CodeSite 2.0 Professional: For Adults Only...

Reviewed by Dave Jewell

Nowadays, software developers have access to a huge arsenal of debugging tools, such as black-box flight recorders (eg Mutek's BugTrapper), profilers, integrated IDE debuggers and other utilities for peeking inside your own (and other people's) code.

CodeSite is the brainchild of Ray Konopka, creator of the respected Raize Components. You can find his website at www.raize.com, where there's more information on the Raize family of products, including the intriguingly named BabyType, which is designed to entertain and educate small children by turning your PC into a children's activity centre.

In the adults-only department, you'll find the recently released CodeSite 2.0 Professional, a debugging tool which is specific to VCL developers. It supports Delphi 3.01, 4.02 and 5.0 as well as C++Builder versions 3, 4 and 5. With the release of CodeSite 2.0, earlier versions of C++Builder and Delphi are no longer supported. The documentation states that you'll need up to 11Mb of disk space: although this seems to be a very generous guideline, exact requirements depend upon how many installed versions of Delphi/C++Builder you have, and whether or not you install support for all of them. Undoubtedly, support for Delphi 6 will be provided soon after the new development system is released.

I reviewed CodeSite 2.0 using Delphi 5. In this particular case, the install program automatically added all the necessary components and integrated the CodeSite 2.0 help into Delphi. I was impressed to see that the installer auto-detected the presence of CodeRush and popped up a message to tell me that a set of CodeSite keyboard templates had automatically been integrated into CodeRush. As the Americans say: tender!

Actually, it's no surprise that CodeSite and CodeRush have a fairly intimate relationship, since CodeSite has a recommendation from Mark Miller (the brains behind CodeRush) on the Raize website, and Mark's name appears in the credits on the CodeSite `About` dialog.

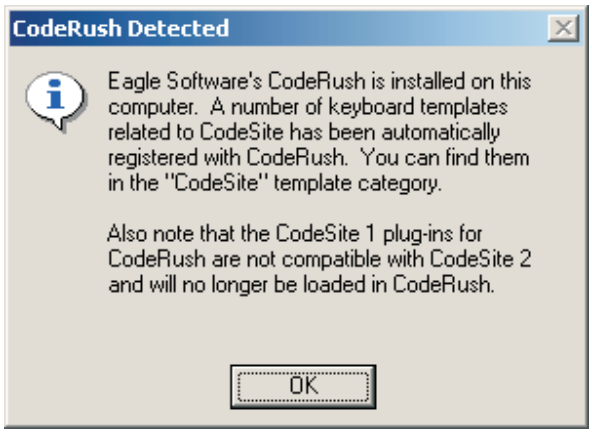
CodeSite 2.0: What's It For?

So what is it, and what do you do with it? Figure 2, which I've derived from the CodeSite documentation, is intended to give you 'the big picture'. The basic idea is that your application, shown by the upper, light grey, box in the illustration, generates a set of diagnostic messages during execution. The other, darker grey, boxes indicate that you might potentially be running multiple instances of your application, or even several different applications, each of which is CodeSite-enabled. Regardless of the exact scenario, all these debugging messages are sent to a single, centralised CodeSite dispatcher.

The dispatcher is a redistributable application (only around 1Mb in size) which you can supply to your customers. This in turn means that you've got a similar scenario to that offered by the likes of Mutek's BugTrapper, whereby it's possible to easily obtain diagnostic info relating to a computer at a remote site. As you can see, CodeSite offers a great deal of flexibility in where these messages get sent to; in the simplest case, it might be to a viewer program or log file on the same machine as the application being debugged, but CodeSite also allows you to set up a TCP or UDP connection to a remote PC. You can even dispatch messages to a web server, which would then forward them on to another dispatcher module via TCP.

With CodeSite installed, fire up the Delphi IDE and you'll find two new components on your palette, `TCSObject` and `TCSGlobalObject`. These correspond to the magenta coloured objects in the diagram which really form the heart of the system. It's these components which are responsible for generating messages and sending them to the appropriate dispatcher. For an explanation of the difference between these two components, see later.

