with ModelMaker (which also contains the class `TComponent`), the virtual constructor `TComponent.Create` is in your model with the correct attributes, ready to be overridden.

To do so:

1. In the Members view tool bar click the 'Wizards' button, and

2. Select ‘Method override wizard’. Alternatively use the same function from the Wizards popup sub menu.
3. In the ‘Override Methods’ dialog select the ‘`Create`’ method.
4. Make sure the option ‘Call inherited method’ is checked. This will instruct ModelMaker to add a section of code containing a call to the inherited method.
5. Click OK.

In the Class Members view we’ll see that a method called `Create` is added to the list. Just for your information you may check the methods attributes by selecting it in the member list and clicking the ‘Edit Member’ button, or double clicking it in the member list.

Notice that all relevant attributes are copied from `TComponent.Create`:

- The methods name is `Create`.
- The parameter list is `AOwner: TComponent`.
- The method is ‘public’.
- The data type is ‘void’.
- The method type is ‘constructor’.
- The binding kind is ‘override’ (since `TComponent.Create` is virtual).
- The option ‘Call inherited’ is checked because we checked the option ‘Call inherited method’ running the override wizard.
- The option ‘Inheritance restricted’ is checked. If this option is checked, the method will automatically be updated to reflect any changes applied to the overridden method in the ancestor class.

Click Cancel to leave the method in it’s original state.

Implementing Create, non-user sections in code

To implement the method `Create` switch to the Method Implementation view again. Notice that in the section list on the left, already one section is added containing the code:

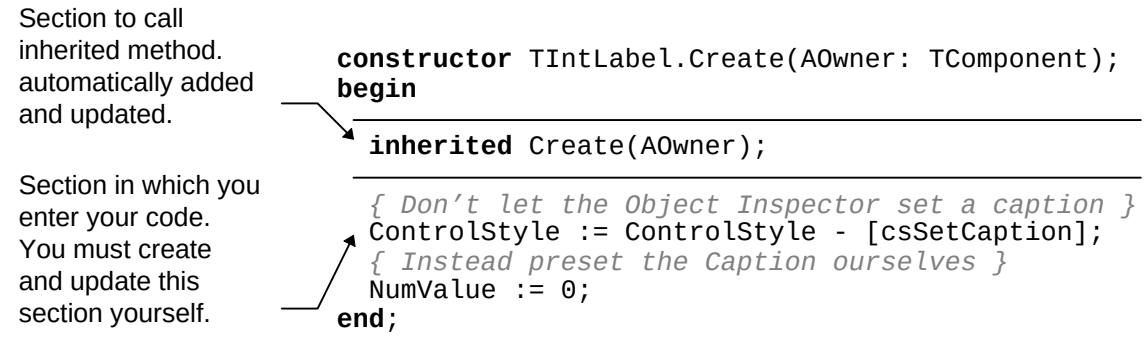
```
inherited Create(AOwner);
```

This section is marked with a red line, indicating that we cannot edit it’s contents. The section was added because the method’s option ‘Call inherited’ is checked.

To add the other lines of code,

1. Add a new section by clicking the ‘Add section’ button.
2. Enter the code in the code editor on the right.
3. Click the ‘Save code’ button

In the section list on the left we'll see the complete implementation of `Create`.



Instant code generation

If we have a look in the Delphi editor, we'll see that the source file has not been updated yet. To do this we need to regenerate the unit. Therefore switch to the Unit list view again.

Regenerating the unit could be done by clicking the 'Generate' button again, but it is more instructive to demonstrate ModelMaker 's *instant code generation* feature. Rather than having to manually regenerate a source file whenever something has changed, it is possible to '*Enable Auto generation*' for a unit. The source file will then be regenerated each time anything changes in the Model that affects the source file. It is a nice feature that ModelMaker not only regenerates the source file, but also instructs Delphi to reload the file if it's opened in Delphi's code editor. Refer to Integration with Delphi.

To watch this:

1. Make sure you have the `INTLABEL.PAS` file loaded and is on top in the Delphi code editor.
2. Click the 'Enable auto generation' unit button in the Unit view tool bar.
3. See how the Delphi editor now reflects the last changes in your file.

Now return to ModelMaker again, and let's play around:

1. Switch to the Class Members view.
2. Edit the Create Method (Double click or click the Edit button).
3. Now uncheck the 'Call inherited' option and click OK.
4. Watch the code being updated in Delphi.
5. Now check the 'Call inherited' option again.

To have a closer look at the Method Code editor,

1. In the Create method's section list you can drag sections up and down, do this and see how Delphi's code editor follows your changes.

Lets have play around with the units view:

1. Make sure the units view is in ‘display as tree mode’ (popup menu)
2. Select the class `TIntLabel` by clicking it.
3. Press the Del key once, this will remove the class from the unit. The class is still in the code model. In fact, the class is listed under the ‘classes not assigned to units’ node. Check the code in the IDE: you’ll see that the entire interface and declaration have been removed.
4. Drag the class on the IntLabel unit. This will add the class to the unit again. Note that the VCL Component registration page is now reset to ‘none’. Edit the unit (double click unit or use toolbar) to change this back to ‘MM Demo’.

After you have played around with the instant code generation feature, make sure the `TIntLabel` class still is as you want it to be. In order to actually see the improved behavior,

1. Rebuild your VCL in Delphi.
   Delphi 1: menu ‘Options|Rebuild library’.
   Delphi 2: menu ‘Component|Rebuild library’.
   Delphi 3 and higher: recompile the package MMtest.dpk.
2. Remove any old `TIntLabel` components from forms.
3. Add a new `TIntLabel` to a form and notice how the Caption is set to ‘0’ now.

Documenting your component

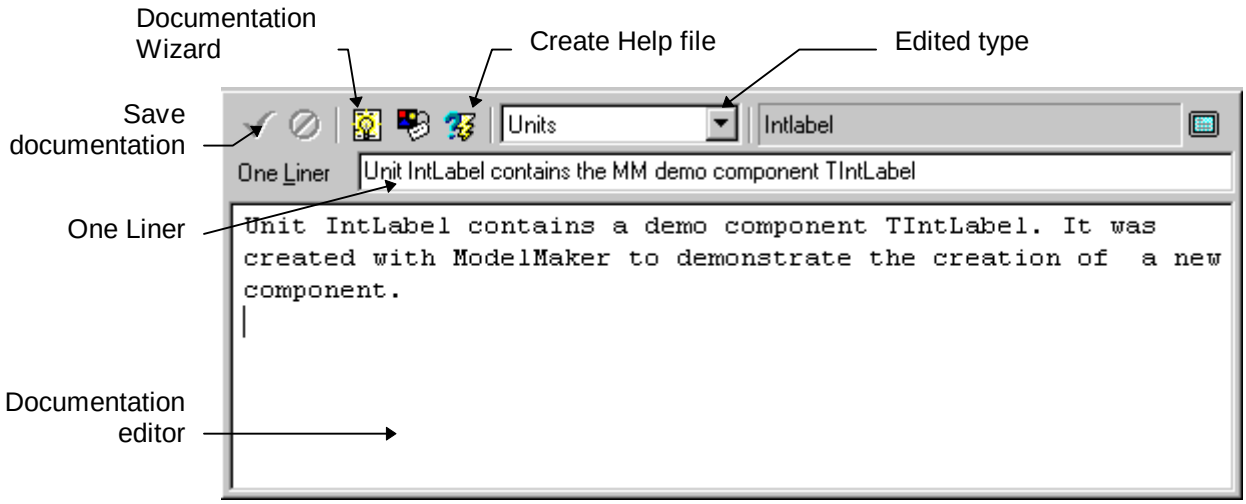
ModelMaker not only supports source code generation for your component, but it has also advanced wizards and generators to document your component. These include

- Documentation wizard which inserts basic standard documentation for all members in a class.
- In source documentation.
- Help file generation.
- Instant visualizing in class diagrams. Although creating diagrams is usually done in the design process, it is also possible to create diagrams from existing code.

To demonstrate these features we will now create a help file and a class diagram for the `TIntLabel` component.

Adding documentation to your component

Each unit, class, all members of a class, ebent types and symbols in diagrams can be documented with a short description named One Liner and a longer text named Documentation. For editing One Liner and documentation we’ll use the Documentation view. Alternatively we could have used the floating documentation window which is available from the main menu “Views”.



To make this view visible: Select menu 'View|Documentation Editor.

With this Documentation editor you add a One Liner and a more descriptive text to each unit, class and member (method, property etc.). That can be quite a job, so ModelMaker includes a documentation wizard, which does some of the nasty work for you. This wizard will insert pieces of documentation in the currently selected class.

To demonstrate this:

1. Select the class `TIntLabel` in the Classes or Units view.
2. In the Documentation view click the button 'Documentation Wizard'.
3. Click OK to confirm creation of standard documentation.
4. In the drop-down box 'Edited type' select 'class members'.
5. In the Class Members view select the `GetNumValue` method.

What you see now in the documentation editor is that the wizard inserted text like:
GetNumValue is the read access method for the NumValue property.
Usually this is sufficient to document `GetNumValue`, since you will be documenting the exact meaning of the property `NumValue` and this avoids redundancy.

Selecting the method `SetNumValue` in the Class Members view makes the documentation for `SetNumValue` visible:
SetNumValue is the write access method of the NumValue property.
Again this is usually sufficient to document the `SetNumValue` method.

What remains to be done is documenting the constructor `Create` and the property `NumValue`. But here too, the wizard inserted already some useful text.

Select the documentation for the constructor `Create` and change this to:
Constructor Create overrides the inherited Create. First inherited Create is called, then the Caption is pre-set to 0, reflecting the initial NumValue state. ControlStyle is modified to exclude csSetCaption.

Now select the `NumValue` property's documentation and change this to:
Property NumValue is read/write at run time and design time.

It reads and writes the Caption property as an Integer.

To edit the documentation for the class `TIntLabel`:

1. Make sure class `TIntLabel` is selected in the Classes or Units view,
2. In the Documentation view, select 'classes' in the drop-down box 'Edited type'.

Enter the text:
TIntLabel is a simple TLabel descendant created with ModelMaker.
It adds the property NumValue which reads and writes the Caption property
as an Integer.

To edit the documentation for unit IntLabel which contains the `TIntLabel` class:

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, select 'units' in the drop-down box 'Edited type'.

Enter the text.
Unit IntLabel contains a demo component TIntLabel. It was created with
ModelMaker to demonstrate the creation of a new component.

The unit `IntLabel` is now completely documented. Documentation is typically used to create a helpfile or for in-source documentation. Third party plug-in experts use the ModelMaker ToolsApi to output documentation to other formats.

You add One Liners (short, single line descriptions) the same way. In the Views menu you'll find a 'Floating documentation' view. This view can be used to insert One Liners and documentation too. This view can be docked or stay floating. The edited entity type in this view is automatically updated to reflect the last focused view in ModelMaker.

Creating a help file

ModelMaker can create a Borland style help file from your documentation. This includes the generation of the Borland `/B` keywords which are necessary for interaction with Delphi's on-line context sensitive Help. Help files are generated from unit's. In our example we'll create a help file for unit `INTLABEL`.

ModelMaker let's you select the visibilities you want to include in your help file. These are:

- 'User' (public, published, automated and default)
- 'Component writer' (user visibilities plus protected)
- 'Developer' (all visibilities)

The default visibility 'User' is the most restricted, since this will include only help for the public, published or automated interface of a class. Use this filter to create a help file you distribute with your components. Selecting the 'Component writer' visibilities will also include help for the protected interface. This is the type of help file you would typically distribute with components if other developers should be able to derive a descendant class from your component. The last selection includes also the private details (typically fields and or property access methods etc.) which you might want to have documented internally.

To create a help file for the `TIntLabel` component,

1. Make sure the unit `IntLabel` is selected in the Units view.
2. In the Documentation view, click the button 'Create help file'.
3. In the 'Create help file' dialog you'll be prompted to enter a file name for the unit's RTF file. A help project file with the same name and extension `.HPJ` will automatically be created. In our example enter `[installdir]\TEST\INTLABEL.RTF`.
4. In the same dialog, select the visibilities you want to include in your help file. In this example we'll go for the 'Component writers' visibility.
5. Leave the reformat paragraphs option checked and Click OK.
6. Open the explorer (file manager) and notice that both `INTLABEL.RTF` and `INTLABEL.HPJ` have been created.
7. Run your Delphi help compiler with the `INTLABEL.HPJ` project. Be aware that you need to use the help compiler that was shipped with the Delphi version you want to create help for. For *Delphi 1.0*: Use `DELPHI\BIN\HC31.EXE` in a DOS box and run it from the folder your HPJ file is in. For example: `DELPHI\BIN\HC31.EXE INTLABEL.HPJ`
   For *Delphi 2, 3 and higher*: Use the `Delphi 2(or 3 / 4/5/6).0\HELP\TOOLS\HCW.EXE` to compile your project: double clicking the `INTLABEL.HPJ` file in the explorer should be enough.

The help files have been created now. You might want to have a look at them, using windows help. Double clicking the newly generated `INTLABEL.HLP` starts help.

The Help File Generator source is available as plug-in expert on request. It can be extended to create help for multiple units at once etc.

Integrating your help files with Delphi's on line help

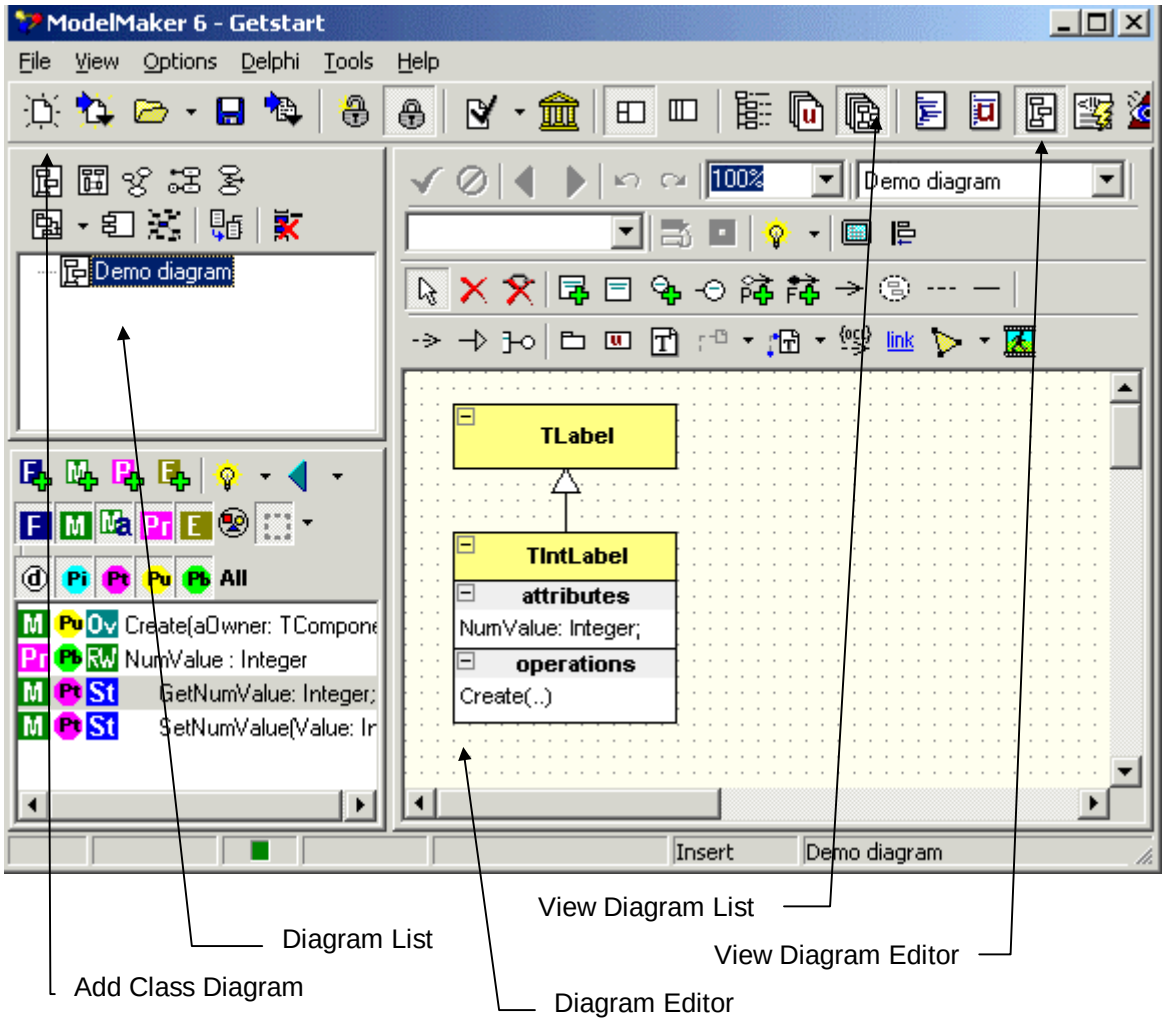
To be able to invoke Delphi's help on your `TIntLabel` component you must integrate your `INTLABEL.HLP` help file with the Delphi 1 and 2 on line help. To do this:

1. Generate key words using `KWGEN.EXE`
2. Install help using `HELPINST.EXE`

Installing help in Delphi 3 and higher is documented in Delphi's Component Writers Guide / User's Guide.

Documenting the design in a diagram

Although creating diagrams is usually done in the first stages of the design process, it is also possible to create diagrams from existing models using ModelMaker's instant visualization feature. A list of diagrams in the model is edited in the Diagram List view. The actual diagrams are edited in the Diagram Editor.



Make the Diagrams List view and Diagram Editor visible:

1. Select menu ‘View|Diagrams’. This will make the diagram list visible in the top left window.
2. Select menu ‘View|Diagram Editor’. This will make the diagram editor visible in the editor pane on the right
3. In the Diagram list create a new class diagram by clicking the ‘Add Class diagram’ button.
4. You can inplace edit the diagram’s name. Enter ‘Demo diagram’. The name is intended only to distinguish different diagrams.

In the newly created diagram we’ll demonstrate ModelMaker’s instant visualization feature:

1. Make sure the Classes or Units view is visible (View|Classes or Units)
2. Drag the class `TLabel` from the Classes or Units view and drop it on the diagram editor.
3. Do the same with `TIntLabel`.
4. Save the Class diagram by clicking the ‘Save diagram’ button.

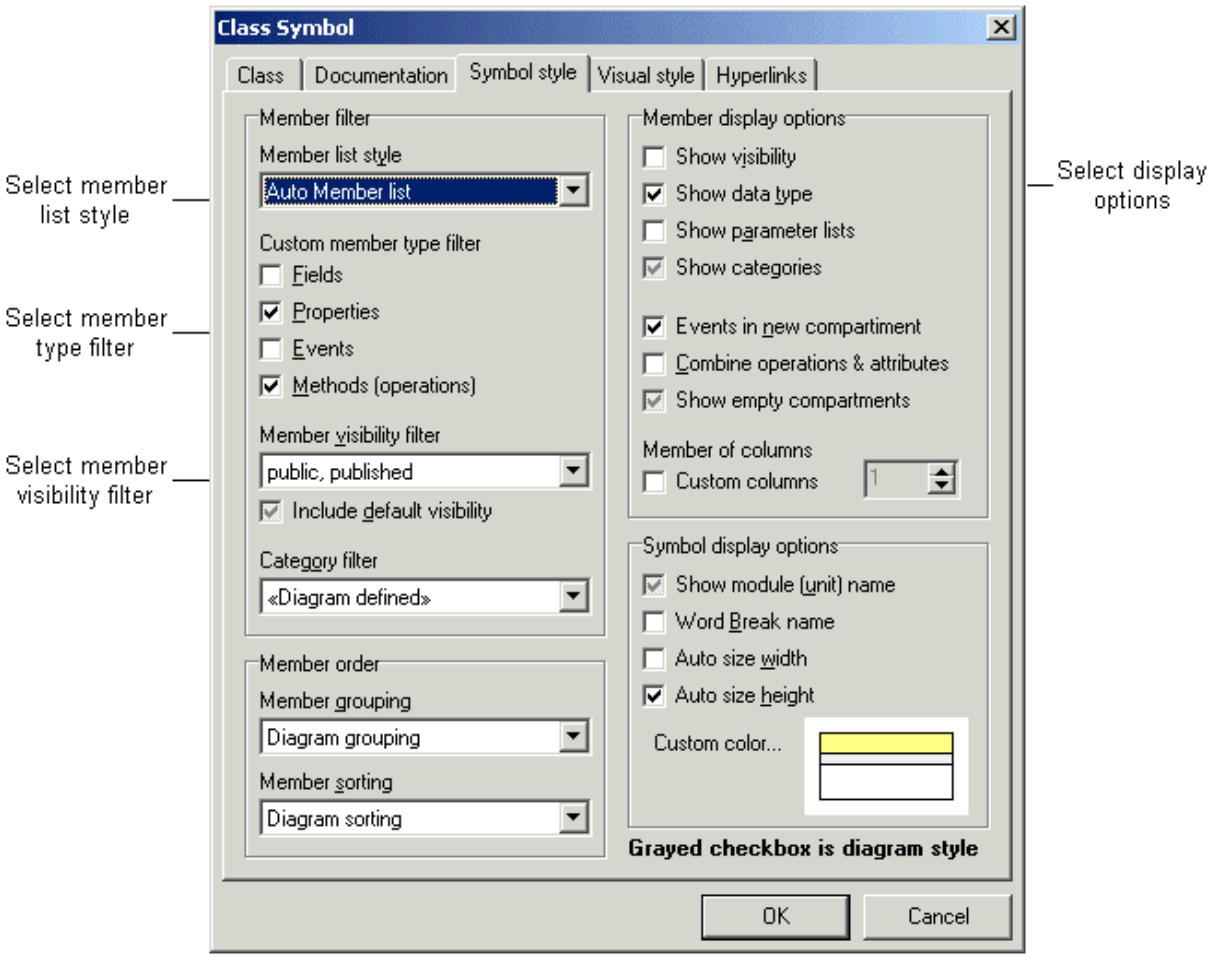
Notice how the inheritance relation `TIntlabel = class (TLabel)` is automatically visualized. If appropriate, ModelMaker will also visualize ‘uses’ relations if a class is dragged onto a diagram.

You can copy the current selected diagram to the clipboard (in WMF format) and paste it in your word processor to document your design. To do this: press `Shift+Ctrl+C` or use the local menu 'Export as Image'|'Clipboard'. Alternatively you could export the image to a file (bmp, wmf and jpg). To print the diagram, press `Ctrl+P` or use the pop-up menu.

Symbol styles in Diagrams: displaying members

ModelMaker supports many UML styles of displaying classes and interfaces: show module (unit) names, show members, collapse interfaces etc. To demonstrate a few and give you an idea of what is possible we'll change the symbol style for the `TIntLabel` class symbol. Symbol styles define how a symbol is displayed in a diagram. Note: the visual appearance (colors, fonts etc) is controlled by the visual style which is not part of the symbol style; check this manual for details. Default all symbol styles are "as defined in current diagram". The diagram symbol style defaults to "as defined in the current project". This gives you the possibility to change style on any level you like: just a single class symbol, all class symbols in a diagram or all diagrams in a project.

Here we'll change the symbol for TIntLabel only. Double click on the TIntLabel class symbol. This will show the class symbol editor.



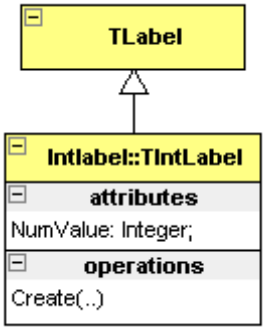
First you define ***which*** members are displayed. In this dialog change the Member list style from "Diagram Auto Member list" to "Auto Member list". This style will automatically add all members defined by the member filter in this editor. In the Member filter check

“Properties” and “Methods”. Change the visibility filter to public and published members only. This will suppress display of the protected property access methods GetNumValue and SetNumValue.

Then you define ***how*** members are displayed. Use the Member display options to modify this. Check “Show Data type” and “Events in new Compartment”. Note that a grayed option reverts the option to the parent (diagram) style.

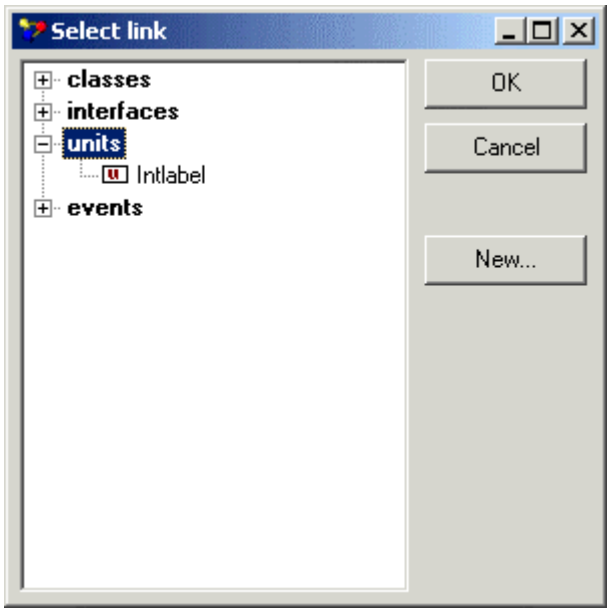
To display the name of the unit that contains this class in the symbol name compartment, check the “Show module name” option.

After clicking OK you should have a diagram that looks something like this (you can stretch a class symbol as needed depending on the Auto Size options – check the class symbol dialog). The property NumValue is displayed in the ‘attributes’ compartment and the method Create is displayed in the ‘operations’ compartment. Likewise events can be displayed in a separate compartment or combined with the attributes. If you wish you could even combine all members into a single compartment.

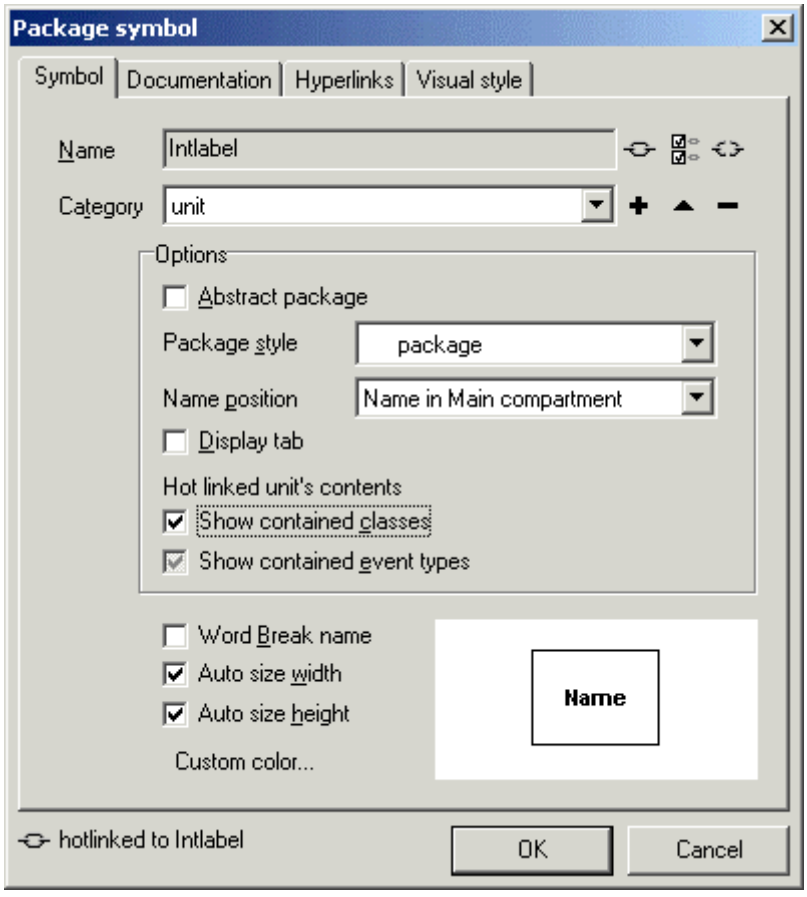


Visualizing the unit IntLabel.pas

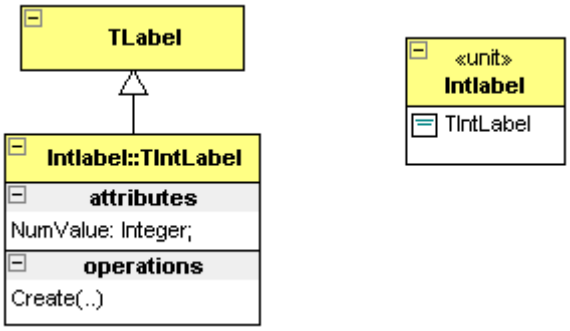
Just like classes can be visualized by class symbols, the unit IntLabel which contains TIntLabel can be visualized. To do this, click the “Add Unit Package” tool on the diagram editor tool bar and then click on the diagram. The following dialog will appear which lets you select the unit to visualize in the package. In this dialog, select “IntLabel” and click OK.



Now the Package Symbol editor will be visible, which is used to edit the visualization of the linked unit.



In this dialog, Make sure the “Show contained classes” option is checked (not grayed) and click OK. Your diagram should now look something like the picture below.

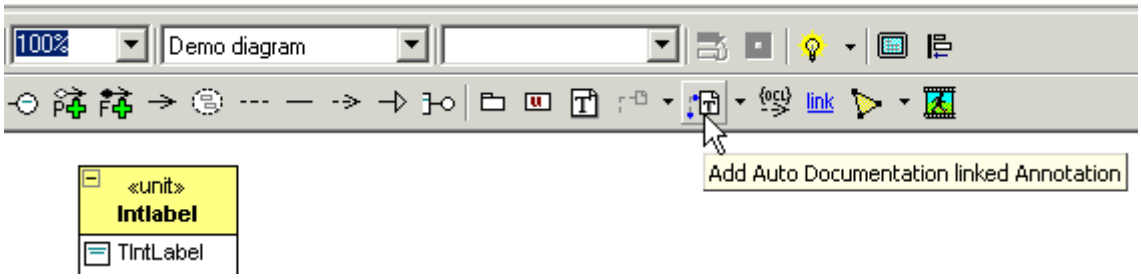


The package symbol displays it's contained classes (TIntLabel) and has a stereotype (category) named <<unit>>. Of course, if you add more classes to the unit, the symbol will be updated automatically.

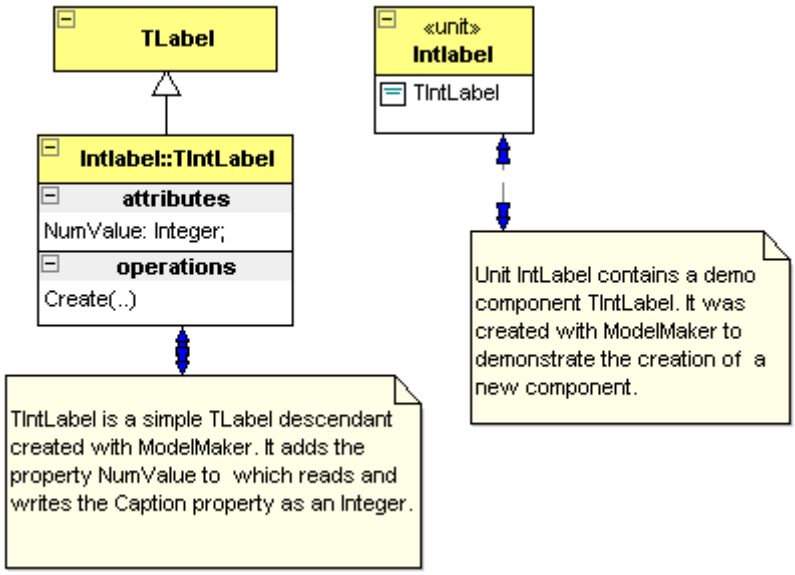
Visualizing the Documentation

The UML uses Annotation symbols to add notes to diagrams. ModelMaker supports hotlinking annotations to symbol documentation (or OneLiners). This is a two way hot link: the annotation text will automatically show the symbol's documentation and editing the annotation text will update the symbol's documentation. To demonstrate this, we'll add a linked annotation to the class symbol TIntLabel and to the unit package symbol IntLabel.pas.

On the diagram editor toolbar select the "Add Auto Documentation linked Annotation" tool. This tool allows three link styles which can be selected with the drop down button next to it. Links styles are: passive (not linked), documentation and one liner. Make sure the Documentation style is selected.



Then after selecting this tool, click on the class symbol and drag the mouse below it. This will create the link and annotation. Repeat this for the package symbol that is linked to unit IntLabel. Your diagram will now look something like this.



Now try to edit the documentation for TIntLabel in the annotation and see it change in the class dialog and the diagram. To do this, click TIntLabel's annotation and press F2. This invokes the annotation's inplace editor. Change the text to your liking and after pressing the Enter key, check the documentation tab in the class editor dialog (classes view). Similar to linking the documentation, a symbol's One Liner can be linked to an annotation.

Summary

In this chapter you got a first glimpse of some basic features in ModelMaker.

- ModelMaker is all about creating classes.
- It is important to start with the right project template especially if you want to override methods or properties.
- The interface of a class consists of Class Members (fields, methods and properties).
- Properties create and update their access fields and methods.
- Method code consists of sections, which can either be created automatically or manually.
- A Unit provides a link with a source file. Units can have 'auto generation' enabled to instantly reflect any change in the model to the source file.
- ModelMaker's IDE integration experts will take care of reloading units in Delphi's code editor.
- To jump from ModelMaker to the IDE and back, use the Locate in Delphi and Locate in ModelMaker commands.
- It's easy to document a design using the documentation wizard.
- Help files can be created for a unit, which are ready to integrate with Delphi's on-line help.
- Instant visualization in class diagrams can be used to document your design.
- Class symbols can automatically display contained members, package symbols can automatically display classes contained by a unit (source module).

- Symbols can be hotlinked to annotations, which will then display documentation or One Liners.
- Diagrams support multiple symbol styles that can be defined at different levels.

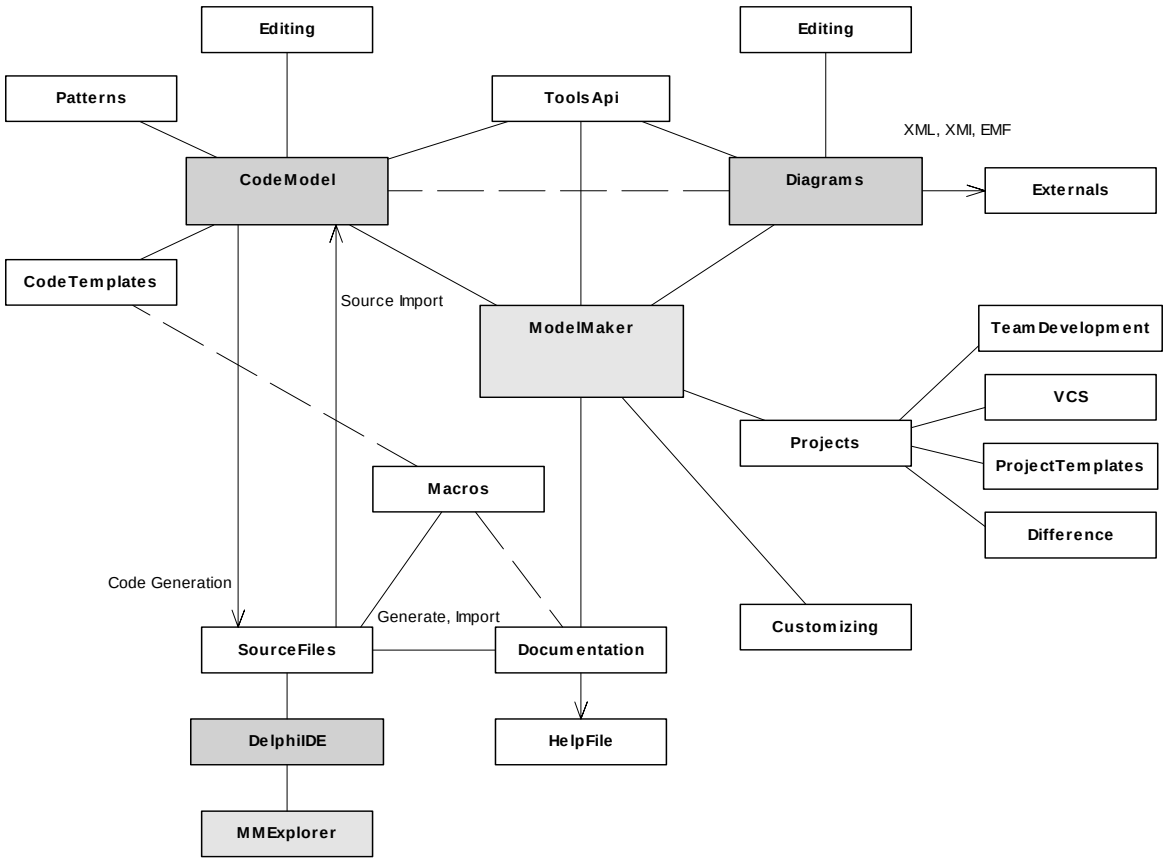
Where to now?

Now you've seen ModelMaker's basic mechanisms. You can have a look at the following topics:

- There is another step-by-step example in the Design Patterns manual that demonstrates the use of design patterns. You should have enough background now to work through this demo.
- ModelMaker's *Basic Concepts* page 36.
- The other chapters in this manual focus in greater detail on common aspects of ModelMaker such as diagrams, code generation and import.

Basic Concepts

Overview



This picture gives an impression of ModelMaker's main parts and how they relate.

The two main entities are the Code Model and the Diagrams. Around them you'll find importing and generating source code, interaction with the Delphi IDE, saving models etc. Going around this picture more or less anti-clockwise we'll see:

The ***Code Model*** contains the classes, members, units etc. that map to the corresponding concepts in Delphi's Object Pascal. The Classes view, Members view, Method Implementation view and Units (Code) view all deal with visualizing and manipulating the code model directly.

Code Generation is the process of creating an Object Pascal source file containing classes, members and unit code. ***Units*** provide the link between the Code Model and a source file. A unit contains classes and/or event type declarations plus user defined ***unit code*** such as

module procedures. Refer to Code Generation, page 45, for a more detailed description of source code generation. Code generation is typically, but not only, controlled from the Units view.

Code Import is the process of reverse engineering an Object Pascal source file into entities that make up the *Code Model*. This can be initial importing - the unit and / or classes it contains did previously not exist in the Code Model - or ***refresh import*** - re-importing an existing unit and / or the classes it contains in order to synchronize the Code Model and source file. Code Import is typically controlled from the Units view. It is amongst others also available in the classes view and difference view. Code Import is described in detail in chapter Importing Source Code, page 56

Design patterns are proven solutions for a general design problem. It consists of communicating classes and objects that are customized to solve the problem in a particular context. In ModelMaker patterns are active agents that will insert code into the model and stay *alive* to *reflect changes* in the model to the pattern related code. Design patterns currently only relate to the *Code Model*. Patterns are manipulated in the Patterns view. There is more on patterns in the Design Patterns manual.

Code Templates are user definable and parameterizable snippets of related code. They are like user definable patterns. Code templates can be created form the Members view and applied from the Patterns view or Members view. There's more in Code Templates page 71

ModelMaker's ***Delphi IDE integration*** takes care of synchronizing the Delphi IDE editor buffers whenever a file is (re-)generated by ModelMaker. Depending on the Delphi version there are other functions available like: add IDE editor file to model, refresh IDE editor file etc. There's more in chapter Integration with the IDE page 105

The ***ModelMaker Code Explorer*** is a separate *ModelMaker Tools* product that brings basic ModelMaker Code Model related functionality into the Delphi IDE. With this explorer you can navigate and add, edit, copy properties, methods or even entire classes with the same ease and concepts as in ModelMaker, usually even with the same dialogs.

A ***Macro*** is a fragment of text that is identified by a macro identifier. While generating source code and in-source documentation, ModelMaker will expand the text, replacing macro identifiers with the macro's text. Macros are also used to customize certain parts of the generation process (custom class separator, method section separator etc.). Macros are maintained in the Macros view. Macros are described in detail in chapter Macros, page 74 Macros are also used to parameterize *Code Templates* as described in chapter Code templates, page 71

All Code Model entities and Diagram symbols can be documented with ***Documentation*** and a ***One Liner***. In all relevant editors you'll find a Documentation tab. The Documentation view and Floating Documentation view are dedicated to editing One Liners and Documentation. Emitting "in-source documentation" during code generation is controlled by macros. You can redefine these macros to customize the documentation format.

Documentation can be converted to a unit based ***Help File***. This is done in the Documentation view. Other documentation output formats can be created with (third party) plug-in experts that use the ***MMToolsApi*** to access the model.

The ***Diagrams*** contains multiple types of diagrams. Some diagrams visualize aspects of the Code Model in UML-style. Others visualize entities such as Use Cases that only exist in Diagrams. Most symbols can be 'HotLinked" to entities in the code model. Class symbols for example are linked strictly to classes in the Code Model: changing the class in a diagram will also change the class in the code model. Messages in sequence diagrams can be weak linked: they can or cannot be linked to a class member. If they are not linked the message name is just text. Symbols such as Use Case symbols only exist in the Diagram model and have usually no relation with the code model. It is important to realize that symbols linked to the code model (for example Class symbols in class diagrams) only *visualize* an entity (class) in the code model. The same class can be visualized many times in multiple diagrams in different styles depending on the context. Diagrams are created and maintained in the Diagram list view. The actual diagrams are edited in the Diagram Editor view. There's more in chapter Diagrams, page 39

Diagrams can be ***exported*** as image, native XML format or in XMI format (third party plug-in expert).

The Environment and Project options are a first means to ***Customizing*** ModelMaker to your taste or coding style.

In Chapter "Customizing" page 80 there's more on customizing ModelMaker.

ModelMaker ***Projects*** or ***Models*** contain both Code Model and Diagrams. In ModelMaker you work on a single model at a time. Opening a different model will close the model you were working on.

Working with models requires care in the areas ***team development*** and model (system) ***boundaries*** as described in chapter Team development. Model boundaries and Version Control, page 43

The ***Difference*** view is used to compare a model unit to the associated file on disk. Most powerful is the structured difference that does a syntactical comparison rather than a plain file based comparison. Also use the Difference view to compare any disk file or model unit with any other file or model unit or to compare two classes.

Code Model contents

A ***Class*** is the most important entity in the Code model, it matches the corresponding concept in Delphi and it is a container of Class Members. Classes always have an ancestor (super) class and sometimes have descendent (sub) classes. The default class ancestor `TObject` is always present in the model.

An ***Interface*** is similar to a class, as it matches the corresponding concept in Delphi. It is also a container of a restricted set of Class Members. Interfaces always have an ancestor (super) interface and sometimes have descendent (sub) interfaces. From Delphi 6 onwards there are two interface roots; IUnknown and IInterface. These default interface ancestors are always present in the model. Because classes and interfaces are so similar, in the remainder of this manual usually where you read class you can also read interface. Both are

Class Members are the fields, methods, properties and events making up a class's interface. They always belong to a class (or interface).

Fields, Methods and ***Properties*** match the corresponding concepts in Delphi. Fields are used to store a class's state and/or data. Methods are used to implement behavior. A method's implementation consists of ***sections*** of code. This allows ModelMaker to locate specific code within the method's body. Properties let you have controlled access to a class's attributes as though they were fields. ***Events*** are a special kind of properties. They are used to represent method pointer type properties (delegates). This way ModelMaker makes the same distinction as Delphi's Object Inspector does. It is possible to create a *property* of type `TNotifyEvent` and ModelMaker will generate the correct code for the property, but ModelMaker will not recognize this property as an event. Therefore: use Events rather than properties to model event types.

Event type definitions are used to define the signature of event type properties. ModelMaker relies on these definitions to create and update event handler methods and event dispatch methods. The most used event type `TNotifyEvent` is automatically inserted in each model. Event types are maintained in the Events view.

Design patterns and ***Units*** as described earlier complete the code model contents.

Diagrams

ModelMaker supports a set of UML diagrams:
1. Class diagram or static structure diagram
2. Sequence diagram
3. Collaboration diagram
4. Use case diagram
5. Robustness analyses diagram (not defined in the UML)
6. Activity diagram
7. State chart diagram
8. Package diagram or Unit dependency diagram, a static structure diagram, just showing unit package symbols.
9. Implementation diagrams: Deployment diagram and Component Diagram
10. Mind Map diagram (not defined in the UML)

The basic elements of diagrams are ***symbols*** and ***associations***. The meaning and attributes of the symbols and association used are according to the UML specification. This manual will not explain the meaning and details of each symbol. In the ModelMaker on-line help you'll

find a short description for each symbol and association and it's attributes. There are a number of good books available on the UML. Alternatively you could download the latest version of the UML specification as available for free on the Rational web site.

The Chapter "Diagrams" contains a detailed description of organization and editing of diagrams in ModelMaker.