Both components include a number of properties, including `DestinationDetails` which uses a very cute-looking property editor (Figure 3) to configure the intended destination for all generated messages. In the simplest



➤ *Figure 1: As with Raize Components, CodeSite exudes attention to detail. The install program automatically detects CodeRush and adds a set of relevant keyboard templates to the IDE.*

case, you'd click the `Viewer` tab and check the `Viewer` checkbox, thereby sending all messages to the local viewer application. More complex scenarios let you set up a log file, a TCP or UDP connection, or whatever. A nice touch here is that when configuring a TCP or UDP connection, you can specify the destination using either a fully qualified domain name, an IP address, or the name of the computer. Another nice touch is the `Alias` tab which (somewhat like database aliases) allows you to set up the details of a specific destination in the dispatcher's .INI file, referring to it by name. Oh yes, and you're probably thinking that those cool-looking tabs are a nice touch, aren't you? I thought so too! Never fear, Ray has promised to include them in the next version of Raize Controls...

Incidentally, referring back to Figure 2, I should stress that the remote dispatching facilities in CodeSite are specific to the Professional version of the product, which is what I'm reviewing here.

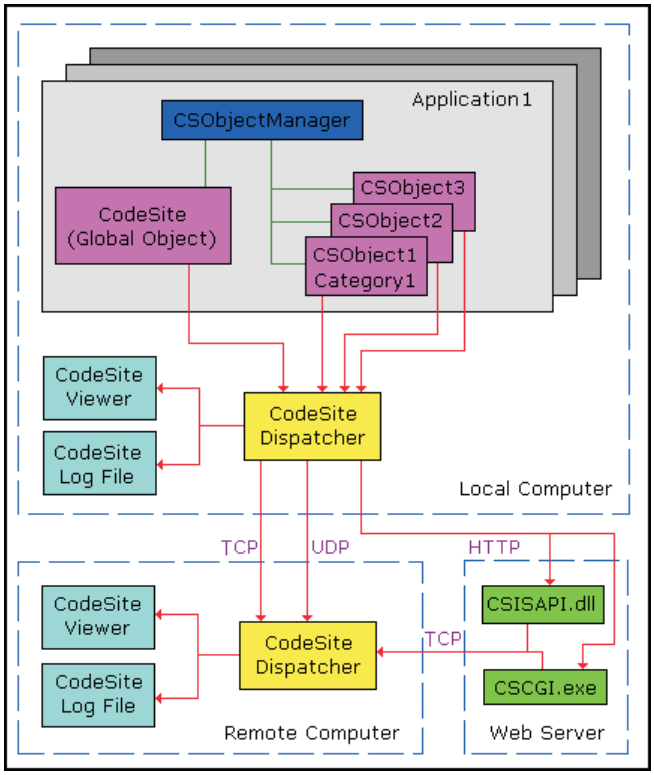
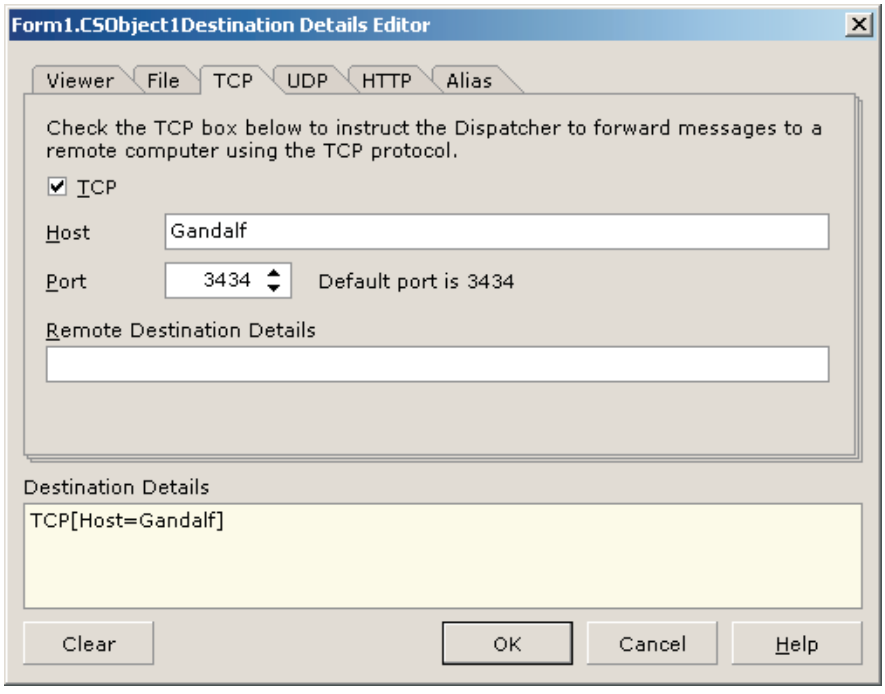


Figure 2: The big picture. CodeSite can use TCP, UDP or HTTP to communicate with a remote dispatcher or web server. Note that only the functionality in the upper blue-dashed rectangle is available with the Standard edition.

The uppermost dashed blue box encompasses the functionality provided by the Standard version, whereas the Pro product adds the TCP, UDP and HTTP facilities. I should also point out that you can set things up so as to despatch messages to multiple destinations simultaneously. So, you can (for example) examine messages in the viewer application whilst at the same time copying everything to a log file.

Figure 3: Double-clicking on a TCSObject or TCSGlobalObject brings up this property editor from where you can select one or more destinations for diagnostic messages. Here, I'm sending messages to Gandalf on the local network.



Making Messages

As you'll have gathered by now, CodeSite lays a strong emphasis on the versatility of its message dispatching. But none of that is going to be of any interest if the product doesn't actually deliver a set of powerful features for generating messages. There are a whole host of methods to choose from, some of which are trivially simple, and some of which are not. Just to give you a flavour of how it all works, let's suppose you begin by dropping a `TCSGlobalObject` onto your form. This will automatically pull in the CSIntf.PAS file, for which Ray usefully provides the full source code. The initialisation clause of this unit creates a global variable of class `TCodeSite` called `CodeSite`, and this object is the primary mechanism for creating messages.

Thus, to indicate that a specific type of form has been created, you might do something like this:

```
procedure TForm1.FormCreate(
  Sender: TObject);
begin
  CodeSite.SendMsg(
    'Created instance of ' +
    ClassName);
end;
```

This will send the message `Created instance of TForm1` to the dispatcher, which in turn will send it to whatever destination you have

chosen. When you execute your application, the code inside `CSIntf` automatically starts the dispatcher application which, in turn, causes the viewer program to be invoked. Well, so what, you may say? This is just a very fancy way to execute a `Writeln` statement! Actually it gets a lot more interesting than that. Here's another code fragment to chew on:

```
procedure TForm1.Button1Click(
  Sender: TObject);
begin
  CodeSite.SendObject(
    'The form', Self);
end;
```

Rather than sending just a string, this method sends a descriptive string followed by a list of all the published properties of the form, together with the current setting of each property. This routine isn't `TForm` specific: rather, it uses the standard Delphi RTTI mechanism to enumerate all the published properties of the passed object, obtaining a string representation of each property value. The whole lot appears inside the viewer application, formatted very similarly to the standard Object Inspector view that we're accustomed to seeing. Nested properties are correctly represented, as are set properties.

I would have liked to see some mechanism for suppressing the display of properties which are merely set to their default values (perhaps an additional Boolean flag in the `SendObject` method, or a filter setting in the viewer application) and it would have been nice to have some indication of whether or not a particular event handler has been assigned, but nevertheless, `SendObject` is a useful diagnostic method.

There are a bunch of basic methods for sending simple data types to the dispatcher. Thus we have `SendChar`, `SendFloat`, `SendInteger`, `SendColor` (the viewer displays a sample of the sent colour) and so on. Ray has put a lot of thought into the sort of information that can be passed to the dispatcher, and so we also find ourselves with `SendRegistry` for dumping out part of a registry sub-tree, `SendStream` for generating the stream representation of a component, `SendStringList`, `SendTextFile`, `SendMemoryStats` (which provides a number of important metrics relating to global memory and the local heap) and even `SendBitmap` which sends the bitmap representation of a `TBitmap` object to the ultimate message viewer or logfile. As the documentation tactfully points out, sending the contents of a large bitmap is not recommended because of the potentially large amount of data involved.