Working with models

In this chapter, a model is the equivalent of a ModelMaker project that contains both Code Model and Diagrams.

Model files

Native ModelMaker will save a model into set of files:

1. <model>.mpr; contains the project settings.
2. <model>.mma; contains the project related macros.
3. <model>.mmb; contains the code model data
4. <model>.mmc; contains the documentation for the model.
5. <model>.mmd; contains the diagrams.
6. <model>.mme; contains the event type definitions.
7. <model>.mmf; contains the project messages.

If you manage your source files using a version control system, you should add these model files to version control too.

However, ModelMaker is able to bundle the project files into a single project bundle: *.mpb. To enable this, check the option 'Bundle project files' in “Options|Environment|General”. This option is checked by default. If this option is checked, the file Open and Save dialogs will have the *.mpb file type as well as the *.mpr. Using single file bundles makes it easier to work with ModelMaker projects. To convert existing projects to single file project bundles, use File|Save as and manually change the .mpr extension to an .mpb for the project. In very large projects you may find that saving and loading bundles takes some more time than the multi file projects. When using project bundles it is sufficient to add the <model>.mpb file to version control.

ModelMaker cooperates with version control systems by not allowing you to save a project that exists with read-only file attributes. Note that only the file <model>.mpr or .mpb is checked for read only attributes. In an unbundled project file the *.mma.*.mme files are not checked.

Model templates

As you probably have noticed (for example in the Getting Started demo), one of ModelMaker’s powerful features is that it’s easy to override methods and properties and that changes are automatically propagated down the inheritance tree. But to let this work, the

ancestor class and the methods and properties to override must be part of the model. So it is important with which (new) model you start.

Now you may ask: why didn't I get the complete VCL as default model? This would contain all classes I ever need! The answer is that this would result in very large models, through which it is hard to navigate. We have been working with ModelMaker for quite a few years now, and it shows that it is most practical if models contain classes of a single domain only. Classes contained in a single component package typically reside in single model too. Generally this result in models typically containing 5 to 20 classes or may be up to 50 for really large models.

And that's where you need templates. Templates are just ordinary models containing classes for a certain domain that you use to derive new classes with in a certain domain. Since it's possible to have as many templates as you like, you can create nice compact templates for each relevant sub-domain: like a template for creating simple components, one for simple TCustomPanel descendants etc.

You load a template by selecting 'File|New from template' which will load the template model and then reset the model's name to 'untitled'. Any model can be used as a template, but by design ModelMaker looks in the `[installdir]\TEMPLATE` folder for template models.

ModelMaker has one special template that it uses as default. This is the model `[installdir]\DEFAULT.mpb`. You may open this model and change it to your needs or overwrite it with another template model.

Editing a model

To edit a model, ModelMaker has multiple views on the model. Most of these views are interlinked: selecting something in one view will show related information in another. Interlinking is based on:

- The current *class*, selected in the Classes view or Diagrams view. For example, the Class Members view displays the members of the current class.
- The current *class member* (if any), selected in the Class Members view.
- The current *method* (if any), equal to the current class member if that is a method. For example, the Method Implementation view displays the implementation of the current method.
- The current *unit*, selected from the Units view.

The other view like the Macros view and the Event Library view are (more or less) independent from these selections.

In the Environment options Navigation tab you'll find options to synchronize and activate views on certain events.

Ownership in ModelMaker

ModelMaker assigns an *owner* to each entity in the model. Entities can be anything from classes to methods or a section of code in a method. The owner of an entity created the entity and has exclusive rights to update or delete it whenever suitable. Usually you, 'User' will be the owner since you create most classes, class members etc. ModelMaker does not discriminate between users: it does not remember that John created this class and Mary that property.

Deleting an entity will automatically delete all entities it owns too. For example: a property owns it's read access method. You cannot delete or edit these methods, other than by deleting or editing the property.

Team development, Model boundaries and Version Control

In ModelMaker team development support and model boundaries are related issues as they both deal with the question: what should and what should not be in a model. There are a few important reasons to use multiple relative small models according to logical boundaries rather than one big model containing all classes and diagrams you have.

1. It enables team development: while one developer works on one model that is part of a larger project, the other can work on another. There are limited possibilities to merge changes made to the same model. Therefore only one developer can work on a single model at the time.
2. It improved ease of navigation and overview
3. It improves performance.

Usually you'll find logical boundaries to split up models - and usually well-designed modules (as in units or classes) have low coupling and dependencies. Boundaries could be: units in a certain Delphi package, units containing classes that perform a related task etc. In ModelMaker itself for example, we have lots of models, sometimes only containing a single unit. Large models for example contain all classes related to diagrams, the entire code engine or the source importer. And yet other models contain units that are used in many projects: timers, filters etc.

Although we have some happy customers that have models containing 300+ classes in 150+ units, practice shows that a good model size is about 1 to 30 classes in 1..10 units.

The drawback on having multiple smaller models is that ModelMaker maintains active relations only within the same model and not beyond model boundaries. As a result you occasionally might need to re-import a unit after it has been altered in another model.

It's a good idea to use a version control system and put the model files under version control too. Although a model contains everything that is needed to (re-)generate the source files it

contains, you should ***always*** store the actual source files under version control too. It’s from the source files that you build your product, not the model.

Merging models is only partly supported: source code can be generated and imported and diagrams can be exported/imported. However no code model meta-information can be exported/imported. Because ModelMaker stores its data in a native binary format, you cannot use the merging capabilities of a Version Control System without corrupting the model.

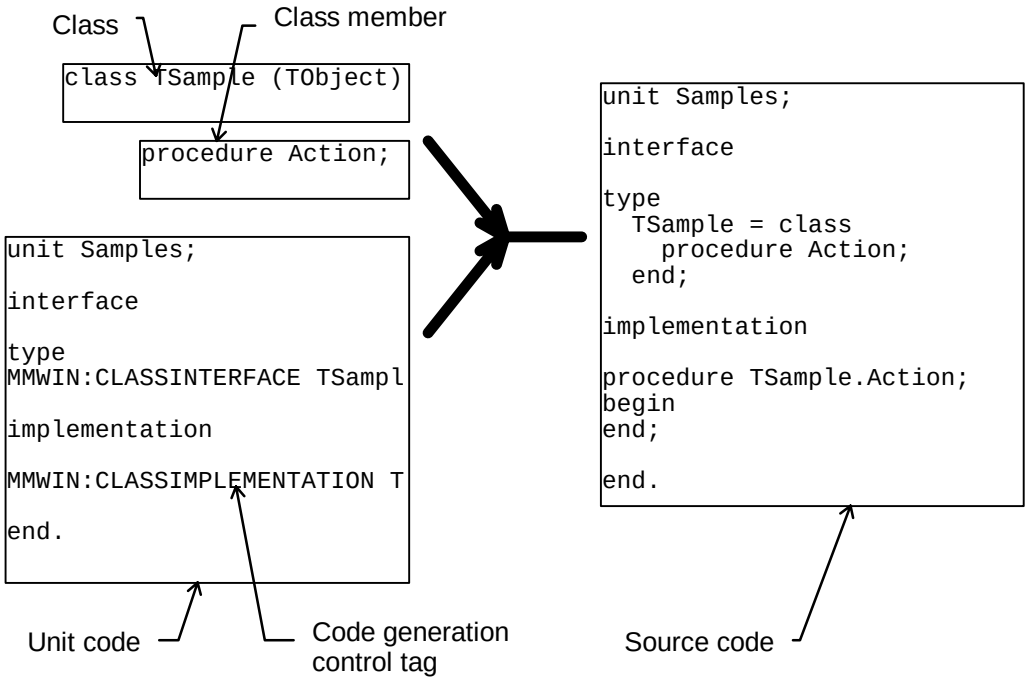
If models do get out of sync with the source code or other models you've always got the Difference View with it's powerful structural difference function to help getting the model synchronized with source or other models.

Using source aliases rather than hard coded directories is a must in team development. Check chapter Source Aliases, page 52 on source aliases.

Generation source code

Overview

In ModelMaker units are the gateways to source files on disk. During code generation, unit code, classes and class members from the code model are combined into a source file. A unit's unit code contains (can contain) code generation control tags. At the position of a tag ModelMaker will insert the associated entity such as class interface or implementation. Here



is a picture that visualizes this process.

The unit code is read line-by-line and scanned for code generation control tags. If a line contains a code generation control tag, the entity as defined by that tag is inserted instead of the tag. Any lines not containing code tags are just copied to the source file.

During code generation macros in both unit code and method implementation code will be expanded using the predefined macros and the project and environment macros you define yourself. Macro expansion and line formatting are the last stages in the source code generation process for both text generated from code tags and text just copied from a unit's unit code. Macros are explained in detail in chapter Macros, page 74

During generation of the implementation section, class separators, method separators and method section separators can be emitted. This is controlled by code generation settings and macros.

Insertion of in-source documentation can also be part of the generation. This is explained in detail in chapter “In source Documentation” , page 67.

Code generation control tags

ModelMaker uses code generation control tags to control source file generation. Only code generation control tags placed in the unit code are interpreted. Tags in a method’s implementation are not interpreted. Here are the rules that apply to code generation control tags:

1. Code generation control tags are *case insensitive.*
2. All code generation control tags start with `MMWIN:` at the *first* position of a line.
3. Code generation control tags must reside on a *single line*.
4. Any *semi-colons* or *equal* signs defined in a tag are obligatory.
5. Code generation control tags can contain any white space after the `MMWIN:` definition. For example: `MMWIN:STARTINTERFACE` is the same as tag `MMWIN: START INTERFACE`

Class related tags

These code generation control tags are used to define the insertion position of class and interface related code:

```
MMWIN:CLASS INTERFACE classname ;ID=###;
MMWIN:CLASS IMPLEMENTATION classname ;ID=###;
MMWIN:CLASS REGISTRATION classname ;ID=###;PAGE=vcl page name
MMWIN:CLASS INITIALIZATION classname;ID=###; // OBSOLETE
```

These class related tags are automatically inserted and maintained by ModelMaker whenever you add a class to a unit or remove it again. Normally you would use the Unit editor dialog or drag and drop in the Units view to insert or delete classes in/from a unit or change the relative position within a unit. However, in special cases you can manually move these tags to any other position in the unit code. This is for example useful if you want the interface of a class to reside in the unit’s implementation.

In these tags ModelMaker ignores the ‘`classname`’ and just uses the ‘`ID=###`’ tag to identify the class. The class name is inserted just for your convenience. Modifying it will have no effect.

Normally, for classes both the `CLASS INTERFACE` and `CLASS IMPLEMENTATION` tags should be put in the unit code. If either one is missing after you’ve edited the unit code manually, you’ll get a warning. In special cases - for example in include or documentation files - you may manually remove either the interface or implementation tag. Interfaces (as opposed to classes) ignore the `IMPLEMENTATION` tag.

The `CLASS REGISTRATION` tag is optional and is used to insert a snippet of code to register the class as component. You may manually remove them from the unit code. You can manually

edit the ‘`vcl page name=...`’ text in the registration tag to change the VCL registration page. However, usually you would do this in the unit editor dialog. If you remove the registration tag or the page name, no registration code will be generated for the class.

When importing a unit containing a procedure Register, the registration code is automatically converted to tags.
The tag `CLASS INITIALIZATION` is obsolete from version MMv6.0 onwards. This tag is maintained for backward compatibility only. The tag is used to insert initialization code for `TStreamable` descendants: `RegisterStreamable(..);` If you remove the initialization tag, no initialization code will be generated for the class.

Event type declaration tag

This code generation control tag is used to define the insertion position of an event type declaration.

```
MMWIN: EVENT DEFINITION eventname type declaration; ID=###;
```

This declaration is maintained by ModelMaker and is obligatory for each event definition in a unit. Normally you use the unit editor dialog or drag and drop from the Events and Units view to insert or remove event type definitions in/from a unit, or change their relative positions within a unit. However, you may manually move these tags to any other position. If you remove them, the event type definition is also removed from the unit. The ‘`eventname`’ and ‘`type declaration`’ texts are ignored, only ‘`ID=###`’ is used to identify an event type definition.

Editing marker tags

These code generation control tags are used to mark the positions at which you want ModelMaker to insert the first class or event type declaration in a unit.

```
MMWIN:STARTINTERFACE
MMWIN:STARTIMPLEMENTATION
```

These tags are for editing purposes only and they have no role in the code generation process. There’s one exception to this. The `MMWIN:STARTINTERFACE` tag is also used to determine the insertion position of class forward declarations - if any. If this `MMWIN:STARTINTERFACE` tag is absent, class forward declarations will be inserted before the first event type declaration or class interface, which ever comes first.

Macro expansion control tags

These code generation control tags are used to switch on and off macro expansion during code generation:

```
MMWIN:START EXPAND
MMWIN:END EXPAND
```

By default the expansion is switched ON. You need these tags if your unit or method code contains the text "<!" ("" not included). The macro expander will interpret the sequences

```
"<!" + Identifier + "!>" on a single line
"<!" + Identifier + "(" param list + ")" + "!>"
```

as a macro. "Identifier" (the macro name) can consist of characters ['0'..'9', 'a'..'z', 'A'..'Z', '_']. Which is similar to Object Pascal identifiers, although macro names can start with a number. White space surrounding the identifier is ignored.

Because macros can be in any text including comments and strings, this would make it impossible to generate code for units that contain a valid macro sequence <!ident!>.

There are a few workarounds for this problem. The most sensible uses these control tags in the unit code - remember the tags do not work in method code!

```
MMWIN:ENDEXPAND
const
  HTMLCommentStart = '<!';
  HTMLCommentEnd = '!>';
MMWIN:STARTEXPAND
```

In the rest of the code you can now use the constants and still leave the macro expansion on.

Note that these tags do not affect generation of in-source documentation, which is also based on macros. Documentation related macros are always expanded regardless of the setting of these switches.

Unit documentation tag (obsolete)

This code generation control tags defines the insertion position of a unit's documentation. Because inserting unit documentation is much better controlled with the 'In-source documentation generation' options, we recommend avoiding the use of this tag although it's still supported for backward compatibility.

```
MMWIN:INCLUDE UNITDOC;INDENT=##;
```

The tag '`INCLUDE UNIT DOC`' defines the position at which ModelMaker will insert the unit's documentation. The unit's documentation can be edited in the Documentation view. The '`INDENT=##;`' extension is optional and may be omitted. This defines an indention for the documentation of ## spaces. A typical use would be:

```
{
MMWIN:INCLUDE UNIT DOC;INDENT=2;
}
unit <!UnitName!>
```

Obsolete tags

These code generation control tags are obsolete from MMv6.0 onwards.

```
MMWIN:STARTREGISTRATION
MMWIN:STARTINITIALIZATION
```

On loading a model, ModelMaker will remove these tags from the unit code. If you add them manually they will simply be ignored.

Code generation options

In the Project options|Code Generation tab you will find options that control code generation:
1. Formatting the layout of source code
2. Sorting of class members and method implementations
3. Inserting a (custom) class header to that precedes the class's method implementations
4. Inserting a (custom) method separator to that precedes each method's implementation
5. Inserting a (custom) method section divider in between each section.

The on-line help file explains the meaning of these options.

The Project options|Source Doc generation tab controls the generation of in-source documentation. This is explained in detail in the chapter on "In source Documentation", page 67.

Maintaining Code Order / Custom member order

ModelMaker supports a user definable custom member interface and method implementation order.
These custom orders can be assigned during import used during code generation. When that is done, the effect is that ModelMaker will maintain the imported code order during generation.

This is how it works:
In the Project options Code Generation tab you define a member sorting scheme in class interface and implementation generation. In these sorting schemes Custom orders can be used as a grouping or additional sorting property.

When ***grouped*** on custom order, members will be sorted according to the (original) custom code order. Members that have been added later with an unspecified custom order, will be placed after all members with a specified order.
When using the custom order to perform ***additional sorting***, the default sorting scheme will be applied and within each 'section' (visibility etc.), the custom order will be applied.
If class members have an unspecified order (which is the default for new members), the effect of enabling Custom Order during Generation or in the Members list is null.

Custom orders can be assigned during Import. In the Project options|Code Import tab you'll find settings to enable / disable assigning the custom order during import. The import dialog allows temporarily overruling these settings.

If the Importer assigns custom orders and Generation uses grouping on Custom Order ModelMaker will effectively maintain the original code order during 'refresh import' and append new members at the end.

Additionally Custom orders can manually be defined - per class - with the "Members custom order" dialog or with the Members view 'Rearrange mode'.

The Rearrange dialog is available from the Members and Classes view 'Wizards' local sub menus and from the Units view 'classes' local sub menu. In this dialog you either you drag and drop to rearrange a class interface and method order or use one of the predefined sorting schemes. The dialog can be also used to clear an interface or implementation custom order. Note that manually defining a custom order will erase an imported code order.

The Members view has a 'Rearrange mode' (members local menu). In this mode an interface custom order can be assigned using drag and drop in the members view itself. For method implementation order you must use the Rearrange dialog. In the in the rearrange mode the Members view filter settings (visibility, type, category) are ignored to make sure all members are displayed. Also Members view grouping is implicitly set to "Custom Order" and sorting is predefined and cannot be changed. As a visual feedback, the background of the members is changed to a silver color in this mode.

Adjusting the unit template

Whenever you add a new unit to a model, ModelMaker looks for the file `DEFUNIT.PAS` in the folder `ModelMaker\6.0\BIN`. The default unit may also be (re-)defined when adding a new unit. This text file is used as a template for the newly created unit code. You may edit this file to your needs. You can use these code generation control tags to mark the insertion position for the first class ModelMaker will insert in the unit:

```
MMWIN:START INTERFACE
MMWIN:START IMPLEMENTATION
```

You might want to adjust the default unit template in order to:

- Customize the **`uses`** clauses in the unit **`interface`** and **`implementation`**.
- Add a company-defined header.
- Control macro expansion.

As an example: here's a unit template we use for freeware units. In the chapter Macros the same template extensively using macros is shown.

```
{
  File      : <!UnitName!>
  Version   : <!Version!>
  Comment   : <!Comment!>
  Date      : <!Date!>
  Time      : <!Time!>
  Author    : <!Author!>
  Compiler  : <!Compiler!>
```

```
+-------------------------------------------------------------------------+
| DISCLAIMER:                                                             |
| THIS SOURCE IS FREEWARE. YOU ARE ALLOWED TO USE IT IN YOUR OWN PROJECTS |
| WITHOUT ANY RESTRICTIONS. YOU ARE NOT ALLOWED TO SELL THE SOURCE CODE.  |
| THERE IS NO WARRANTY AT ALL - YOU USE IT ON YOUR OWN RISC. AUTHOR DOES  |
| NOT ASSUME ANY RESPONSIBILITY FOR ANY DAMAGE OR ANY LOSS OF TIME OR MONEY|
| DUE THE USE OF ANY PART OF THIS SOURCE CODE.                            |
+-------------------------------------------------------------------------+
}

unit <!UnitName!>;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
MMWIN:START INTERFACE

procedure Register;

implementation

uses StrUtils, NumUtils;

procedure Register;
begin
end;

MMWIN:START IMPLEMENTATION

initialization
end.
```

Notice how this template contains (from top to bottom):

- A simple standard header and a free ware disclaimer.
- Some statistics macros, like `<!Date!>` and `<!UnitName!>`
- The basic unit structure:
  `unit``..`**`interface`**`..`**`uses`**`..`**`implementation`**`..`**`uses`**`..`**`initialization`**`..`**`end`**`.`
- A macro `<!UnitName!>` to define the unit's name.
- The **`procedure`** `Register;` definition and implementation for registering components.
- A default uses clause in the unit's implementation to include some often-used units `StrUtils` and `NumUtils`.

Unit Time Stamp Checking

The ModelMaker code generator by default uses Time Stamp checking to prevent overwriting source files that may have been changed outside ModelMaker. It checks if the file on disk is newer than the last time a unit was generated. If this is the case you'll be warned that you are about to overwrite that modified file.

You can switch on and off time stamp checking in the Environment options|General tab.

There is a limitation on time stamp checking you must be aware of:

1. If you rename a unit in ModelMaker the time stamp is not reset to 'unknown', so if you have an existing file on disk which is NEWER than the last time the unit was generated (with the old name) you will NOT get a warning. The Unit editor dialog warns you for this (file xxxx already exists, overwrite?), but the in place editor in the Units view does NOT.

The Unit difference View displays the time stamp comparison on activation. The function "Check Time stamps" refreshes this comparison.

Source Aliases

ModelMaker supports source path aliases in units to avoid hard-coded directories and make your models machine-independent. An alias is associated with an aliased directory, similar to database aliases. In the model a unit's alias is saved rather than the aliased directory. On each machine aliased paths can be defined differently. This allows you to transport models to other machines.

Source aliases are a must in team development.

Example:

Suppose on machine A you have a source directory:
`C:\DATA\PROJECTS\COMMON`
An alias defining this directory could be COMMON.
On machine B, COMMON could de defined as
`\\PROJECTDATA\COMMON`

You add aliases from the main menu "Options|Source aliases", or from the Units view popup menu.

To avoid that you need to mimic a large directory structure in aliases, unit names can be relative to an alias. That way you only need to define a few aliases for root paths.

Here's an example:
Suppose you have a directory structure which looks like this:

C:\Project1\App_a\source
C:\Project1\App_a\components
C:\Project1\App_a\utils
C:\Project1\App_b\source
C:\Project1\App_b\components
C:\Project1\App_b\utils
C:\AllProjects\source
C:\AllProjects\components
etc.

You could define three aliases
App_a = C:\Project1\App_a

App_b = C:\Project1\App_b
AllProjects = C:\AllProjects

A unit named C:\Project1\App_a\source\Samples.pas could then use

Alias = “App_a” (omit the “”)
Relative unit name = “source\Samples.pas” (omit the “”)

Source code aliases are also used to offset the Import source code dialog's initial directory. If you define an alias VCL Source for C:\Program files\Borland\Delphi 3.0\Source\Vcl, the import dialog can be offset to this directory by simply selecting this alias from the drop down menu.

Source aliases also participate in Version Control integration. See next chapter.

Version Control support and Aliases

By using a plug-in VCS Expert you can add Version Control capabilities to ModelMaker. Check the ModelMaker Tools web site for ready available third party VCS Experts or create your own using the MMToolsApi VCS interface.

If a VCS expert is installed, in the units view popup menu and main file menu VCS related menu items are available to manually check-in/out a model or unit. Also each time a read-only unit is about to be generated an attempt is made to check the unit out (after your confirmation). VCS Experts can add more VCS related commands to the popup menu such as ‘Add to project’, ‘History’ etc.