You're probably wondering what the difference is between `TCSGlobalObject` and `TCSObject`? In essence, `TCSGlobalObject` is a wrapper around the previously mentioned `CodeSite` global variable, allowing you to configure it at design-time, especially the all-important destination information. This means that, anywhere in your application that you execute some method of `CodeSite`, the message will always be sent to the destination that you've specified in the `DestinationDetails` property of your `TCSGlobalObject` component. You should only ever have one instance of `TCSGlobalObject`, although this isn't enforced. By contrast, you can have several `TCSObject` components, each of which might be responsible for reporting messages of a different category. You can give each of them a distinct category colour, category name, and you can selectively enable or disable each one as required.

➤ *Figure 4: CodeSite provides two sets of message types, general purpose messages, and bound messages. The latter are specific to a particular Inspector in the viewer. You can also create your own custom data messages and inspectors using plug-ins.*

General Purpose Message Types

Constant	Icon	Description
csmInfo		Default message type--Informational message (no attachment)
csmWarning		Warning message (no attachment)
csmError		Error message (no attachment)
csmNote		Informational message highlighted with a yellow icon
csmReminder		Reminder message
csmLevel1/csmRed		Highlighted message with red icon
csmLevel2/csmOrange		Highlighted message with orange icon
csmLevel3/csmYellow		Highlighted message with yellow icon
csmLevel4/csmGreen		Highlighted message with green icon
csmLevel5/csmBlue		Highlighted message with blue icon
csmLevel6/csmIndigo		Highlighted message with indigo icon
csmLevel7/csmViolet		Highlighted message with violet icon

The Dispatcher And Viewer

But before I start rattling on about category names and category colours, notes, message types, the scratch pad, etc, it makes sense to introduce you to the dispatcher and viewer applications. Most of the time, the dispatcher sits in Explorer's tray area, minding its own business. It has popup menu options for ignoring all messages and for viewing its own log (not to be confused with the log file for messages) which primarily shows when connections were established with a running application, when the viewer was started and so on. More interestingly, there's a settings dialog from where you can control which message types and/or category names are blocked or sent on to the destination.

CodeSite implements two different kinds of message type. Firstly, there are the general purpose message types. These message types can be supplied as an

argument to many `CodeSite` methods which accept a message type. When you see a message in the viewer application, it has a small icon next to it which identifies the message type: see Figure 4, which shows the general purpose message types along with their icons.

In addition, CodeSite also implements what are called 'bound' message types: bound in the sense that the message type is bound to a specific data inspector in the viewer application, and is assumed to be of a very specific type. Examples of bound message types are the `csmBitmap` type, which is associated with bitmaps, this message being generated by the previously mentioned `SendBitmap` routine. Similarly, there are message types which correspond to streams, string lists, registry messages, and so on. Within the dispatcher, you can control the transmission of different message types simply by selecting or deselecting the appropriate icon in the dispatcher settings dialog.

Message types aside, you can also categorise messages. Unlike the message type system, nothing is predefined, and you can therefore add your own category names as required. You can also associate a category with a specific colour, and these colours show up in the viewer program. `Category` and `CategoryColor` are properties of `TCSObject`, so this is where it makes most sense to have multiple `TCSObject` components in your program, assigning a different property and colour to each. If you were creating a graphics program, you might have one for image manipulation, another for drawing, one for saving/loading preferences, etc.

You can see the viewer program itself in Figure 5. This screenshot, taken from the Raize website, really captures the heart of what the viewer can do. In the top left region of the window you've got the main message view area, where you can step through and examine individual messages. This can be configured in a variety of ways: in the screenshot you can only see the message itself plus a timestamp field, but you can also choose one or more of: display category, computer name, application name, process ID, thread ID and date. Even if none of these extra fields are displayed, you can still see them in the message detail panel (immediately below the main message area) for the currently selected message.

You might potentially be in a debugging scenario where you are dealing with multiple processes or multiple threads, so the process ID and thread ID are very important. It ought to be obvious here that you could also use CodeSite to debug the network interaction between two different machines, so it is important to have the computer name field as well! There are some nice touches, such as the ability to create new, named, views which filter the message display in various ways. Another nice touch is that the timestamp field can be configured to show the time difference from the previous message, which is

handy for debugging time-critical exchanges.