Usually VCS systems usually need a VCS project name to perform an operation. In ModelMaker Source aliases are used to store the user definable VCS projects.

Each source alias can (but does not need to) store a Version Control project. This string is passed on to an installed VCS integration expert whenever a VCS file related action is performed.

If you install a VCS expert in ModelMaker, the ModelMaker IDE integration experts will use the same expert to integrate VCS in the Delphi IDE.

For details, refer to your VCS system and MM VCS expert provider.

Using ModelMaker to generate Instrumentation code

ModelMaker supports generating Method instrumentation. This feature makes ModelMaker suited for generating instrumentation code for CodeSite, GpProfile and other tools instrumenting source code on a method base for profiling, tracing etc.

Method instrumentation generation is controlled with the option 'Instrumented' in the Method editor dialog. If this option is checked instrumentation code will be generated for the method. Instrumentation can be (de-) activated for all units at once with "Project options|Code generation|Instrumentation". The actual instrumentation code is defined by two macros you must add manually to the environment or project macro list: “MethodEnterInstrumentation” and “MethodExitInstrumentation” (omit the “”). These macros are expanded before the first and after the last section in a method's main body. For example (note the use of the predefined macros ClassName and MemberName in these macros)

macro MethodEnterInstrumentation=

```
CodeSite.EnterMethod('<!ClassName!>.<!MemberName!>');
try
```

macro MethodExitInstrumentation=

```
finally
  CodeSite.ExitMethod('<!ClassName!>.<!MemberName!>');
end;
```

When this is applied to:

```
procedure TSample.DoSomething;
begin
  ShowMessage('Doing something');
end;
```

This will result in the following code:

```
procedure TSample.DoSomething;
begin
  CodeSite.EnterMethod('TSample.DoSomething');
  try
  ShowMessage('Doing something');
  finally
    CodeSite.ExitMethod('TSample.DoSomething');
  end;
end;
```

To avoid creep when re-importing instrumented methods you can use the code remove tags. Check chapter “START and REMOVE tags, page 59 for details:

```
MMWIN:>>STARTREMOVE
MMWIN:>>ENDREMOVE
```

These tags may be part of a comment. Depending on the setting in the Project options|Code import tab the importer will filter out any code in between a start remove / end remove pair.

The macro MethodEnterInstrumentation using these tags would for example look like:

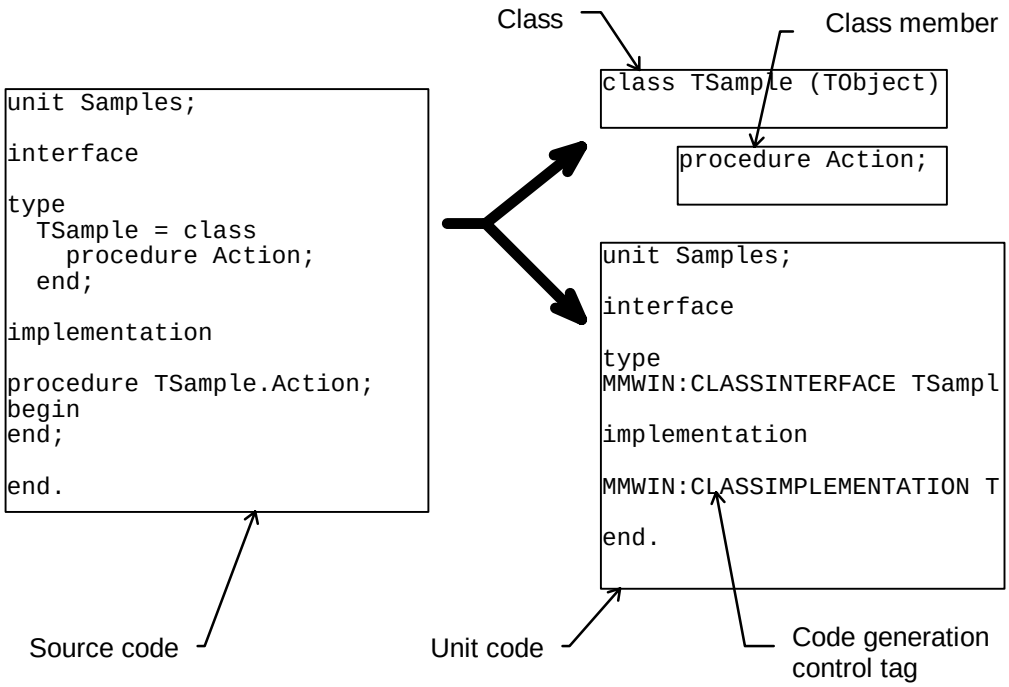
```
//MMWIN:>>STARTREMOVE
CodeSite.EnterMethod('<!ClassName!>.<!MemberName!>');
try
//MMWIN:>>ENDREMOVE
```

The Member Manipulator can be used to switch on and off Instrumentation for multiple methods at once. On the ModelMaker Tools web site you'll find a third party plug-in Instrumentation expert. This expert is dedicated to controlling method instrumentation code.

Importing source code

Background

ModelMaker imports Delphi Object Pascal source files. This process is basically the inverse from generating a source file from classes, members and unit code. The class related code is converted into Code Model entities such as classes, interfaces, members and method implementations. All non-class or interface related code is moved into a ModelMaker unit's



unit code.
This process is called reverse engineering.

It is important to realize that if an imported class is not currently existing in the model, members and code sections are inserted as 'User created/owned" and no attempt is made to extract meta information, such as applied patterns, inherited calls etc. The only exception to this, are the read and write access members of properties, which are restored and linked to the property. For example: all code inserted by patterns is read back, but marked as "user" and the patterns itself is not recreated. The same applies for inherited method calls etc. In general you loose the meta information.

However, if an imported class already occurs in the model it is 'refreshed' and all meta-information is restored. Even applied design patterns can be traced back.

Therefore importing source code is great for:

- Importing (an interface of) a class you want to inherit from or use as a client.
- Importing existing code originally not developed with ModelMaker
- Updating an existing class from source code

However it is advised that once imported, you keep editing your code in ModelMaker. The only exceptions to this are form and (other resource module's) source files which by their nature you (partially) need to edit in Delphi.

Importing a source file

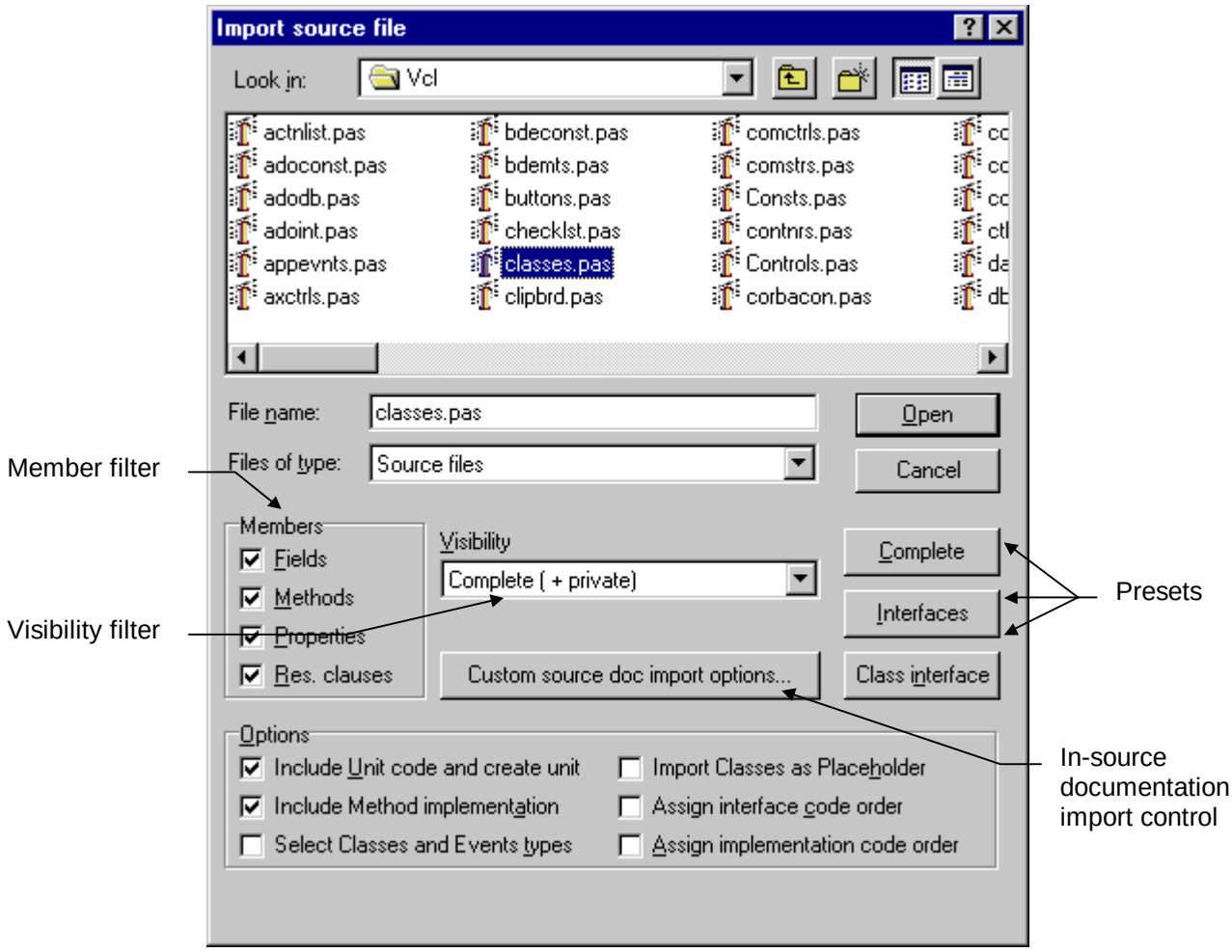
The main toolbar and units view pop-up menu contain buttons and commands to
- Import a source file (in the current model or a new model).
- Refresh import source file.

Additionally you'll find import functions in the Classes view (refresh class or associated unit) and the Difference view (refresh unit, class or method).

In the Project options|Code Import tab you'll find the options to control source code import. Check the on-line help file for a detailed description on each option. The project options are used for all (refresh) imports except when you use the Import dialog to interactively import a file. The import dialog allows temporarily overruling some project import control options. The import dialog has some other options that by their nature are not in the project options, typically related to initial first time import. The options in the import dialog are preset (each time!) to the project options.

The import dialog's initial directory is pre-set by selecting a source code alias. You'll find more on (defining) Source code aliases on page 52.

When importing a source file using the import dialog, you enter a source file name and set filters and options to control the import. Most options are rather self-explanatory.



If the option “Include Unit code and create unit” is checked, not only the classes will be imported, but also the non-class related unit code. Check this option if you want a complete import. If you just need a class’s interface this option can be unchecked.

If the option “Select Classes and Events to import” is checked, you may select which of the classes or events found in the source file will actually be imported in the model. Useful if you’re not interested in all classes and events contained by a unit.

If the option “Import Classes as Placeholder” is checked all imported classes will be marked placeholder. If it is unchecked classes will remain their current state or ‘real’ if not found in the current model.

There are three pre-set buttons, which set the filters to a “complete”, “interfaces only” “class interfaces” mode. The default settings (as displayed) are those for a complete import.

The In-source documentation import control settings are explained in detail in chapter “Importing a source file” page 57.

Importing (adding) versus Refreshing

'Refresh Import' and Import as in 'Add to Model' act similar - the difference is how existing units and classes are treated.

Importing with the 'Add to Model' function (in the IDE or ModelMaker) will add non-existing units or classes contained in the added unit. If either unit or class already exists in the model it will be refreshed.

'Refresh Import' issued from the IDE integration expert will only refresh the unit if already existed in the model and will not add a unit not currently in the model. In ModelMaker you can only refresh an existing unit.

Avoiding creep - removing code during import

ModelMaker supports removing certain fragments of code during import to avoid creep in a full generation / re-import cycle.

ModelMaker generated default class separators such as

```
{
**************************** TSample ****************************
}
```

and ModelMaker generated default category markers such as
{<<Category>>: Model linking}

are automatically removed.

Additionally ModelMaker will remove the following code:

1. All code marked by a MMWIN:>>STARTREMOVE and MMWIN:>>ENDREMOVE pair.
2. All comments with the 'removal signature' as defined in the Code Import options.
3. Matched (and optionally unmatched) comments according to the documentation style comment.

For importing "In source documentation" and removing comments with the documentation signature(s), refer to chapter on "Importing in source Documentation", page 68.

STARTREMOVE and ENDREMOVE tags

You can use these tags in any code, comment or string to instruct ModelMaker to remove all code between the tags including the tags themselves:

```
MMWIN:>>STARTREMOVE
MMWIN:>>ENDREMOVE
```

The corresponding setting in the Project options|Code import tab will enable/disable filtering based on these tags.

These tags are commonly used to remove (part of) a macro that was generated by ModelMaker from the input file. Check the chapter on generating Method Instrumentation code for an example.

Comments with remove signature

If you want to remove certain comments from the source file you can use comments with the Removal Signature. This signature is defined in the Project options|Code import tab which also enables and disables removing comments with this signature.

You will need to use this type of comment remove filter if you
1. Define a custom class separator,
2. Define a custom method separator,
3. Define a custom method section separator
4. Redefine the category expansion macros 'IntfCategory' or 'ImplCategory'

Assuming {- } to be the removal style comment, a custom class header should look like

```
{-
****************************
*
*            <!Classname!>
*
****************************
}
```

Similar, a custom category expansion tag could look like
```
{- Category: <!Category!> }
```

Note that Method End Documentation is automatically removed due to the fact that the importer will remove the method including the line containing the method's final **`end;`**.

Import restrictions and limitations

ModelMaker usually imports in about 99.9 % of all cases without problems. If ModelMaker generated the source file, the imported code is usually 100 % correct. ModelMaker uses a combination of syntactical analyses and line based extraction to support importing of code that is not entirely syntactically correct.

ModelMaker's import mechanism imposes the following restrictions on source files in order to be imported correct. ModelMaker's importer uses the same parser as the ModelMaker Code Explorer. The ModelMaker Code Explorer that integrates in the Delphi IDE will display a list of parse errors. That way it acts more or less as tool to check code before importing into ModelMaker.

Class and Interface interfaces

Restrictions in class and interface declarations.

Any comments, compiler directives and white space in a class's interface are/is ignored except in method parameter lists.

Comma separated Field declarations are converted into separate fields:

```
FA, FB: Integer;
```

is imported as

```
FA: Integer;
FB: Integer;
```

Procedure or method pointers that are not defined as an type are not imported correct. The work around is to use a type definition.

The following code causes import errors.

```
TSample = class(TObject)
  FEvent: procedure of object;
end;
```

Which can be replaced by this code that will import correct:

```
TMyEvent = procedure of object;

TSample = class(TObject)
  FEvent: TmyEvent;
end;
```

Method implementation

Restrictions in method declarations.
Any comments, compiler directives and white space in are/is ignored except in parameter lists.

Local variables immediately following the method declaration will be converted to ModelMaker method variables. If local variables for example are preceded by a **`type`** or **`const`** declaration, they will be added to the method's local code section, just like all other local code for that method.

In the following example the local vars. I, J and S will be converted to ModelMaker local vars., the const declaration and procedure CheckIt will be placed in the method's local code section.

```
procedure TSample.Action;
var
  I, J: Integer;
  S: string;
  const CheckSum = $AAAA; // this will go into local code
  procedure CheckIt;
  begin
  end; // this is the end of the local section
begin
  CheckIt;
end;
```

In the following example the local vars. I, J and S will be placed in the method's local code section together with the const declaration:

```
procedure TSample.Action;
const CheckSum = $AAAA; // this will go into local code including the vars
var
  I, J: Integer;
  S: string;
  procedure CheckIt;
  begin
  end; // end of local code section
begin
  CheckIt;
end;
```

During refresh import of a method already existing in the model, the importer will leave the code sections intact wherever possible. If the importer cannot locate a non-user owned section of code, it will simply leave the section in the method and give a warning.

Comments and white space

The following table shows how ModelMaker treats comments and compiler directives

Comment or compiler directive in:	Import result
Class interface	Ignored
Method header	Ignored except in parameter list
Local vars.	Ignored
Method local code and body	Copied to method
All other code	Copied to unit code

Check the ModelMaker generated default class separators such as

```
{
**************************** TSample ****************************
}
```

and ModelMaker generated default category markers such as
{<<Category>>: Model linking }

are automatically removed.

If you define a custom class separator, method section separator, or redefine for example the category expansion macro TODO, you should use the remove style comments to avoid creep during import. Check paragraph " Comments with remove signature", page 60.

Unsupported language constructs

Include files are not read during import, so if you find yourself thinking: "where's my method implementation gone?" you probably need to add the included files to the imported unit using a text editor and re-import the file.

Compiler directives in class interfaces are not supported and are a potential problem source. Using inheritance may sometimes solve this. Worst case you need to create two units.

Compiler directives *around* method implementations are not supported. Placing the directives inside the method can solve this:

```
{$IFDEF DEMO}
procedure TSample.Action;
begin
end;
{$ELSE}
procedure TSample.Action;
begin
  { actually do something useful }
end;
{$ENDIF}
```

Won't be imported correct, but can be replaced by the following code which will be imported fine:

```
procedure TSample.Action;
begin
{$IFDEF DEMO}
{$ELSE}
{ actually do something useful }
{$ENDIF}
end;
```

The importer matches **`begin..end try..end, case..end`** pairs etc. to locate methods. Because conditional defines are not interpreted, using conditional defines you can create code that will compile correct but will not import correct. In fact The Delphi IDE background compiler uses a similar mechanism and will not be able to function properly either when inserting new methods in code completion or creating a new event handler.

This code for example will confuse the importer's begin end matching. The method will not be imported correct.

```
procedure TSample.Action;
{$IFDEF DEMO}
var
  S: string;
begin
  S := 'Demo';
  ShowMessage(S);
{$ELSE}
begin
{$ENDIF}
end;
```

You can replace the previous code by the following code that will import correct and as a side effect allows the Delphi IDE to stay on track too:

```
procedure TSample.Action;
{$IFDEF DEMO}
var
  S: string;
{$ENDIF}
begin
{$IFDEF DEMO}
```

```
S := 'Demo';
  ShowMessage(S);
{$ENDIF}
end;
```

Pure assembler methods are not supported:

```
procedure TSample.Fast; assembler;
asm
end;
```

Expressions in an indexed property's index specifier are not supported:

```
property FirstPicture: TBitmap index BM_USER + 0 read GetPicture;
property SecondPicture: TBitmap index BM_USER + 1 read GetPicture;
property ThirdPicture: TBitmap index BM_USER + 2 read GetPicture;
```

This can be solved like:

```
property FirstPicture: TBitmap index BM_FIRST read GetPicture;
property SecondPicture: TBitmap index BM_SECOND read GetPicture;
property ThirdPicture: TBitmap index BM_THIRD read GetPicture;
```

Conversion errors

Any import conversion errors or warnings will be displayed in the Message View. The messages may be printed, saved etc.

Not reported conversion errors are:
1. Minor changes in property access method parameters lists.
2. Positioning of code generation tags in the unit code.

The best thing to do after importing a complex unit, is to perform a Delphi syntax check on the re-generated unit. From our experience it shows that if there are any remaining errors, they will evolve here.

Another option is to make a file based difference in the Difference view between the imported unit and the original source file. You should make sure that Code generation sorting scheme matches the scheme used in the original file. You might need to import Custom Code order and use the same order during generation to maintain code order. Check chapter "Maintaining Code Order / Custom member order" on page 49.

Note that to build a *structured* difference the same importer is used that will hide the same type of errors!

Auto Refresh Import

The Auto Refresh Import feature is only available together with the Delphi 4 and higher. This function will automatically refresh a unit in ModelMaker if you save the unit in the Delphi IDE. This improves synchronization of code developed both in ModelMaker and the Delphi IDE at the same time. However there are some serious warnings.

How it works

If you change a unit in the IDE editor that is also maintained in a ModelMaker model, the model and source file will be out of sync. Normally you have to 'refresh import' the unit to synchronize the model with the changes on disk. The auto-refresh import feature will do this automatically each time you save a unit in the IDE.

How it is activated and controlled

In the ModelMaker menu in the Delphi 4 (and higher) IDE check the item 'Enable Auto Refresh'. If this option is set, each time you save a unit (or project) in the IDE, the 'Auto Refresh command is send to ModelMaker which checks if:

1. The Environment option 'Auto refresh Import' is checked
2. The unit is in the current model
3. Unit generation is not (user) locked
4. The unit has 'auto code generation' enabled

If all above conditions are met, ModelMaker will do a refresh import and unlike after a manual 'refresh import' leave the unit in 'auto generation enabled' mode and - this is ***very important*** - *regenerate* the unit.

Effectively Auto Refresh improves synchronization between ModelMaker and the Delphi IDE: whenever you change something in ModelMaker, the auto-generation enabled unit will regenerate the file and reload it in the IDE. Whenever you change and save a file in the IDE, ModelMaker will resynchronize it in the model.

Warnings

In the normal, non-auto refresh development model you always have one master and slave: either ModelMaker refreshes the IDE using automatic code generation or you manually refresh the ModelMaker model with the IDE if you want to resynchronize again. With this feature there's no master or slave anymore. This can seriously damage your work as may be clear from the following example: When refreshing the unit, ModelMaker assumes it's reading a 'compilable unit'. If you for example have omitted a single begin or 'end;' or worse, comment out something, have unterminated strings etc. class and method import will be in trouble and not detect your error but simply remove all 'unwanted' methods. Since ModelMaker detected a change the unit is auto-generated and immediately after you saved the unit is reloaded with disastrous results. Experience shows that it's easy to loose lots of work instantly. Auto Refresh must be used with great care. Note that if Auto Save is enabled in the IDE, the Compile / Run command will auto save modified units depending on your IDE environment settings.

If you want more control, rather than just save in the IDE invokes auto refresh, you can use the ‘Refresh Import’ command from the ModelMaker IDE expert to save a file. This command will not only generate a manual refresh import command but also automatically save your unit in the IDE. Therefore you could use this command with shortcut Ctrl+Shift+H rather than the conventional Ctrl+S to save and refresh. You can add the ModelMaker Refresh command to the IDE tool bar. Check chapter (Integration in) Delphi IDE page 106.

Editing Form source files

ModelMaker Tools developed the ModelMaker Code Explorer to help editing Form, Data Module and other resource module source files. Due to the nature of these resource module files the IDE editor is more suitable for editing them. The ModelMaker Code Explorer will dock into the IDE editor and bring basic Code Model editing and navigation actions right into the IDE.

If you do not have the ModelMaker Code Explorer installed, you *can* edit form (and other resource module) source files with ModelMaker. This offers many advantages:
1. Use ModelMaker’s high level view and filters to navigate through your form code.
2. Automatically *restructure* your source files by regenerating them.
3. Improve your form code quality by adding methods and (array) properties with the same ease as for “normal” non form classes. Especially when you turn your forms into components, you want them to have nice and clean code to improve maintainability.
4. In general speed up form implementation.

For a smooth cooperation between Delphi and ModelMaker, stick to these rules:

In Delphi,
- Create and rename the form and unit.
- Add, delete and rename components.
- Set component properties.
- Create, rename and delete event handler methods.

Delphi adds all it’s components and event handler methods with the default visibility, so they are easy recognized in ModelMaker (use the members filter to filter out default visibility).

In ModelMaker,
1. Import your form file in an (empty) ModelMaker model.
2. Add, edit and delete all other members
3. Add additional classes to the unit.

To synchronize between Delphi and ModelMaker,
1. Regenerate the ModelMaker unit whenever you changed your code in ModelMaker. The auto-generate feature will help you doing this automatically.
2. Refresh the ModelMaker unit whenever you changed your code in Delphi, the integration experts will help you doing this automatically.

In source documentation

Overview

ModelMaker supports generating and importing "in-source” documentation. That is: the documentation and One Liner attributes of Code Model entities can be inserted during code generation and / or read (back) during import. Generation and importing of in-source documentation is controlled by the Project options tab “Source Doc Generation” and “Source Doc Import”. The Import source dialog allows temporarily overruling the import settings.

"In-source" documentation must be marked with special (user definable) documentation and one liner signature tags, more or less like in Java Doc. These signatures must be an Object Pascal comment symbol followed by one or more letters, Common used signatures are:

Documentation tags:
```
{{
{:
(**
(*:
```

One liner tags:
```
{1
//:
//1
```

Generation is internally based on macros. To customize the generated format, you can redefine these macros. To ensure a correct round trip (generation followed by a re-import) the generation and import settings must match. Normally ModelMaker enforces a correct match by using the essential import settings for generation. If you redefine the documentation generation macros, you must ensure correct matching.

Generating in-source documentation

ModelMaker supports generating and importing ‘in-source’ documentation in source files. Generation is controlled by the Project options on tab ‘Source Doc Generation’ and some macros (refer to Macros). You do not need to define these macros, as ModelMaker will on the fly insert the required macros. You can however redefine them either as environment or as project macro to customize the generated format.

Documentation macros

Macro name	Description	Example
ModuleDeclDoc	Used to expand Module (unit) documentation. Check Module related macros	{{ module <!ModuleDoc!> }
EventDoc	Used to expand Event Type documentation	{{ event type <!MemberDoc!> }
ClassIntDoc	Used to expand Class documentation in the class declaration	{{ class <!ClassName!> <!ClassDoc!> }
ClassImpDoc	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	{{ class <!ClassName!> <!ClassDoc!> }
MemberIntDoc	Used to expand member documentation in the class interface	{{ <!ClassName!>.<!MemberName!> <!MemberDoc!> }
MemberImpDoc	Used to expand method implementation documentation	{{ <!ClassName!>.<!MemberName!> <!MemberDoc!> }
MemberEndMacro	used to expand method termination documentation	{ <!ClassName!>.<!MemberName!>
OneLinerMacro	Used to expand one liners in classes, members, units and event types	//: Summary: <!OneLiner!>

In the examples it is assumed that {{ is the documentation signature and //: is the One Liner signature.

As you see in the examples, you can use any of the predefined macros such as <!ClassName!> inside the documentation macros.

However, ModelMaker is only able to import certain styles of source documentation. This is important if you want to ‘Refresh Import’ a unit.

Importing in-source documentation

ModelMaker supports importing "in-source" documentation. Importing source documentation is controlled by settings in the Project options tab “Source Doc Import” and the Import source dialog. As explained: "In-source" documentation and One Liners must be marked with special (user definable) documentation signature tags. More or less like in Java Doc.

An example:

```
//1 SomeMethod does something useful.
{{
procedure TSample.SomeMethod
This method does something useful
It would take pages to tell what.
}
procedure TSample.SomeMethod;
begin
end;
```

In the above example the OneLiner signature is defined as //1 and the documentation signature is defined as {{. The whole comment until the matching comment end symbol is treated as documentation for the first entity defined following or preceding it - depending on the documentation import options. In the example above the comment will be assigned to TSample.SomeMethod. You cannot use multiple line // style comments for documentation.

In the Source documentation options there are three documentation import modes:
1. Import: enables source documentation import and replaces documentation in the model with that in the source file,
2. Clean up: leaves the documentation in the model unaffected, but removes documentation from the unit-code
3. Inactive: which does nothing and leaves the documentation in the unit code.

The option ‘Remove unmatched documentation determines if only matched documentation should be removed or just any comment with the documentation signatures.

When assigning the comment to the Documentation attribute, the first and last #n lines are ignored. Defining any other value than 1 (one) is only useful if you redefine the documentation generation macros. The standard macros assume a value of 1.

You could use the fact that the first n lines (user definable) are ignored and place there the additional macros you'd like to generate. For example:

```
{{
<!ClassName!>.<!MemberName!>
(<!Visibility!>)
<!MemberDoc!>
}
```

This macro would require the first three (3) lines and the last one (1) to be skipped. Using this technique the member documentation won't grow each time you (refresh) import a unit. Another option is to use the Import "clean-up" import mode after in-source documentation has been imported once.

Alternatively you could leave the number of lines to be skipped at 1, and use additional removal style comments to customize your in-source documentation format.

For example:

```
macro ClassIntDoc =
{- **********
<!ClassName!>
```

**********}
{{
<!ClassDoc!>
}

Note the removal style comment in the first part of this macro. Check chapter “Comments with Remove signature”, page 60.

Related to this is the use of a custom ClassSeparator or MethodSeparator to emit more than just the documentation preceeding the class or a method implementation. Refer to Code Generation options. Using custom separators has the advantage of not needing the redefine the documentation macros and that way ensuring that all expanded documentation looks similar.

For example, defining a MethodSeparator macro like this:

```
{- <!ClassName!>.<!MemberName!>
  (<!Visibility!>) }
```

Combined with enabling the method implementation in source documentation (without redefining the related macro), this behaves as if the entire macro were:

```
{- <!ClassName!>.<!MemberName!>
  (<!Visibility!>) }
{{
<!MemberDoc!>
}
```

Which would be emitted for example like this:

```
{- TSample.SomeMethod
  (public) }
{{
This is the documentation for the method
}
```

Note the use of removal style comments for the MethodSeparator that avoids creep when re-importing the generated code.

Code templates

ModelMaker supports the use of code templates. They are like user definable patterns and consist of a (usually consistent) set of members that is put in a code template source file. This template can then be applied whenever needed again. The template file acts like a structured persistent copy / paste buffer. The powerful aspect is that templates can be parameterized using user definable macros. ModelMaker will extract the macros, let you edit them and expand the macros before importing the template file. The template files can be edited in Delphi, but should not be imported in ModelMaker directly.

Creating a Code template

To create a code template, in the Members view select the members you want the template to contain use the popup menu 'Create code template'. You'll be prompted for a file name. ModelMaker then generates the selected members as part of a stand-in class named `TCodeTemplate`. Here's an example of a simple template file containing a simple `Items` property with a standard `TList` implementation

```
unit SimpleItems;

  TCodeTemplate = class (TObject)
  private
    FItems: TList;
  protected
    function GetItems(Index: Integer): TObject;
  public
    property Items[Index: Integer]: TObject read GetItems;
  end;

function TCodeTemplate.GetItems(Index: Integer): TObject;
begin
  Result := TObject(FItems[Index]);
end;
```

Applying a Code template

To apply a previously created template, select the popup menu 'Apply template' from the Members view. You'll be prompted for a template file name. ModelMaker will import the members contained by the first class in the template unit and add them to the currently selected class. Other classes and all other code in the unit are/is ignored. There's one exception: event type definitions can also be added to a code template, they will automatically be added to the event library when applying the template.

Registering a Code template

Code Templates can be registered on the patterns palette (patterns view) and Code Template palette (members view popup menu). These palettes that look like the Delphi component palette make it even easier to apply a Code Template. To (un)register a template use the popup menu functions in the palettes or select the corresponding option when you create a Code Template. Code Templates are shared with the ModelMaker Code Explorer.

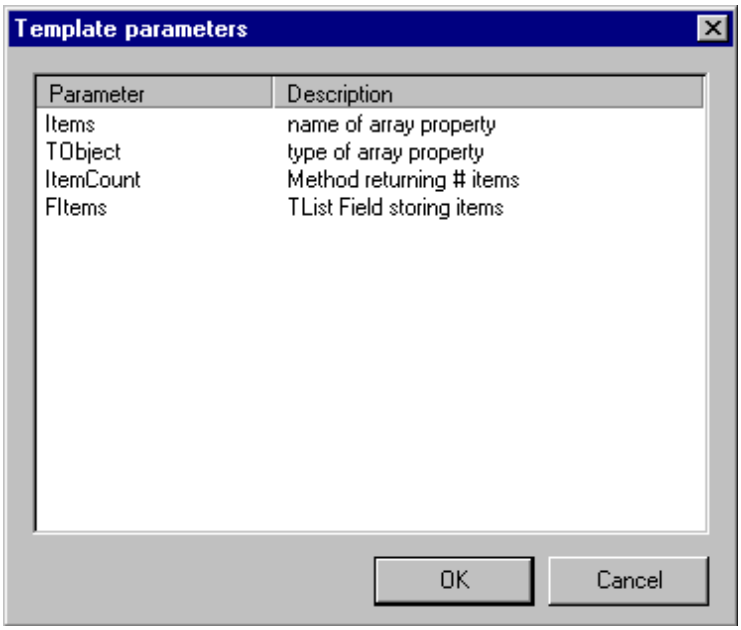
Parameterize a Code template using macros

You can parameterize a code template by adding macros to the template unit. When applying a template, ModelMaker will first extract the macro parameters, let you edit them, expand the code template and finally apply the template. This allows you to create more generic templates. A macro definition should be formatted as

```
//DEFINEMACRO:macroname=macro description
```

The standard ModelMaker macro rules apply. For example: macroname must be an identifier. Parameterizing the above example could for example be done like this.

```
unit SimpleItems;
//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

  TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
  end;

function TCodeTemplate.Get<!Items!>(Index: Integer): <!TObject!>;
begin
  Result := <!TObject!>(<!FItems!>[Index]);
end;
```



If you apply this template, ModelMaker will show a dialog with the list of parameters you defined: Items, TObject, ItemCount and FItems allowing you to change them for example in Members, TMember, MemberCount and FMembers. This way the template can be added multiple times in different contexts. ItemCount is not used in this example, but in the sample code template that is shipped with ModelMaker method ItemCount is part of the template.

ModelMaker predefines one macro: “ClassName” contains the name of the class the macro is applied to. You can redefine ClassName when parameterizing a template. Use ClassName for example to create a singleton implementation macro:

```
function TCodeTemplate.Instance: <!ClassName!>;
begin
  // return the single instance.
end;
```

When applied to a class named TMySample this will expand to:

```
function TMySample.Instance: TMySample;
begin
  // return the single instance.
end;
```

Macros

Overview

A macro in ModelMaker is an identifier placed between <! and !> tags. They may also include an optional parameter list. When the macro is expanded, the macro identifier and tags are substituted by the text associated to the macro. For example macro <!UnitName!> will (in unit Samples) be expanded to the actual unit's name 'Samples'.

Macros are used in
- Code Generation.
- Customizing certain aspects of code generation such as a custom class separator, inserting categories etc.
- Generating in-source documentation.
- Parameter zing Code Templates
- ModelMaker Code editor.

During code generation ModelMaker predefines some model statistics macros at run time. Such as <!`UnitName!>, <!Date!>` etc.

You define your own macros in the Macros view. Macros are defined per desktop (environment) and per project. If a macro is both in the project and the environment macros, the project macro overrules (redefines) the environment macro.

For Parameterizing Code Templates using Macros, refer to chapter Parameterize a template using macros , page 72. To expand a Code Template the predefined macros and project and environment macros are not used.

Macros in Code generation

When generating a source file from a ModelMaker unit (refer to chapter Code Generation page 45), ModelMaker will expand macros in all text that is send to the output file. Therefore macros can be placed in any code: in unit code, in a section of a method's implementation or even in a local var definition. Macro expansion is switched on and of with the generation control tags `MMWIN:START EXPAND` and `MMWIN:END EXPAND` By default the expansion is switched ON. Check chapter "Macro expansion control tags", page 47 for an example.

When expanding a macro, first the list with predefined macros is checked, then the Project macros and finally the environment macros. If an identifier is not found, the macro text is

either just removed or generation is aborted depending on the setting of '`Ignore undefined macros`' in the Project options Code Generation tab.

Predefined macros

This table shows the macros ModelMaker predefines when generating a source code file. Some macros may be redefined - especially the documentation expanders, others such as Date and ClassName are fixed.

Macro Name	Allows override	Description	Example
		Generic predefined macros	
`Date`	No	Generation date, for example: `19-02-2003`	
`Time`	No	Generation time, for example: `12:34:56`	
`LineNr`		The 1-based line number in the resulting source file at which the macro is defined.	
		Documentation expanders	

`ModuleDeclDoc`	Yes	Used to expand Module (unit) documentation. Check Module related macros	`{{`
			`<!ModuleDoc!>`
			`}`

`EventDoc`	Yes	Used to expand Event Type documentation	`{{`
			`<!MemberDoc!>`
			`}`

`ClassIntDoc`	Yes	Used to expand Class documentation in the class declaration	`{{`
			`<!ClassDoc!>`
			`}`

`ClassImpDoc`	Yes	Used to expand Class documentation in the class implementation (emitted just before the first method implementation)	`{{`
			`<!ClassDoc!>`
			`}`

`MemberIntDoc`	Yes	Used to expand member documentation in the class interface	`{{`
			`<!MemberDoc!>`
			`}`

`MemberImpDoc`	Yes	Used to expand method implementation documentation	`{{`
			`<!MemberDoc!>`
			`}`

		Customization Macros	

`ClassSeparator`	Yes	Custom Class separator, Only active if corresponding option is activated in Project Code	`{-******************`
			`** Class: <!ClassName!>`
			`** Category: <!Category!>`
			`**********************}`

		Generation options.	assuming {- is the comment remove style
`MethodSeparator`	Yes	Custom Method implementation separator. Only active if corresponding option is activated in Project Code Generation options.	`{- <!MemberName!> -}` assuming {- is the comment remove style
`SectionSeparator`	Yes	Custom Method section separator. Only active if corresponding option is activated in Project Code Generation options.	`{------- section --------}` assuming {- is the comment remove style
`IntfCategory` `ImplCategory`	Yes	Wrappers for Category emission in class or member interface and implementation.	`{- Category: <!Category!>}`
`OneLinerMacro`	Yes	Used to expand one liners (class, member, unit event type)	`//: Summary: <!OneLiner!>}`
`MethodEndMacro`	Yes	Used to expand method end documentation. Only active if corresponding option is set in Project Source documentation Generation options	`{<!ClassName!>.<!MethodName!>}`
		Entity specific Macros	
`OneLiner`	No	All entities: for Modules only during Module documentation generation.	
`OneLiner` `Category`	No	All entities: for Modules only during Module documentation generation.	

`ModuleDoc`
`ModuleName`
`ModulePath`
`Alias`
`UnitName`
`UnitPath`
	No	Module specific macros (units). Always available (not only in module documentation). The UnitXYZ macros exist for backward compatibility. Use the ModuleXYZ macros in new projects	

`ClassDoc`
`ClassName`
`TrimmedClassName`
`Ancestor`
	No	Class specific macros. Valid in class and members of that class. Valid in unit code after the first class has been generated. TrimmedClassName contains the class name with the first character (‘T’/ ‘t’) removed.	In unit code use this to insert auto updated global variables or class pointer types:
			
			`var <!TrimmedClassName!>: <!ClassName!>;`
			`type <!ClassName!>Class = class of <!ClassName!>;`

`MemberDoc`
`MemberName`
`Visibility`
`DataType`
	No	Member specific macros. Valid during generation of declaration, documentation and method	

		implementation code.	

`Parameters`
`CallParams`
`MethodKind`
	No	Method specific macros. Valid during generation of declaration, documentation and method implementation code.	

		Event Types use the same macros as methods. Except: Category and Visibility which are undefined	

The Documentation Expander macros are used to generate in-source documentation. To customize the generated format, you must redefine these macros, either in the project or environment macros. Refer to chapter “Generating in source documentation”, page 67.

The Customization macros can be (re)defined to customize the format of the related aspect. ClassSeparator, MethodSeparator and SectionSeparator are only active if the corresponding options are checked in the project Code Generation options. Refer to chapter “Code Generation Options”, page 49.

Using Macros in code

Some rules that apply to using macros:

- An entire macro including start and end tags must reside on a single line.
- Macro identifiers can contain characters ['0'..'9', 'a'..'z', 'A'..'Z', '_']. This is similar to Object Pascal identifiers, although macro names can start with a number.
- White space surrounding the macro identifier is ignored.
- A start tag not followed by a valid identifier is not considered to be a macro
- If the macro identifier is not followed by either the end tag or the parameter list, the macro is not considered to be a macro.

Rather than presenting the macro syntax diagram, an example will demonstrate the use and definition of macros.

Assume these (user) macros to be defined, in either the project macro list or in the environment macro list.

Name: `Author`
Parameters:
Text:
```
S.M.A.R.T. Programmer
```

Name: `Assert`
Parameters: `Cond, Msg`
Text:
```
{$IFOPT D+}
if not (<!Cond!>) then
  raise Exception.Create('Assertion error in line <!LineNr!> of unit <!UnitName!>' +
```

```
                          #13 + Msg);
{$ENDIF}
```

This is how you could use these macros:

```
procedure SomeAction(Index: Integer; C: Char);
begin
  <!Assert(Index >= 0, 'Index out of range')!>
  <!Assert(C in ['a', 'b'], 'Char out of range')!>
  <!Assert(ValidPair(Index, C), 'Additional checks failed')!>
  ShowMessage('<!Author!> created this code');
end;
```

Assuming the procedure was placed on line 100 in unit Demos, this text would expand to:

```
procedure SomeAction(Index: Integer; C: Char);
begin
  {$IFOPT D+}
  if not (Index >= 0) then
    raise Exception.Create('Assertion error in line 102 of unit Demos' +
    #13 + 'Index out of range');
  {$ENDIF}
  {$IFOPT D+}
  if not (C in ['a', 'b']) then
    raise Exception.Create('Assertion error in line 107 of unit Demos' +
    #13 + 'Char out of range');
  {$ENDIF}
  {$IFOPT D+}
  if not (ValidPair(Index, C)) then
    raise Exception.Create('Assertion error in line 112 of unit Demos' +
    #13 + 'Additional checks failed');
  {$ENDIF}
  ShowMessage('S.M.A.R.T. Programmer created this code');
end;
```

The basic rules that apply to the use of macros are:

- When using macros in text, the complete macro including its parameter list must reside on a single line. So this won't work:
  ```
  <!Assert((A > B) and (B > C),
           'This is bad input')!>
  ```
- Arguments in the parameter list are comma delimited, such as in the use of `Assert`.
- Arguments can contain even pairs of () , [] and '' characters, such as in sets, arrays and string literals.
- If no parameters are defined, as in <!Author!>, you omit the brackets when using the macro.
- Macros can use other macros in their macro text. In fact even parameters can be macros. Nesting is allowed up to 15 levels.
- Circular macro definitions are illegal.
- Macros expand to plain text. See for example the use of the predefined LineNr and UnitName macros in the Assert macro's text. The expanded macro LineNr is not an Integer, and the expanded macro UnitName is not a string.

Using macros in the code editors

An entirely different use of macros is to expand a macro in the code editor. To do this, press Ctrl+Space after typing the macro name - or Shift+Space, depending on your Environment Options|Editors settings. This is convenient if you create macros like: **tryf** that could expand to:

```
try
  >< // >< in macro text positions cursor after expansion.
finally
  .Free;
end;
```

Using macros in your default unit template

To demonstrate the use macros, here's the effect of using macros on the default unit which was described in "Adjusting the unit template" page 50.

```
{
<!UnitHeader!>

<!FreeWareDisclaimer!>
}

unit <!UnitName!>;

interface

// rest of the unit template is unchanged
end.
```

Notice how this template differs from the previous template:

- The company header is now placed in a macro `UnitHeader`. This saves a lot of redundant text.
- The macro `UnitHeader` contains other macros like unit name, model name, etc.
- The disclaimer is now placed in the macro `FreeWareDisclaimer`.

Diagrams

Diagrams, Diagram List view

ModelMaker supports a set of diagram types as defined by the UML. Currently the UML v1.3 style is implemented. A model can contain any number of diagrams of any type. The Diagram list view (Main menu View or F5) contains a list of all diagrams in the model. Diagrams contain symbols and associations that can be, but do not need to be linked to entities in the code model.

The Diagrams list View is used to create, rename and delete diagrams. Select a diagram in the Diagrams list view to open it in the diagram editor.

Diagrams are named. Names do not need to be unique.

Hierarchy

Diagrams can be organized hierarchically in the Diagrams view. This organization is pure visual and has no further meaning in the model. Any diagram can serve as a parent for any other diagram. To rearrange parent child relations, make sure the list's "Order by" is set to Hierarchy (Diagram list Popup menu). Then use drag and drop to rearrange diagrams. If the Ctrl key is pressed while dropping, the hierarchy is edited; else the order within the current parent is changed.

Styles

Each diagram has a visual style and a symbol style that define the default styles for the symbol inside that diagram. Editing these styles (diagram editor popup menu Diagram attributes), affects the appearance of all symbols in that diagram. Refer to chapter "Visual styles" for more information.

Default properties

Whenever a new diagram is created (except a Cloned diagram), the format (size) and orientation (portrait or landscape) are set to the defaults as defined in the project options.

Symbols and visual containment

Symbols can visually contain other symbols. For example package symbols can contain class symbols or other package symbols; and Nodes in a deployment diagram can contain Component symbols. Visual containment implicates visual ownership only. Visual containment is not linked to the code model. Inserting a class symbol into a package that is linked to a unit will not actually move that class to that unit. It is just visualized as being part of that unit.