Below the message detail panel (most of these individual panels can be turned on and off, by the way) you can see the scratch pad area. As the name suggests, the scratch pad is used to store non-persistent, transitory information. In addition to all the `SendXXX` methods that I've discussed so far, you've got another set of methods whose names are prefixed with `Write`. These methods are for sending information to the scratch pad area. Each of these methods takes, as a parameter, a special 'Line ID' string which appears to the left of the message itself. Thus, referring back to Figure 5, you can see two scratch pad messages with Line IDs of `Mouse Cursor` and `Counter`. Whenever the viewer receives a scratch pad message with a Line ID that it hasn't seen before, it adds a new entry to the scratch pad area. However, when it receives a message with a previously encountered Line ID, it removes the old message with that ID and replaces it with the new one. At the same time, the light to the left of the new message is flashed to indicate that a change has taken place. From this, you'll see that the scratch pad is ideal for use as a sort of remote 'watch window', showing the changing values of a variable as you go round a loop, etc.

In the top right-hand area of Figure 5 (partly obscured by the drop-down menu) is a picture inspector which is inspecting a bitmap sent from the debugged application. This is because the currently selected message is a picture message. CodeSite's viewer actually implements no less than nine built-in inspectors for bitmaps, streams, memory blocks, memory status, string lists, objects, colours, registry data and custom data. Yes, that's right, there are classes and methods in the CSINTF.PAS unit which enable you to create your own specialised messages using a custom formatter. You might want to do this to obtain more info on the internal state of one of your own custom components, for example. Over at the viewer end, it's possible to write your own specialised inspector using the plug-in support provided by the viewer. As one would hope, these plug-ins are implemented as standard Delphi .BPL packages, and the Professional version of CodeSite includes four custom plug-ins (with complete source code) just to get you started.

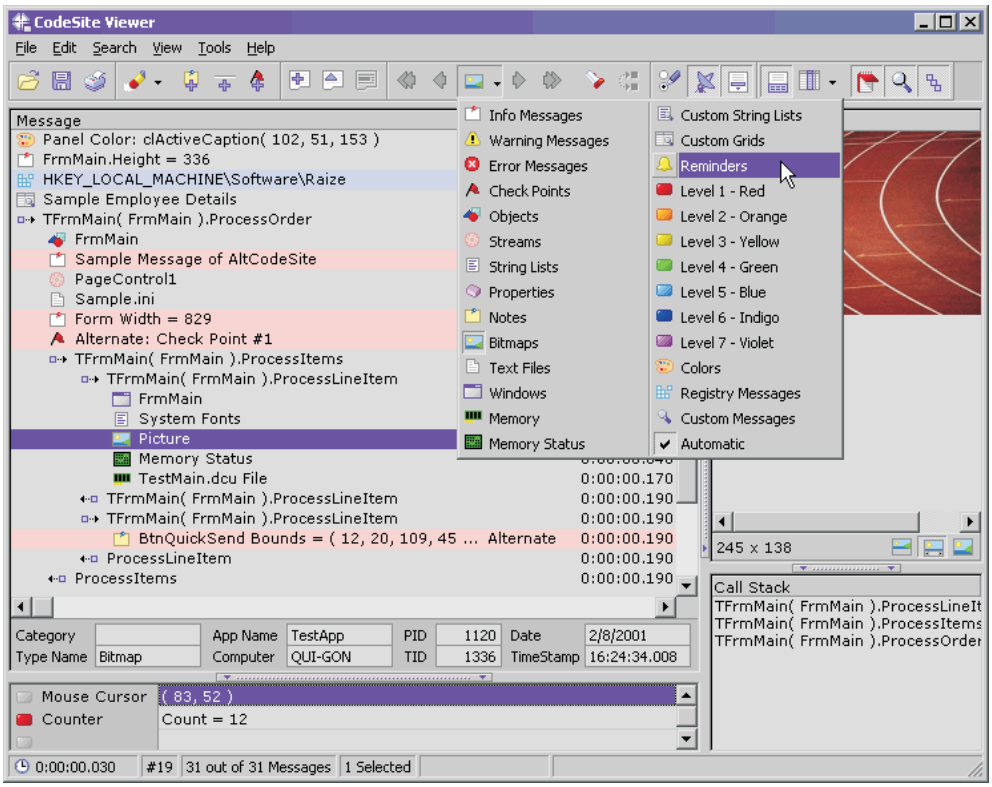
Well, I'm running out of space (and, as usual, time!) but I should briefly mention some of the other message calls that are available. In the bottom right-hand corner of Figure 5 you will see a small `Call Stack` display. One might reasonably ask how CodeSite knows what calling level you're at, considering that all it 'sees' is the messages it gets sent? Here again, we've got a whole bunch of messaging methods that relate to the actual structure of your application rather than the current value of a variable or object. Thus, there are `EnterMethod` and `ExitMethod` calls, which tell the viewer when a method has been entered or exited. This automatically enables the Call Stack window and also causes the viewer to indent messages on entry to a method and 'outdent' (ugh, hate that word!) on exit. If you do use the `EnterMethod` and `ExitMethod` calls, I would advise you to put `ExitMethod` inside the `finally` clause of a `try..finally` block so that even if an exception occurs, the viewer will know that the method has been exited.