After a symbol is created, visual containment is fixed and cannot be changed in the diagram editor. The only way to change a symbol's parent is to cut the symbol and paste it on the new parent. Take care: connected associations are only copied if both ends of the association are copied.

Symbol names and other visual adornments

Most symbols have a symbol name text adornment. Depending on the visual style options, the name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of a symbol is also displayed in the name adornment.

Other visual adornments are depending on the type of symbol. For example: a state region for a concurrent state contains a state region divider and icons, a sequence diagram role symbol contains a lifeline etc.

Associations

Basics

Associations are used to model relations between symbols. Some associations such as “documentation link” and “constraint” can also be connected to other associations. An association that is not connected to both ends is invalid and will automatically be removed from the diagram.

All associations have a direction: they lead from a source symbol to a target symbol. Usually a navigation arrow is displayed is appropriate. These arrows can be suppressed in the visual style on project, diagram or association level.

Usually the visual path of an association is formed by the two association end points. Shape nodes can be inserted to create more complex paths.

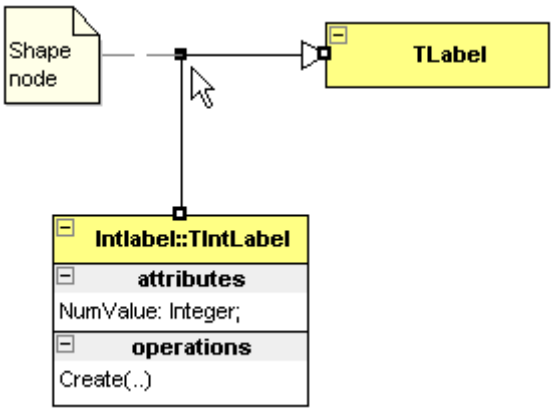
Associations are created by clicking the mouse on the source symbol and while keeping the mouse down, drag to the target symbol where the mouse is released. Once created, associations can be connected to different symbols by moving one of the endpoints to the new source or target symbol.

Anchors

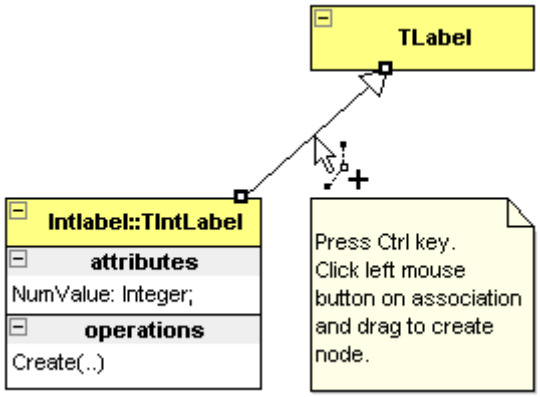
Associations are connected with an anchor point to symbols. Usually the anchor is connected to the center of a symbol. You can however drag-move the connection anchor point within the bounds of the connected symbol. This will change the intersection point of symbol and association. The diagram editor’s popup menu has a function ‘Reset Anchors” which will reset both anchors to the symbol’s center.

Shape Nodes

Shape nodes can be inserted to change the visual path of an association. Shape nodes allow bending associations visually. The association anchor points (connection points on the connected symbols) and the association’s shape nodes make up the actual visual path of the association. The picture below shows an generalization association from TIntLabel to TLabel with one shape node.



To insert a shape node into an association, press the Ctrl-key and drag on a line segment of an association. After the mouse is released, a new node is inserted. Alternatively, use Insert Shape Node from the diagram editor “Association” popup sub menu. This command is available if you invoke the popup menu on an association.

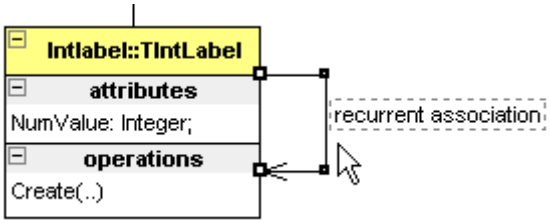


Shape nodes can be moved by selecting the association and then drag them with the mouse.

A shape node can be deleted by aligning it with the two surrounding nodes and is basically the reverse of creating a new shape node. Alternatively, use Delete Shape Node from the diagram editor “Association” popup sub menu. This command is available if you invoke the popup menu on a shape node. The command ‘Clear shape nodes” in the same popup menu will clear all shape nodes in all selected associations.

Recurrent associations

Normally associations connect two different symbols. If both ends of an association are connected to the same symbol, the association is recurrent. Most associations can be made recurrent. Some recurrent associations will display a rounded curve rather than a square path. The rounded curve is a bezier that is controlled by two shape nodes. Inserting another shape node will turn the bezier curve into normal straight lines.



To create a new recurrent association, simply make the target symbol the same as the source symbol. Two shape nodes will automatically be inserted that allow control of the visual path.

An existing non-recurrent association can be made recurrent by moving on of the endpoints to the same symbol as the other endpoint. Again, two shape nodes will be inserted automatically.

To convert a recurrent association into a non-recurrent association, move either source or target endpoint to another symbol. The shape nodes will be removed.

Association Name, Qualifiers, Roles and other adornments

Most associations can be named and have an association name adornment. If the name is currently not visible, select the association and press F2 to invoke the in place editor. Depending on the visual style options, the association name will display a hotlink status icon, a navigation icon and the hyperlink status. Check chapters Hyperlinks and Hotlinks. The stereotype (category) of the association is also displayed in the name text adornment.

Other adornments include

1. Qualifiers in qualified association such as class associations. To reduce visual space, the qualifier's type can automatically be suppressed. This is controlled with the visual style options.
2. Role name, both source and target endpoint can be named in for example class associations and object links.
3. Multiplicity (cardinality), both source and target endpoint can have a multiplicity in for example class associations and actor communications.
4. Conditions, usually only in the source endpoint of sequence diagram messages and state transitions.

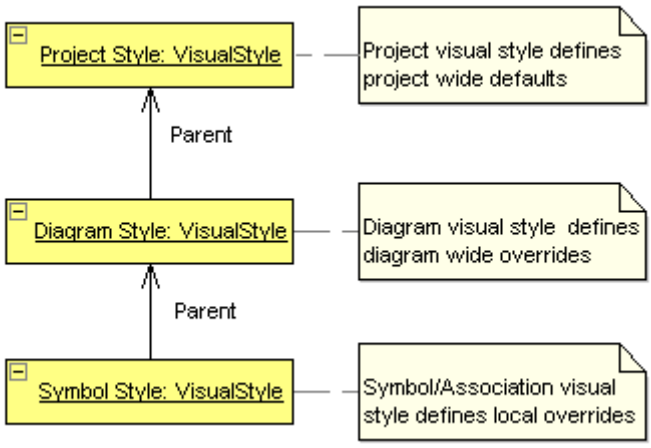
These adornments are all texts and can be moved freely in the diagram. The word break property of text adornments is controlled with the Word break functions on the "Align and Size" palette.

Visual styles

All symbols, associations and diagrams have a visual style. It is this visual style that defines how a symbol appears. A visual style contains font settings, a color palette and some options that control display of icons. The symbol styles, that are discussed in the next chapter, control *what* is displayed; the visual styles control *how* a symbol is displayed.

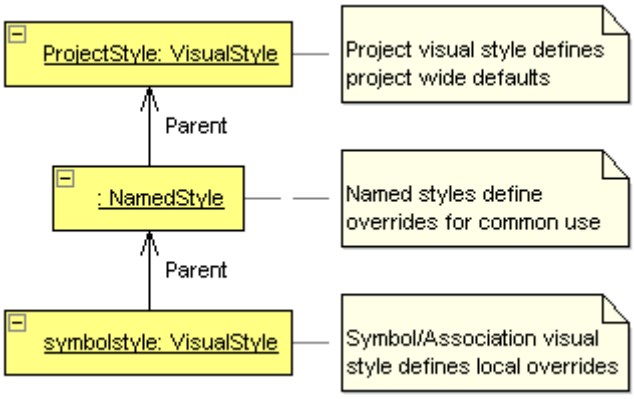
Style hierarchy

All visual styles are linked in a lookup hierarchy. A style normally looks up an attribute in the parent style but can also override “parent” attributes. By default symbol and association styles are linked to the diagram visual style and have all properties set to “lookup from parent”; that is override or change nothing. As a result, all symbols will appear as defined by the diagram visual style. On their turn, all diagram visual styles are linked to the project visual style and also have all properties set to “lookup from parent”.



This hierarchical structure allows easy adjusting of visual appearance on any level. To change the appearance of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify the appearance of all symbols a single diagram. To change the appearance of a single symbol, edit the symbol’s style.

To make reuse of visual styles easy, the visual style manager allows creation of “Named styles”. Named styles can be used to define a specific appearance that can be reused. A named style is applied making it the parent of a diagram or symbol style. Named styles can have other named styles or the project visual style as their parent. The diagram editor tool bar contains a parent style combo which is used for this.



Visual style properties

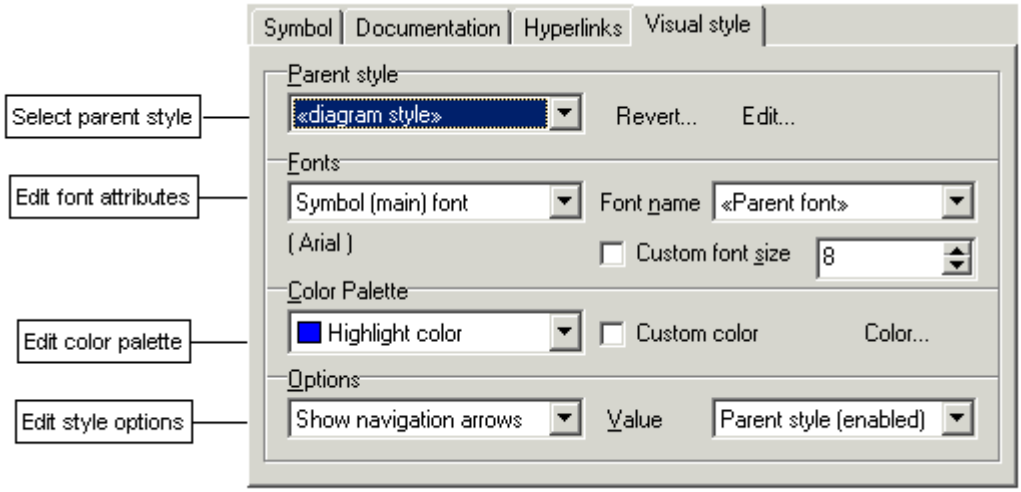
A visual style consists of a set of properties. Not all symbols / associations use all properties.

1. A main font name and size. The main font is used for symbol and association names. The font's style, bold, italic and underline, cannot be defined because that is usually has a syntactical meaning in the symbol as defined in the UML specifications: classifier names are bold, instances named are underlined etc.
2. An adornment or text font. Used for all other texts. For example the members in a class symbols and the roles in an association.
3. A color palette defining the colors for basic drawing entities such as main font, symbol compartment, symbol pen, symbol tab, association line etc. Which entries on this palette are used is dependent on the type of symbol.
4. Options that control display of some visual elements, such as: navigation arrows in associations, hotlink icons, navigation (hyperlink) icons etc.

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing a style. Refer to next paragraph.

Controlling & assigning styles

Most symbols properties dialogs and the diagram properties dialog contain a 'visual style' tab that allows editing the symbol's visual style. In this tab you change and or edit the parent style and the style's attributes.



To entirely revert to the parent style, erasing all overridden / redefined attributes, click the 'Revert' button.

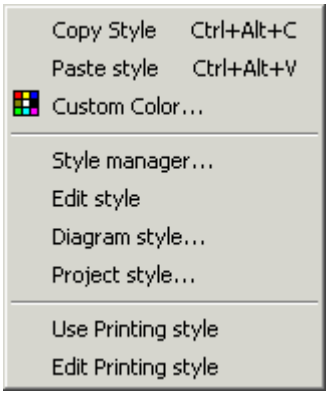
The diagram editor's tool bar contains some visual style specific tools.

The "parent style" combo displays the parent style for the selected symbols and associations. It is blank if selected symbols have different parents. It allows assigning a new parent for the selected symbols.

The “Revert to parent style” tool will reset the visual styles for all selected symbols and associations. Useful to erase any local redefined style attributes.

The Apply color tool will let you select a color and apply it to the selected symbols and associations. The color is applied as the “main” feature color. Usually this is the symbol color palette entry, but for tabbed symbols like class symbols and package symbols, the symbol tab color is changed.

The Diagram editor contains a ‘Visual style’ sub popup menu that contains some additional visual style related functions.



Most striking functions in this popup menu are:

1. Copy / Paste a visual style. This is useful in case you have redefined the style of a symbol and want to apply the same visual style to a selection of other symbols. Note that similar functions exist for the symbol style that controls the display of members etc.
2. Show the Style Manager and it’s named styles, which is described in the next paragraph.
3. Toggle (and Edit) “Printing Style”. The printing style is described in the next paragraphs.

Style Manager

The visual Style Manager is used to create and maintain named visual styles. Named styles can be assigned as parent style for symbols and diagrams. They allow creating a predefined set of appearances that can be applied by simply assigning the style.

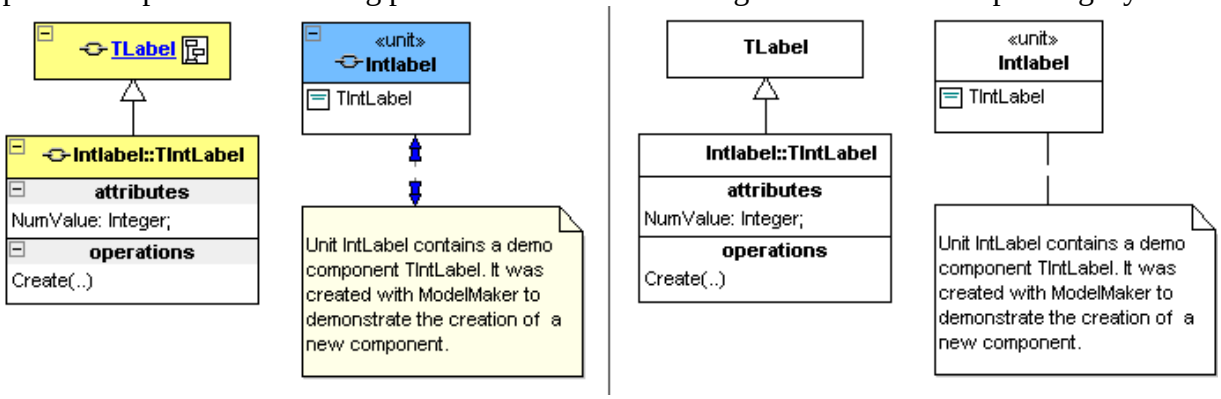
For example, you could create a style named “System components” which specifies a specific blue palette to paint symbols. Another style could be named “GUI components” and define a yellow palette to paint symbols.

In class diagrams, you can then easily change the visual appearance of a class symbol by assigning “System components” as parent style for the class symbol using the diagram editor’s tool bar parent combo.

Named Visual styles can be imported or exported with the Style Manager. This allows synchronizing named styles in projects.

Printing Style

ModelMaker allows suppressing a specific set of graphical features when printing diagrams. These include: printing in black and white (suppressing colors), no navigation icons, no hotlink icons etc. While these features can help while designing, they may be unwanted in printed output. The following picture shows the same diagram in normal and printing style.



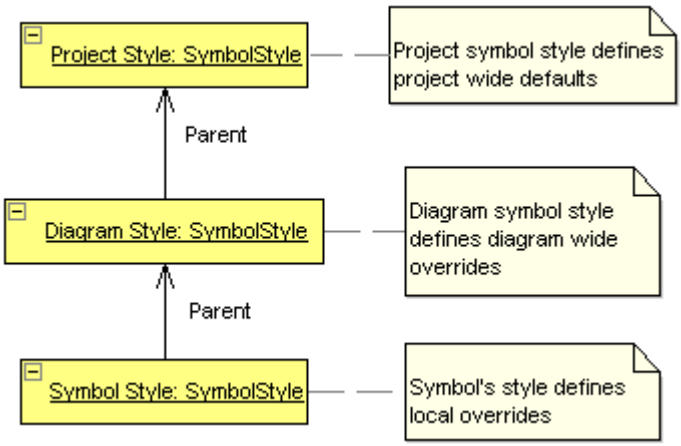
The printing style is defined in the diagram environment options. It is automatically superimposed on all other styles when printing, and can also be manually activated in the diagram editor. The diagram editor popup menu “visual style” menu contains an item that toggles the “Use Printing style” state. If the printing style is active in the diagram editor, it will also be effective when creating visual exports as image file or to the clipboard.

Symbol (contents) style

Just like visual styles hierarchically control the visual appearance of diagrams and symbols, symbol styles control the contents of displayed symbols. For example: which members are displayed in a class symbol and in which format is controlled by the class symbol “symbol style”. The symbol styles control *what* is displayed; the visual styles control *how* it is displayed. The symbol styles are only applicable for a specific set of symbols such as class, interface and (unit) package symbols. In classes the symbol style controls which members are displayed and how they are displayed. In unit packages they control whether contained classes are automatically displayed.

Style hierarchy

Just like visual styles, all symbol styles are linked in a lookup hierarchy and can override parent style attributes. Symbol styles are linked to the diagram symbol style and have all properties set to “lookup from parent” that is: override or change nothing. As a result, all symbols will appear as defined by the diagram symbol style. On their turn, all diagram symbol styles are linked to the project symbol style and also have all properties set to “lookup from parent”.



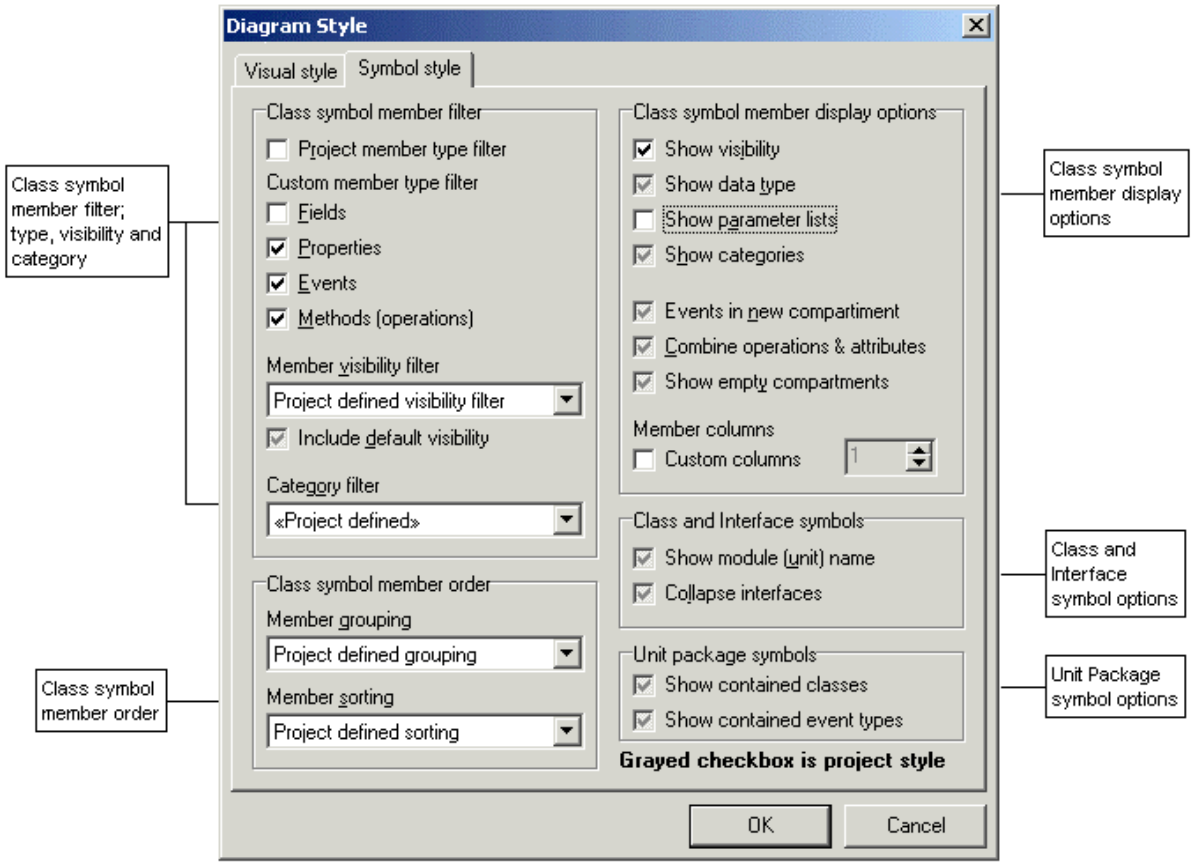
This hierarchical structure allows easy adjusting of what is displayed within symbols. To change the style of an entire project: change the project style. You can even save the project settings as default, and new projects will have the same project style. Change the diagram style to modify all symbols a single diagram. To change the contents of a single symbol, edit the symbol's style.

Unlike the visual style, the symbol style cannot be linked to named styles.

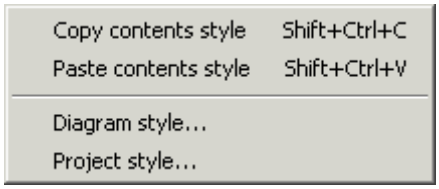
Controlling & assigning styles

The diagram properties dialog contains a 'symbol style' tab that allows editing the symbol style. The project options "symbol style" tab is similar. In these tabs you control how class and interface symbols display members and unit packages display contained

classes.



The Diagram editor contains a ‘Symbol style’ sub popup menu that contains some symbol style related functions.



Most striking functions in this popup menu are Copy / Paste a symbol style. This is useful in case you have redefined the style of a for example a class symbol and want to apply the same visual style to a selection of other class symbols.

Class & Interface symbols

The symbol style in class and interface symbols is edited on the “member style” tab of the symbol’s dialog. These are basically the same tabs as the diagram symbol style tab, except that fixed or non-appropriate elements have been removed.

Package symbols (units)

The symbol style in a package symbol is incorporated in the main symbol tab. Here you control if contained classes and events are displayed. This feature is only available for units and classes that are (imported) in the model.

Size and Alignment

The Drawing Grid

The diagram editor's drawing grid is defined in the project options diagram style tab. It helps aligning symbols. All symbols are automatically snapped to the drawing grid. And for most symbols, the extent (bounds) is automatically adjusted to fit on the grid too. The MindMap node symbol allows enabling/disabling this "Bounds on Grid".

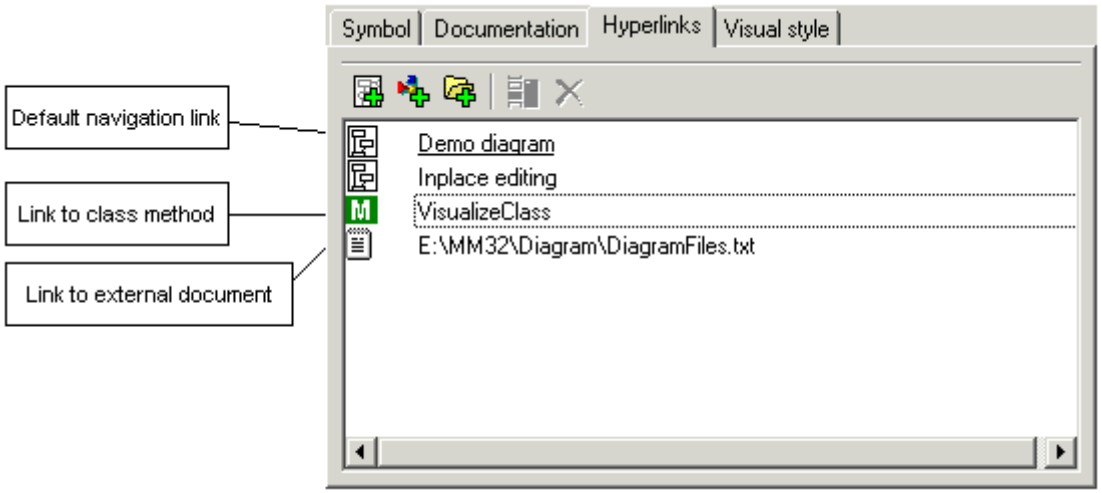
If you change the grid size, all symbols will most likely resize too. Because the grid it affects all diagrams and symbols, it is defined and saved per project.

Align & Size Palette

The Alignment palette, which is available from the diagram editor tool bar, contains a set of functions to control alignment, auto sizing, text alignment and word break properties. These functions are similar to those in other applications such as Delphi, and are not explained in detail.

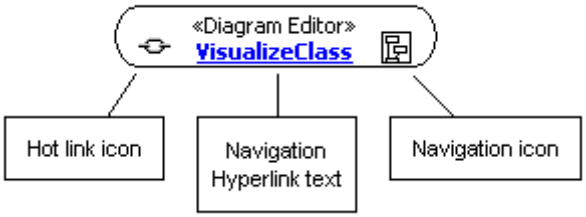
Hyperlinks, navigation

Virtually all symbols can contain hyperlinks. Hyperlinks can point to other diagrams, code model entities (class, member, event, unit) or to external documents. Hyperlinks are created and maintained in the "hyperlinks" tab of a symbol's dialog. Although there can be many hyperlinks, only one is the default navigation link. This is the first link that supports navigation. The default navigation link is underlined in the hyperlinks list. Only HotLinks to code model entities usually have navigation disabled. Check chapter HotLinks to Code Model.



Most symbols will show a navigation icon next to their name if a navigation hyperlink is available. This is shown in the picture below. These icons can be suppressed by the visual style or the printing style. Also, if a symbol has a default navigation link, the symbol's name

will appear underlined and in the hyperlink color as defined in the visual style's palette. The picture below shows this. The Hotlink icon is explained in Paragraph "Hotlinks to the code model".



If the navigation icon is clicked, ModelMaker will follow the link and navigate to the object. If this is another diagram, the current diagram is saved and the referenced diagram is opened in the diagram editor.

If the link refers to a code model entity, ModelMaker will select the entity (class, member) and make the classes and members views visible. If the entity is a method, the method editor will also be made visible.

If the link refers to an external document, ModelMaker will perform a default "open" command on the filename or URL.

Clicking the hotlink icon (left of the symbol name) will edit the hot linked entity rather than the symbol.

If a symbol contains more than one hyperlink, you can navigate to the non-default hyperlinks by using the diagram editor's popup menu Navigate function. This contains a dynamic submenu with all hyperlinks available in the focused symbol.

External documents

External documents are defined with the same alias / relative name mechanism as used for source files. Check chapter 'source aliases' in this manual for details. The use of aliases avoids the use of hard coded, machine dependent paths.

The standard shell "open" command is performed to navigate to an external document. This accepts all kinds of external documents such as executables or files that are associated with an application. For example "c:\temp\manual.doc" will be run MS word and open the document.

URLs to web pages or web sites are also valid. For example:
Alias=""
Relative filename = "http://www.modelmakertools.com"
will open a web browser and navigate to the ModelMaker Tools site. This can be used to navigate to html documentation etc.

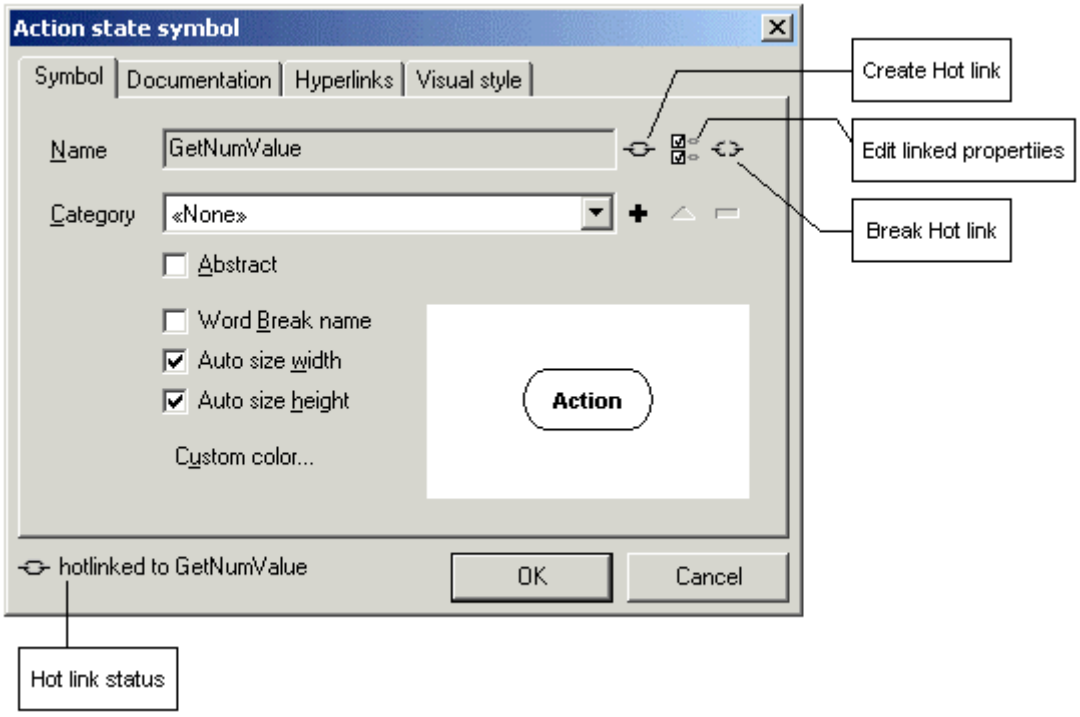
To refer to another ModelMaker model and start another ModelMaker instance, associate the ModelMaker project bundle extension *.mpb with the ModelMaker executable in the Windows shell.

Coupling Symbols to the Code Model

Most symbols and associations are linked or can be linked to entities in the code model. This is either hard coded or can be done manually by creating a HotLink. A hotlinked symbol will share name, documentation, one liner, “abstract” state and stereotype (category) with the linked entity. Modifying one of these properties in the symbol reflected to the linked entity and vice versa.

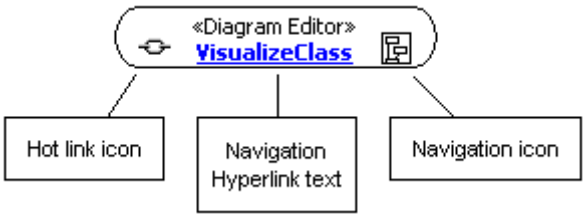
HotLinks

HotLinks are used to link a symbol or association to another entity, usually a code model entity like a class or a method. A hotlink is basically a navigation hyperlink that is (internally) marked as “hot”. Most symbol dialogs contain a set of buttons next to the name that allow creating and editing the hotlink. Here is an example from the Action State editor dialog.



Normally, action states are not coupled to any other entities. If you for example wanted to link an action state symbol to the TIntLabel.GetNumValue method, click the “Create HotLink” button and select the entity to link to. To break the hot link, click the Break hot link button. The icon at the bottom left of the dialog displays the hot link status. Not all properties need to be linked. To edit the linked properties, click “Edit linked properties”. The linked properties dialog lets you select which properties are linked.

If a symbol is hot linked, a hot link icon is displayed at the left of the symbol’s name. The following picture shows this. Hot link icons can be suppressed in the visual style and/or in the printing style.



If you click at the hot link icon in the symbol's name, the linked entity will be edited rather than the symbol.

If a symbol is hot linked to a code model entity, that entity will be selected if you click on the symbol or any of its (text) adornments. Unlike navigation through hyperlinks, this will not ensure that the associated view is made visible.

Delete HotLinked entity

The Diagram Editor toolbar contains two delete tools: one to delete the symbol from the diagram, one to delete the symbol and the hotlinked entity. The last one will not only delete the symbol from the diagram but also remove the linked entity. You will be asked for confirmation before the linked entity is deleted.

Specialized symbols and associations

Some symbols are linked to the code model by design. These symbols do not allow any other linking than the built in one.

Class and Interface symbols
Class and Interface symbols are implicitly linked to a class in the code model. If the class is deleted from the code model, all class symbols linked to that class are removed from all diagrams.

Property and Field associations
Similar to class symbols, property and field associations are hard linked to properties and fields in a class. The data type of these members must match the association target class.

Shared Class Association
Shared class associations are associations between class symbols that allow greater flexibility than field and property associations. They do not need to be linked to existing code model members or classes. Because they are shared, they can be auto visualized if both source and target classes are being visualized on a diagram.

Generalization relation

Generalization (inheritance) relations can be created between all symbols that are generalizable. In most cases they are not coupled the code model. Only if they connect two class or interface symbols, they are implicitly linked to the code model. Therefore, changing a generalization between two use cases does not affect the code model. But creating or changing a generalization between classes will be reflected in the code model by changing the class hierarchy.

Realization relation

Realization relations (such as interface support) can be created between most symbols that allow realization. In most cases they are not coupled the code model. Only if they connect a class and an interface symbol, they are implicitly linked to the code model. Therefore, changing a realization between a package and interface does not affect the code model. But creating or changing a realization between a class and interface will be reflected in the code model by adding or removing interface support to that class.

Package (units)

Units can be visualized as package symbols. Normal hot linking is used to achieve this. However, unit packages can display the contained classes and events. This is controlled by the diagram symbol style and the related options in the package editor dialog. The displayed content is read-only.

Documentation & OneLiners

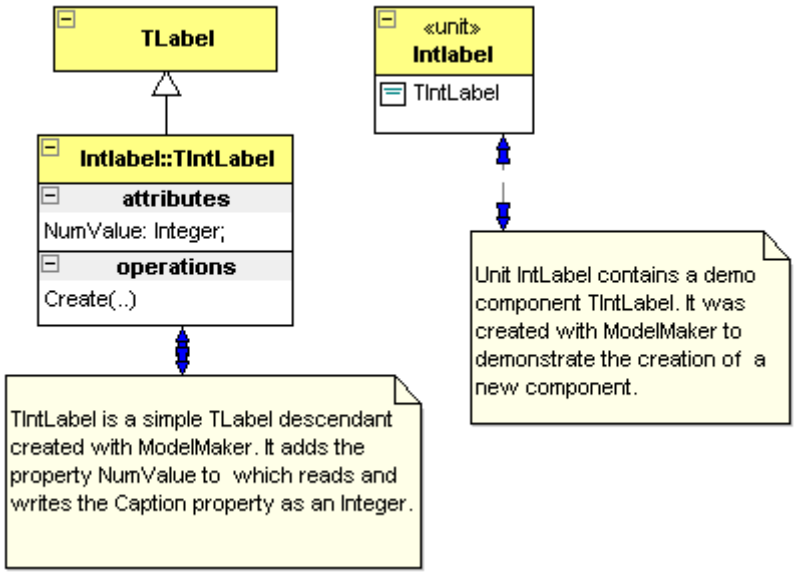
Floating Documentation view

All symbols can be documented with a One Liner and multi line documentation. Most symbol editor dialogs contain a “documentation” tab. The standard documentation view cannot be used to edit the symbol’s documentation because the diagram editor and documentation view cannot be visible at the same time. The floating documentation view however is coupled to the diagram editor’s focused symbol. This view can conveniently be used to edit symbol documentation.

If a symbol is hotlinked to a code model entity, the symbol’s documentation and one liner will be linked to that of the entity. Changing it in the entity will change it in the symbol and vice versa. This is controlled by hot links and the hot linked properties.

Linked Annotations

Annotations can visually be linked to symbols with documentation links. These links can be either passive or automatically link documentation or one liner. Here is an example from the Getting started demo in this manual.



The annotations in this example are coupled in auto documentation style; the blue double arrows on the link paths show this. Changing the annotation text (for example by in place editing) will change the symbol's documentation. In the example the class symbols are implicitly linked to the code model classes. This means that editing the annotation text will modify both the symbol and class documentation. This also works the other way round: if the code model class's documentation is changed, the annotation text will be updated.

To change the style of a documentation link, double click on the link path.

Diagram Editor

Properties

The environment options "diagrams" tab controls the diagram editor's properties. These include: printing style, hint feed back, background color, grid style and color etc.

Keyboard and Mouse control

The diagram editor's keyboard short cuts are:

Scroll and Move	
Up/Down/Left/Right	Scroll
PageUp/PageDown/Home/End	Scroll one page up/down
Ctrl+PageUp/PageDown	Scroll to top/bottom
Home/End	Scroll one page left/right
Ctrl+Home/End	Scroll to left, right side of page

Zooming	
Numeric + / -	Zoom In/Out by 10%

Ctrl+Shift+I	Zoom in
Ctrl+Shift+U	Zoom out

Editing	
Escape	Cancel operation or select containing (parent) symbol
Ctrl+Z	Undo
Ctrl+Shift+Z	Redo
F2	Rename (invoke inplace editor)
Ctrl+Up/Down/Left/Right	Move selected symbols
Ctrl+C/V/X	Copy/Paste/Cut selection
Ctrl+Alt+C/V	Copy / Paste visual style
Ctrl+Shift+C/V	Copy / Paste symbol style
Del	Delete selection
Ctrl+Del	Delete All (clear diagram)
Ctrl+A	Select All
Ctrl+P	Print Diagram
Ctrl+Alt+P	Print Preview Diagram
F12	Toggle full screen mode

Navigation	
Ctrl+U	Navigate Up; select parent diagram
Ctrl+B	Navigate Backward
Ctrl+F	Navigate Forward

Mouse selection	
Click	Select exclusive
Shift+Click	Toggle selected state, incluse in selection
Drag	Multiple Select(lasso selection)
Shift+Drag	Extend selection by lasso selection
Ctrl+Drag	Parent selection (only select symbols within parent, excluding the parent). Similar to Delphi IDE from designer

Mouse Wheel control	
Wheel	Scroll up/down
Shift+Wheel	Scroll left/right
Ctrl+Wheel	Zoom in/out

Drag & Drop and conversions

ModelMaker extensively supports drag and drop between the main model views (classes, members, units, diagrams etc). Each major model view can act as a drag source of entities and as drop target for entities dragged from most other views. The details of each combination are described in the next paragraphs. Since there are always two views cooperating in a drag /drop operation (source and target) the details of conversions are described for the target view only.

Classes view

Internal (tree mode)

- Change inheritance.
- Apply interface support, press Control to invoke interface wizard

Internal (list mode)

- Apply interface support, press Control to invoke interface wizard

Source

Acts as a source for classes (and interfaces)

- Drag a class or interface to **diagram editor** for instant visualization.
- Drag a class to a **code editor** to insert its name as text.

Target

Accepts members, local vars, procedures and event types:

- Drop **members** from the **members view** on a class to copy them to the target class. Press Shift (before releasing) to Move rather than copying the members. If a property is copied or moved, it's read and write access members are also copied, even if not included in the dragged members. Restrictions that apply for interfaces are automatically applied: fields are ignored, property write access is restricted and visibility is made default when dropping a class's members on an interface.
- Drop **procedures** from **Method Local Code Explorer** to convert them to new methods.
- Drop **procedures** from **Unit Code Explorer** to convert them to new methods.
- Drop l**ocal vars** from the **Method Local Code Explorer** to convert them to new fields. Press shift on dropping to move rather than copy the var. Since interfaces cannot contain fields, local vars cannot be dropped on interfaces.

- Drop **event type definitions** from **Event library** view and **Units view** (tree mode) to add an event handler or event property using the dragged event as a template. On dropping a popup menu offers the available options.

Members view

Internal

- In custom order rearrange mode (Ctrl+R toggles this mode), member custom order can be arranged.

Source

Acts as a source of members.

- Drag members the **classes view** to copy them to a class or interface.
- Drag members to a **class** in the **units view** (tree mode only). Similar to dragging members to the classes view
- Drag a field or method to the **Method Local Code Explorer** to convert it to a local var.
- Drag a method to the **Unit Code Explorer** to converted it to a local procedure.
- Drag a method to the **Method Implementation view** (toolbars, or tabs) to "pin" the method.
- Drag a method(s) to the **Event Library view** to create new event types using the methods as template.
- Drag a member to a **code editor** to insert its name as text.

Target

Accepts code sections, local vars, (local) procedures, text, event type definitions and text.

- Drop **code sections** from the **Method Implementation Section list**. The dropped section is copied to the target method. Only if dropped on a method.
- Drop a **Local var** from the **Method Local Code Explorer.** If a var is dropped on a method, the var will be copied to the target method. Press shift to move the var rather than copying it. If a var is dropped on any other member or empty space in the member list, the var is converted to a new field. Drag the vars root node in the method local code explorer instead of a single var to drag all all vars at once.
- Drop a **local procedure** from the **Method Local Code Explorer.** If a local procedure is dropped on a method, it will be copied to that method. Press shift to move rather than copy it. If a local procedure is dropped on any other member or empty space in the member list, the local procedure is converted to a new method.
- Drop a **local procedure** from the **Unit Code Explorer**. Acts similar as local procedures dragged from the Method Local Code Explorer.

- Drop **Event type definitions** from **Event Library View** and **Units view (tree mode)**. A popup menu will let you select between adding an event handler for an event type or creating an event property. Multiple event types may be dragged at once.
- Drop **Text** from the **code editors** on the “add field” “add method” “add property” and “add event” buttons in the toolbar. The corresponding member type will be created and its name will be set to the dragged text. Method parameters are extracted from the dropped text.
- Drop **Text** from **code editors** dropped on the member list acts similar as text dropped on the“add method” button.

Units view

Internal (tree mode only)

- Classes and Event types can be copied or moved between units and “Classes not assigned to any units”. Pressing Ctrl will copy rather than Move (default). Within the same unit classes and event types can be rearranged using drag and drop.

Source

In tree mode acts as source for units, classes and event types. In list mode acts as source for units only.

- Drag a unit to the **diagram editor** to visualize that unit as a package.
- Dragging a class or interface to any other view is similar to dragging a class from the classes view.
- Dragging an **event type** is similar to dragging an event type from the event library view.
- Drag a unit, class or event to a **code editor** to insert its name as text.

Target

Accepts members, members, (local) procedures, local vars (all in tree mode only) and event types (both modes).

- Drop an **event type** from the **Event Library view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class this will add an event handler or event property using the dragged event as a template or alternatively add the event type to the unit. On dropping a popup menu offers the available options. Applies to both tree and list mode.
- Dropping entities on a class is similar to dropping on a class in the classes view (tree mode only).
- Event types contained by units do not accept dropped entities.

Method Implementation view

Method Local Code Explorer

Internal

- Rearrange local vars
- Rearrange local procedures

Source

Acts as a source for **local vars**. Dragging the vars root node will drag all vars at once. Press shift to move rather than copy / convert a local var.

- Drag a Local var (or the root "Vars" node) to a class in the **classes view** to convert it to a field.
- Dragging a Local var to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a local var to the **members view** to copy it to a method or add a new field in the class.
- Drag a local var to a **code editor** to insert its name as text.

Acts as a source for **(local) procedures**. Press shift to move rather than copy/convert a procedure

- Drag a local procedure to a class in the **classes view**. The procedure will be converted to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.
- Drag a procedure to the **members view** to copy it to a method or add a new method.
- Drag a procedure to a **code editor** to insert its name as text.

Target

Accepts members, local vars, procedures and event types:

- Drop a **field** or **method** from **members view**. A field will be converted to a local var, a method will be converted to a local procedure. This is usually only relevant for "pinned" methods as selecting a member to drag it will automatically change the "current method".
- Drop **text** from a code editor containing an "`ident + ":" + type`" list to convert the text to local vars.

Method Implementation Section list

Internal

- Rearrange sections (drag up/down)
- Indent / unindent sections (drag left/right)

Source

Acts as a source for code sections.

- Drag a section to a method in the **members view** to copy it to that method. Press shift to move rather than copy the section.

Target

Does not act as external drop target.

Method Implementation Code Editor

Internal

- Rearrange text inside editor copy or move.

Source

Acts as a source for text.

- Drag a **text** to the members view to add a new member, using the text as name (plus parameter list for methods).
- Drag a **text** containing an "`ident + ":" + type`" list to the Local Code Explorer to convert the text to vars.

Target

Accepts all dragged entities and inserts the associated name at the drop point.

Unit Code view

Unit Code Explorer

Internal

No internal drag and drop support.

Source

Acts as a source for (local) procedures. Press shift to move rather than copy/convert a procedure.

- Drag a local procedure to a class in the **classes view** to convert it to a method.
- Dragging a local procedure to a **class** in the **units view** (tree mode) is similar as dragging it to a class in the classes view.

- Drag a procedure to the **members view**. This is similar to dragging a local procedure from the Method Local Code Explorer.
- Drag a procedure to a **code editor** to insert its name as text.

Target

Accepts methods and procedures.

- Drop a **method** from the **members view** to convert it to a module procedure.

Unit Code Editor

Similar to Method Implementation view Code Editor.

Event Library view

Internal

Does not support internal drag and drop.

Source

Acts as a source for event type definitions.