Speaking of exceptions, there's an `ExceptionHandler` method which automatically causes any unhandled exceptions to be routed through to the current destination. You can use it as easily as this:

```
Application.OnException :=
 CodeSite.ExceptionHandler;
```

There are checkpoint messages for keeping track of how often a particular line of code is called, and there is even a method for automatically clearing the viewer's message display. This is especially useful at the

➤ *Figure 5: The highly configurable viewer program itself is where you'll do most of your detective work. There is also a standalone log viewer program which is intended specifically for the viewing of the CodeSite-generated log files (.CSL) on the local machine.*



beginning of a lengthy chunk of convoluted code.

Conclusions

As I mentioned at the beginning of this review, CodeSite will allow you to gather debugging and diagnostic information from a remote machine, so it doesn't make much sense to remove all your carefully crafted message-generating calls immediately before you ship the final, debugged (or so you think!) product. Internally, the CSINTF.PAS unit is organised in such a way that if it can't find the dispatcher and viewer applications, the various components automatically disable themselves and the various diagnostic calls all become NOPs. There is very little extra overhead (not much more than the function call itself) in this case, so it doesn't make sense to ship a special version of your code with all the CodeSite calls taken out.

At the lowest level, the actual communication between the debugged application and the dispatcher is carried out using our old friend `WM_COPYDATA` (see my recent articles on desktop manipulation in *Beating The System*) but the code has been written in such a way that you can easily override this and do something else. For example, both the `TCSGlobalObject` and the `TCSObject` components have a single defined event, `OnSendMsg`, and if this is assigned to then messages go to this event-handler rather than the dispatcher. The event-handler has the option of setting a `Handled` flag for each individual messages, and messages that it chooses not to process are sent to the dispatcher in the usual way. Because Ray has made the source code to CSINTF.PAS available, the individual developer has a lot of flexibility in how he/she sets things up, determines where to find the dispatcher, and so on.

A debugging product such as CodeSite stands or falls on its versatility and ease of use. A less invasive tool, such as BugTrapper, is able to work with unmodified executables, but it has the disadvantage of not being able to extract highly detailed information from a running program, and there is no VCL-specific version available at the time of writing. CodeSite, on the other hand, requires that you place diagnostic calls into critical areas of your code, but is highly controllable and has full access to everything that's going on. Because of the way in which Ray Konopka has provided the ability to send custom information from the program being debugged, and custom, plug-in properties in the viewer, there are virtually no limits to the sort of information that can be retrieved, remotely, from an executing Delphi program, and as I've already mentioned, there's nothing to lose (and everything to gain) by leaving all the diagnostic methods calls in your shipping application. CodeSite is a very flexible, powerful and easy to use debugging tool, and I can well understand why Mark Miller recommends it. Now, Ray, how about a Kylix version? ☺

CodeSite 2.0 Professional costs a very reasonable $299 and the Standard edition can be had for $149. These prices are direct from Raize at www.raize.com. Upgrades are available for CodeSite 1.0 customers. If you must, you can also get a Raize shirt and baseball cap...

Dave Jewell is the Technical Editor of *The Delphi Magazine*, email him at TechEditor@itecuk.com