- Drag an event to the **members view** or on a class in the **classes view**. This will add an event handler or event property.
- Drag an event to the **units view**. If dropped on a unit, this will add the event type definition to the unit. If dropped on a class in the units view (tree mode) this is similar as dropping it on a class in the classes view.

Target

Accepts methods.

- Drop a **method** from the **members view**. For each dropped method an event type will be created that takes the method's signature (name, parameter list, result type etc) as a template.

Diagrams view

Internal (tree mode only)

- In "sort hierachical" mode, diagrams can be rearranged. Pressing Shift will change diagram hierarchy relations. Other modes: no internal drag drop support.

Source

Does not act as external drag source.

Target

Does not act as external drop target.

Diagram Editor

Internal

The selected editor tool controls the extensive internal drag drop support.

Source

Does not act as external drag source.

Target

Accepts classes and units, depending on the opened diagram.

- A **class** or **interface** dragged from the **classes view** or **units view** will be visualized as class or interface symbol, object flow symbol or role symbol. For class diagrams relations with other classes are automatically visualized. If three or more classes are dropped on a class diagram the “drop visualization wizard” will be invoked. This wizard allows selecting the visualization scheme and relations to visualize.
- A **unit** dragged from the **units view** will be visualized as package symbol. Relations with classes (contained) and other packages (uses and used dependencies) are automatically visualized.

Customizing ModelMaker

Here are some links to customizing ModelMaker to your wishes. Most of them you'll find in the Environment and Project options dialogs. For details refer to the GUI reference, here are just a few:

- To adjust the appearance of ModelMaker use the Environment options
- To adjust prefixes of property access methods and fields: use Project Options|Coding style.
- To adjust the layout of the generated source code, use Project Options|Code Generation.
- To adjust the way ModelMaker imports source code: use Project options|Code Import tab.
- To adjust the in-source documentation generation and import settings, check the corresponding tabs in the project options.
- To define a basic diagram visual style: Project options|Diagram Style
- To define a basic symbol style (displayed members etc): Project options|Diagram Style
- For defining the code editor's shortcut keys, Use Environment Options|Shortcuts.

Here are some other links to customization:

- Add Version Control capabilities by using a plug in expert. (Check ModelMaker web-site for ready available third party VCS Experts)
- To adjust the unit template used for new units, refer to Adjusting the unit template, page 50.
- Create Parameterizable Code Templates for pattern like groups of members that appear in multiple classes and models.
- For creating model templates, refer to Model templates page 41.
- Define and use your own macros for use in code generation or in the code editor.
- Most views have special display settings that are controlled in their popup menu: In Members view you modify toolbar layout and sorting. In the Classes view you adjust navigation order and history etc.
- Use the MM OpenToolsApi to create you own experts.

Integration with the Delphi IDE

ModelMaker is a stand-alone application and you don't need Delphi to run it. Integrating ModelMaker with Delphi's IDE will enable some additional features.

- ModelMaker will automatically update Delphi's code editor whenever a source file is (re) generated.
- You'll be able to access Delphi's on-line context sensitive help from within ModelMaker.
- Call Delphi's 'Syntax Check' "Compile" or "Build" commands from within ModelMaker.
- Open source files and locate the member selected in ModelMaker in the Delphi IDE from within ModelMaker

Inside the Delphi IDE integration experts add several features that enable smooth integration:

- Synchronize ModelMaker with the IDE: refresh import a unit and locate the current member.
- Add (multiple) files to a ModelMaker project

Although it is possible to integrate ModelMaker with all versions of Delphi and run multiple instances or versions of Delphi at the same time, it is recommended that only one is running when working with ModelMaker, as integration is a based on a one-to-one connection.

For the same reason we suggest that you do not run multiple instances of ModelMaker. In the environment options General tab you'll find an option that will ensure this.

Integration with Delphi 3 and higher

ModelMaker is integrated with Delphi 3 and higher by use of an integration expert. These experts are automatically installed by the setup program for the IDE versions you have installed on your PC. You can manually (un)install an IDE expert later using the ModelMaker environment options "Delphi IDE" tab.

The experts add a ModelMaker main menu item to the IDE's menu bar. The ModelMaker menu contains:

- Run ModelMaker, (if not already running)
- Jump to ModelMaker (Ctl+F11 in D3, Ctrl+Shift+M in D4 or higher) - this will activate ModelMaker and select the unit, class and member corresponding to the IDE's topmost editor's position. Note that inside MM the inverse command 'Locate in Delphi' - main menu "Delphi" or main tool bar 'Locate in Delphi' - which locates the member selected in ModelMaker in the Delphi editor. ModelMaker's 'Locate in Delphi' command also has shortcut Ctl+F11.
- Add to Model, adds the topmost unit in the Delphi editor to the current MM model.
- Add files to model, lets you select which files to import in a model. In D4 and higher you may select files contained in a project or files opened in the editor.
- Refresh in Model, will re-import the topmost unit just like add to model, but only if the unit is already in the model. If the unit is not in the model, the command is silently ignored.

- Convert to Model, creates a new model and adds the topmost unit to this new model.
- Convert project to Model, which creates a new model and adds all files in the current Delphi project to the new model.

Note that the file in the IDE editor will be SAVED prior to performing the actual command. Therefore these commands won't work on read-only (project) files.

Delphi 4 and higher

The Delphi 4 and higher integration experts have additional commands in the IDE's ModelMaker menu.

- Open Model, opens the model associated with the top most file in the IDE editor. Check the web-site 'Tips' page for details.
- 'Enable Auto refresh' and 'Lock Auto refresh on Run'. These control the Auto refresh feature that is described in detail in chapter Auto Refresh Import, page 65.
- Version Control. This menu item is enabled if you integrated a Version Control system in ModelMaker using a (third party) VCS expert. The available commands depend on this VCS-expert. They usually include at a minimum Check-in and Checkout. Check the ModelMaker Tools web site for available VCS-experts.

Some additional tools and utilities

- Unit Dependency analyzer. This is the same tool as available inside ModelMaker. For a description, check the ModelMaker on-line help by pressing F1 in this tool in ModelMaker.
- Resource string wizard. This will scan a unit for hard coded strings and help in converting them to a section of resource strings or string constants.
- String to Resource string, similar to the Resource string wizard, but only handles the current string token in the editor.
- Shortcut wizard: checks the active form for duplicate keyboard hot keys in control captions like "&Apply this" and "&Surprise me" and suggests alternatives in case of conflicts.

In Delphi 4 and higher you can also add commands to the IDE toolbars. In Delphi's toolbar 'Customize...' dialog, you'll find these commands in the ModelMaker category - right click on Delphi's toolbar, go to 'all commands'.

Delphi 4 and higher syntax highlighting scheme

In ModelMaker you can specify the syntax-highlighting scheme to mimic your settings in the Delphi IDE. Unlike Delphi 1/2/3, Delphi 4 and higher do not define the default color scheme in the registry unless you manually (re-)define it. If the syntax highlighting scheme in ModelMaker is set to "Delphi 4" or higher, it might display strange settings: anything could happen such as underlined, blue colored normal text.

In order to solve this problem, in Delphi 4+ go to environment options and on the Colors tab define all fore-and background colors you want by explicitly selecting them rather than relying on the 'use default' check boxes. The same applies for the font styles: you must explicitly

select them. After applying these settings and restarting ModelMaker, you should have the highlighting scheme you selected. The following entries should be explicitly defined: "Comment"," Identifier", "Number", "Plain text", "Reserved word", "String", "Symbol", "White space" and "Marked block".

Uninstalling IDE integration experts

If after uninstalling ModelMaker you still get a message when starting Delphi 3 or higher saying: can't find ..\\..MMEXPT.DLL or similar, you must manually uninstall the ModelMaker integration experts.

To do this:
Either use the ModelMaker environment options 'Delphi IDE' tab to uncheck the IDE version you want uninstall, or
Run RegEdit.exe from the "Start" menu and go to
HKEY_CURRENT_USER\Software\Borland\Delphi\3.0\Experts
There should be an entry called ModelMakerExpert, to uninstall the expert you must manually remove that entry. Higher versions of Delphi have a similar registry entry

Integration with Delphi 1 and 2

Integration with Delphi 1 and 2 does not offer the same functions as the Delphi 3 and higher integration experts. Only basic synchronization functions are supported. And the installer cannot activate the integration - you must install the integration yourself. Integration consists of two aspects:

1. Installing the unit MMINTF.PAS (in directory [installdir]\mmintf in your component library. Installing this unit will enable ModelMaker to:
    - Automatically update Delphi's code editor whenever a source file is (re)generated.
    - Access Delphi's on-line context sensitive help.
    - Call Delphi's 'Syntax Check' command from within ModelMaker.
2. Installing the utility UNITJMP.EXE in your Delphi Tools menu. This will enable you to jump from Delphi's code editor to the corresponding code in ModelMaker and perform some basic file related commands.

Installing the integration unit in Delphi 1 /2

1. Start Delphi 1 /2
2. Add the unit `ModelMaker\6.0\Mmintf\MMIntf.pas` to your component library, just like you would do with any other component (refer to your Delphi user manual).
   In Delphi 1: Select menu 'Options|Install Components',
   In Delphi 2: Select menu 'Component|Install',
   select Add and browse to find the unit `MMINTF.PAS` in folder `ModelMaker\6.0\MMintf\`.

3. Don't be surprised that you won't see any changes in your VCL component palette after Delphi recompiled the VCL: there is no new component installed, just an integration link, which is not a component.

Installing UNITJMP.EXE as a DELPHI 1 /2 IDE tool

Installation of UNITJMP is basically the same for Delphi 1 and 2. To install the UNITJMP.EXE utility, (refer to your Delphi user manual)

1. Start Delphi
2. Select menu 'Options|Tools...'. (*Delphi1.0*)
   Select 'Add' and add a new tool, title it 'Jump &to MM'. (D*elphi 1.0*)
   Select 'Add' and add a new tool, title it 'Jump to &MM'. (D*elphi 2.0*
3. Select 'Browse' to locate the UNITJMP.EXE file in the `ModelMaker\6.0\BIN` folder.
4. Select 'Macros' to pass the parameters '`$ROW $EDNAME`', the space is required; the single quotes ('') should not be entered.
5. Select OK and Close to add this utility.

Now you'll be able to jump from Delphi's code editor to ModelMaker (which should be running) by pressing `Alt+T+T` (D1) or `Alt+T+M (D2)`

UnitJump can be installed more than once as a tool to perform different integration tasks, each time passing different parameters.

1. For automatic refreshing of the top most file in the IDE editor, pass parameters '-1 $EDNAME', refer to "Refresh Import" for details. You could enter '&Refresh Import" as title.
2. To add the topmost in the Delphi editor to the current model, use parameters '-3 $EDNAME'.
3. To create a new model and add the topmost in the Delphi editor to this model, use parameters '-2 $EDNAME'. You could enter '&Convert to Model' as title.

Yes: you have identical "tools" `UNITJMP` now; the only difference is the parameters passed.

MMToolsApi primer

This chapter is a introduction on COM interfaces and using the MMToolsApi to create your own experts to ModelMaker's functionality. You should also check the MMExptDemo.dpr that demonstrates most aspects of an expert. If you have not noticed yet: in the ..\ModelMaker\demos directory there's a model MMToolsApi.mpb that contains two diagrams showing relations of the API. Do not use this model to re-create the MMToolsApi.pas, this file will be changed in future versions.

Interfaces basics

Interfaces are like classes in the way that once you've got a pointer to an interface, you can call methods and read/write properties.

Assume for example you have an interface pointer CodeModel: IMMCodeModel. From the MMToolsApi unit you can see that this interface supports the ClassCount and Classes[idx] properties. Also you can see that Classes[idx] returns IMMClass interface pointers. You could now for example iterate the code model for classes and list their names in a list box

```
for I := 0 to CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

Expert DLL basics

The big thing is now: how do you get the first interface pointer CodeModel, because once you've got hold of that, everything is easy. In the MMToolsApi there is a central access point called MMToolServices: IMMToolServices that is declared as a global var in unit MMToolsApi. This interface var is initially nil but gives access to all major aspects of the MM engine such as the CodeModel in the above example. ModelMaker Experts are dll's which are loaded dynamically (all experts must be placed in directory ..\[installdir]\experts.) After loading the library MM looks for a procedure called MMExpertEntryProc, as defined in MMToolsApi.pas and calls this procedure passing the interface pointer of the actual MMToolServices object. You should store this interface pointer as it provides your central access the tools API. You can access this interface and read the CodeModel: IMMCodeModel interface as shown in this example

```
procedure EntryProc(const Srv: IMMToolsServices); stdcall;
begin
  // here you get passed the interface pointer, store it to use later.
  MMToolServices := Srv;
end;
```

The entry procedure should be exported named SMMExpertEntryProc (MMToolsApi.pas). There's also an exit procedure, which is called upon termination of ModelMaker. In this procedure we use the previously stored MMToolServices as central access point:

```
for I := 0 to MMToolServices.CodeModel.ClassCount do
  ListBox.Items.Add(CodeModel.Classes[I].Name);
```

MMToolsApi version control

The unit MMToolsApi is continuously under construction and version control is governed by an ExpertVersionProc. This is the first procedure ModelMaker attempts to call when loading the expert. In your expert you must export this function and return the MMToolsApiVersion constant as defined in unit MMToolsApi.pas. ModelMaker will reject any expert not exporting this procedure or experts that do not match the version of the MMToolsApi with which ModelMaker was created.

```
function ExpertVersion: LongInt; stdcall;
begin
  Result := MMToolsApiVersion;
end;

exports
  ExpertVersion name SMMExpertVersionProc;
```

Interfaces and memory management

Memory management for interfaced classes is controlled by reference counting which is automatic done for you by the compiler. Even assigning an interface to a local var will increase the reference count automatically, as will assigning nil to the var or going out of scope (exiting a procedure) will decrease the reference-count again. This means you don't have to worry about freeing classes after creating them. In general all interfaces passed on from ModelMaker to you are objects existing in MM.exe space. All interface pointers you pass on to ModelMaker are objects you create (such as an expert) and ModelMaker will only have access through the interface pointer. Since these objects are reference counted the objects will disappear after MM and the expert both drop all references to it.

For example if you retrieve an IMMClass interface pointer from ModelMaker and use it in your expert code, the actual interface object will exist as long as you keep a reference to it for example by assigning to a global var. The actual class the IMMClass refers to, may be gone in ModelMaker due to user actions (delete the class, open a new model) but the interface object remains live until the expert drops all references. To check whether an interface is actually connected to a real class you could /should check the Valid property. If Valid is False, the actual class object does not exist anymore, just the interface object. The interface object will return default values in all functions (such as GetName =").

Adding an expert and menu items

After reading the COM interface basics, you might want to create something useful which initiates on user action. The MMToolsApi provides the IMMExpert mechanism for this. You can create an object implementing IMMExpert and register it in ModelMaker, again using the MMToolServices. The fun about IMMExpert is that it allows you to insert menu-items in the ModelMaker main menu bar Tools menu and some predefined pop-up menus. Each expert should support properties Verbs, VerbCount and MenuPositions. VerbCount defines the number of menu-items you want inserted and Verbs are the actual menu item Captions. MenuPositions[..] defines where to which menu an item should be added. If the user clicks one of these menu-items, the Execute(Idx) method of the expert will be called, where Idx is [0..VerbCount -1]

There are some more methods supported by IMMExpert, but these are the basics. In the MMExptDemo.dpr you'll find an object implementing IUnknown and IMMExpert. If you go through the associated code you'll see that it has only one Verb and does only one thing in Execute. This demo expert could serve very well as a base for all other experts.

How do you inform ModelMaker that you want such an expert to be installed? Well create it (instantiate the class) and register the interface pointer with ModelMaker using MMToolServices.AddExpert(..). Reference counting will again take care of memory management. On shutdown you should remove your expert again using the index that was passed by AddExpert.

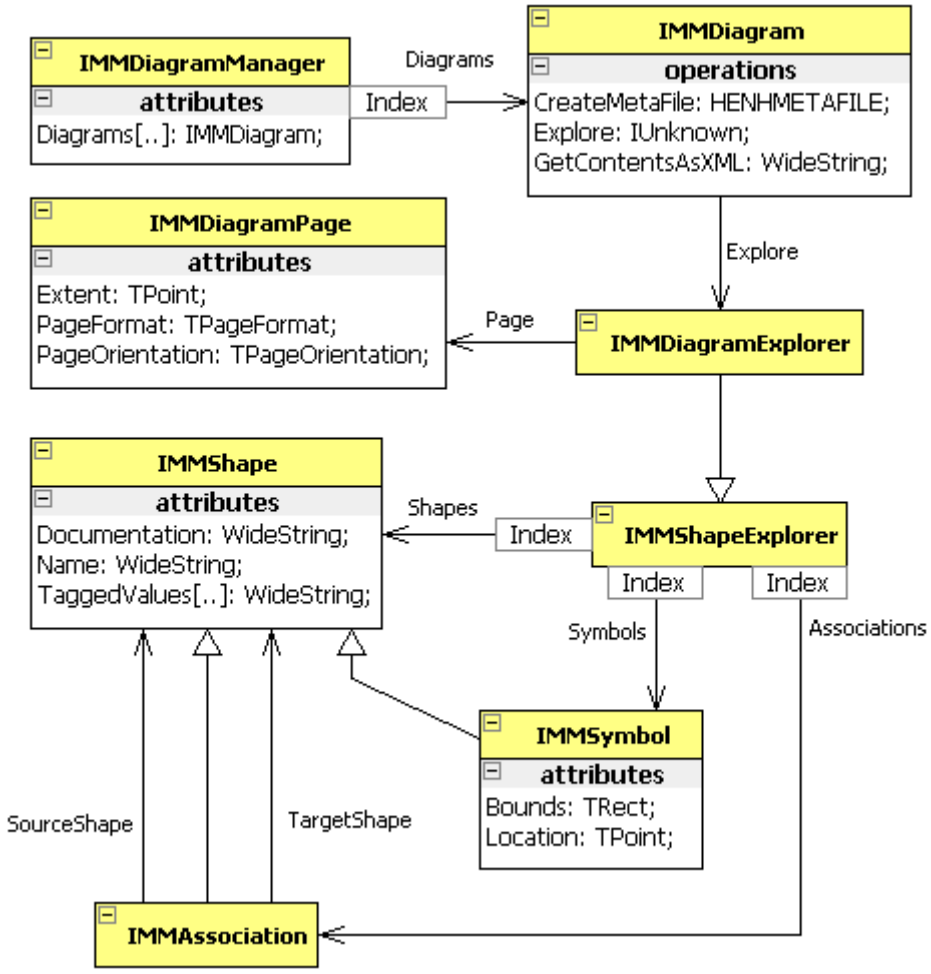
```
var
  ExptIndex: Integer = -1;

procedure MMExpertEntryProc(Srv: IMMToolServices); stdcall;
begin
  MMToolServices := Srv;
  ExptIndex := MMToolServices.AddExpert(TMyExpert.Create);
end;

procedure MMExpertExitProc; stdcall;
begin
  if ExptIndex <> - 1 then MMToolServices.RemoveExpert(ExptIndex);
end;
```

Suppose you want to create an expert that does two reports "Simple" and "Extended", you should create an object that returns VerbCount = 2 and Verb[0] = 'Simple', Verb[1] = 'Extended'. To add the items to the Tools menu, return mpToolsMenu in GetMenuPositions. Then in Execute(Idx) you check whether Idx = 0 or 1 to either create the Simple (0) or extended (1) report. The actual report creation could be something like:

- Create a form containing a list box
- use
  ```
  for I := 0 to MMToolServices.CodeModel.ClassCount do
    ListBox.Items.Add(CodeModel.Classes[I].Name);
  ```
  to fill the list box, and
- Call the Form's ShowModal method to show the current class list

Accessing Diagrams through the API

The following picture explains how to access the Diagrams and their contents through the MM ToolsAPI.



The IMMDiagramManager property of the MMToolsServices is the entry point that gives access to all diagrams. The IMMDiagram interfaces are life pointers to the actual diagrams as displayed in the diagram list view. To get at the contents of a diagram, use the IMMDiagram.Explore method. This will create a Diagram Explorer similar to the diagram editor. The interface this method returns can be cast as an IMMDiagramExplorer. The definition of IMMDiagramExplorer and the symbols can be found in MMDiagramAPI.pas.

Note that each time you call Explore, as new explorer is created. And that the contents of two explorers are not linked. If you create two explorers and modify a diagram in explorer_1, in explorer_2 you won't see the change made in explorer_1.

An explorer gives access to a diagram's symbols and associations. The Shapes property simply concatenates the symbols and associations properties. An IMMShape gives access to basic shape behavior: name, documentation and hyperlinks. An IMMymbol gives acsess to symbol specific properties like Bounds and Location (which associations do not have). IMMAssociation defines the association specific properties like SourceShape and TargetShape.

There are many ways to manipulate a diagram, the ModelMaker Tools demo expert shows a few examples like: create a sequence diagram and create an image containing a single class.

Accessing Experts through scripting

ModelMaker 6 is a self-registering COM server that allows access to plug-in experts that support IDispatch. The ModelMaker type library can be found in the [installdir]\experts directory. It contains interface IApp that contains a single method: GetExpert.

```
  IApp = interface(IDispatch)
    ['{D077CEC1-83F0-11D5-A1D2-00C0DFE529B9}']
    function  GetExpert(const ExpertID: WideString): IDispatch; safecall;
  end;
```

The parameter ExpertID is used to locate the expert based on the value returned by IMMExpert.ExpertID.

If your expert supports IDispatch and inherits from TAutoObject, you can access it for example in a java script like this (assuming your expert has a method named TestMethod which takes a single WideString parameter).

```
// Java script
var mm = new ActiveXObject("ModelMaker.App");
var api = mm.GetExpert("ModelMakerTools.ScriptingDemoExpert_10");
api.TestMethod("Hello World");
```

The above example will start ModelMaker or locate the active instance. Then locate the test expert and call it's method named "TestMethod".

This mechanism can be used to expose specific interfaces to scripting. For example, assume you have a reporter plug-in that supports this interface:

```
type
  IMyReporter = interface(IDispatch)
    procedure CreateReport(const ModelName, ReportName: WideString); safecall;
end;

  IMyReporterDisp = dispinterface
    ['{F9BA1301-84EB-11D5-A1D2-00C0DFE529B9}']
    procedure CreateReport(const ModelName, ReportName: WideString); dispid 1;
  end;
```

You could then call this expert from a script to load a model and create a report.
To learn more about disp interfaces and IDispatch, please check the Borland Delphi developers guide.