Visual Execution Analyzer in Enterprise Architect

Enterprise Architect is an intuitive, flexible and powerful UML analysis and design tool for building robust and maintainable software.

This booklet explains the Visual Execution Analyzer (Debugger) feature of Enterprise Architect.

Visual Execution Analyzer in Enterprise Architect

Printed: May 2010

Publisher

Sparx Systems

Managing Editor

Geoffrey Sparks

Technical Editors

Geoffrey Sparks
Steve Meagher

Special thanks to:

All the people who have contributed suggestions, examples, bug reports and assistance in the development of Enterprise Architect. The task of developing and maintaining this tool has been greatly enhanced by their contribution.

Table of Contents

Foreword

This user guide provides an introduction to the Visual Execution Analyzer feature of Enterprise Architect.

Visual Execution Analyzer

The *Visual Execution Analyzer* provides facilities to model, develop, debug, profile and manage an application from within the modeling environment.

The Visual Execution Analyzer can generate a number of outputs, including:

- Sequence Diagrams, recording live execution of an application, or specific call stacks
- State Transition Diagrams, a Sequence diagram with states, illustrating changes in data structures
- Profiler reports, showing application sequences and operation call frequency.

These outputs provide a better understanding of how your system works, enabling you to document system features and providing information on the sequence of events that lead to an erroneous event or an unexpected system behavior.

Note:

The Visual Execution Analyzer is available in the Enterprise Architect Professional, Corporate, Business and Software Engineering, System Engineering, and Ultimate editions.

1 Access and Use the Visual Execution Analyser

With the Visual Execution Analyzer, you can create and store custom scripts that specify how to build, test, run and deploy code associated with a package. You can investigate and manipulate the output of the debug process. The Analyzer also includes an Execution *Profiler*, which enables you to determine how the functions in an application are called and executed.

You access the Visual Execution Analyzer using the **Project | Execution Analyzer** menu option, or the context menu of the required package in the Project Browser. These menus provide a number of options to facilitate debugging, such as setting recording options or breakpoints.

The Visual Execution Analyzer can be used to:

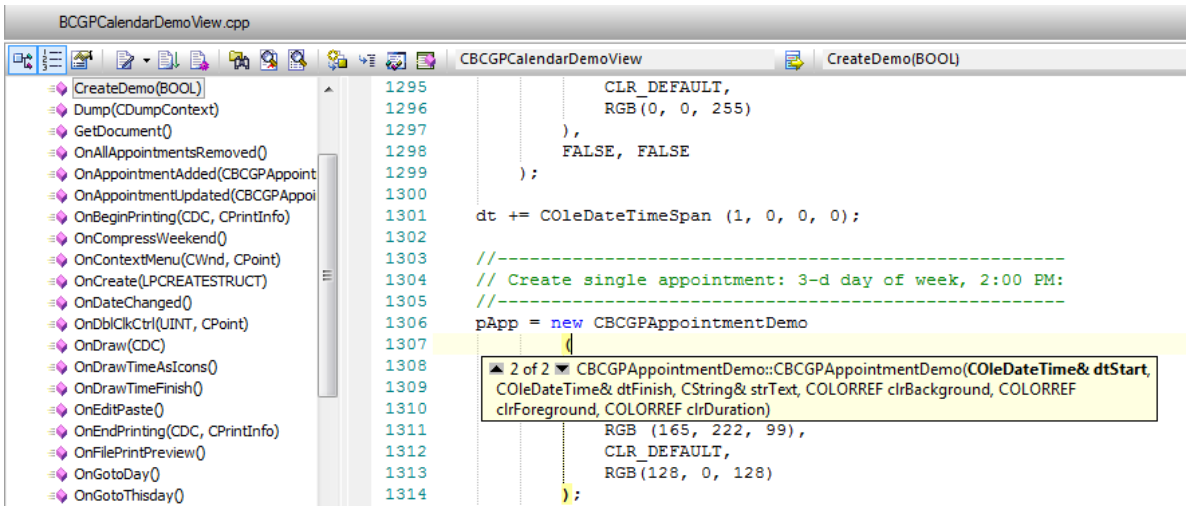
- Optimize existing system resources and understand resource allocation
- Ensure that the system is following the rules as designed
- Produce high quality documentation that more accurately reflects system behavior
- Understand how and why systems work
- Train new employees in the structure and function of a system
- Provide a comprehensive understanding of how existing code works
- Identify costly or unnecessary function calls
- Illustrate interactions, data structures and important relationships within a system
- Trace problems to a specific line of code, system interaction or event
- Visualize why a sequence of events is important
- Establish the sequence of events that occur immediately prior to system failure.

2 Structure of the Visual Execution Analyzer

The Visual Execution Analyzer comprises a Model Driven Development Environment and an Execution Analyzer.

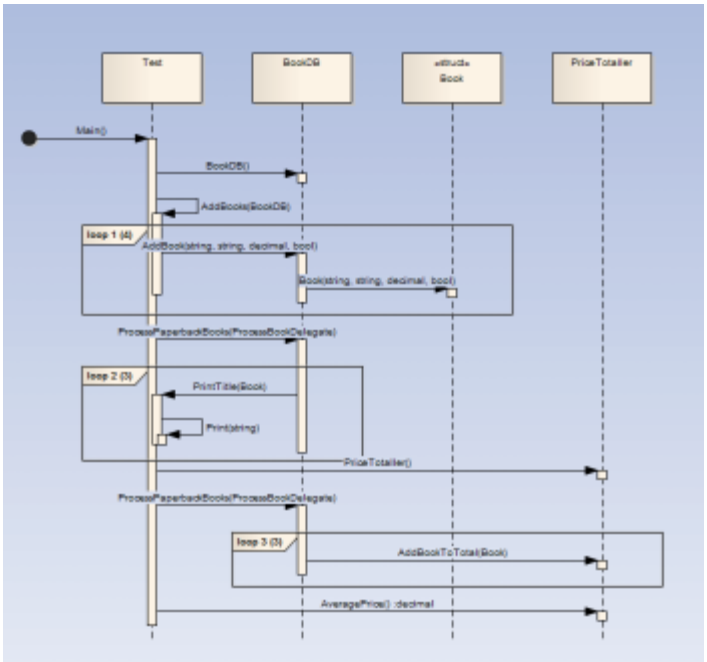
The Model Driven Development Environment 6 (MDDE) provides tools to design, build and debug an application:

- UML technologies and tools to model software (see *Extending UML Using Enterprise Architect* and *UML Modeling With Enterprise Architect - UML Modeling Tool*)
- Code generation tools to generate/reverse engineer source code (see *Code Engineering Using UML Models*)
- Tools to import source code and binaries (see *Code Engineering Using UML Models*)
- Code editors that support different programming languages 11
- Intellisense to aid coding (see *Using Enterprise Architect - UML Modeling Tool*)
- Package scripts that enable a user to describe how to build, debug, test and deploy the application 9.



The Execution Analyzer 57 (EA) provides tools to visualize an existing application's behaviour:

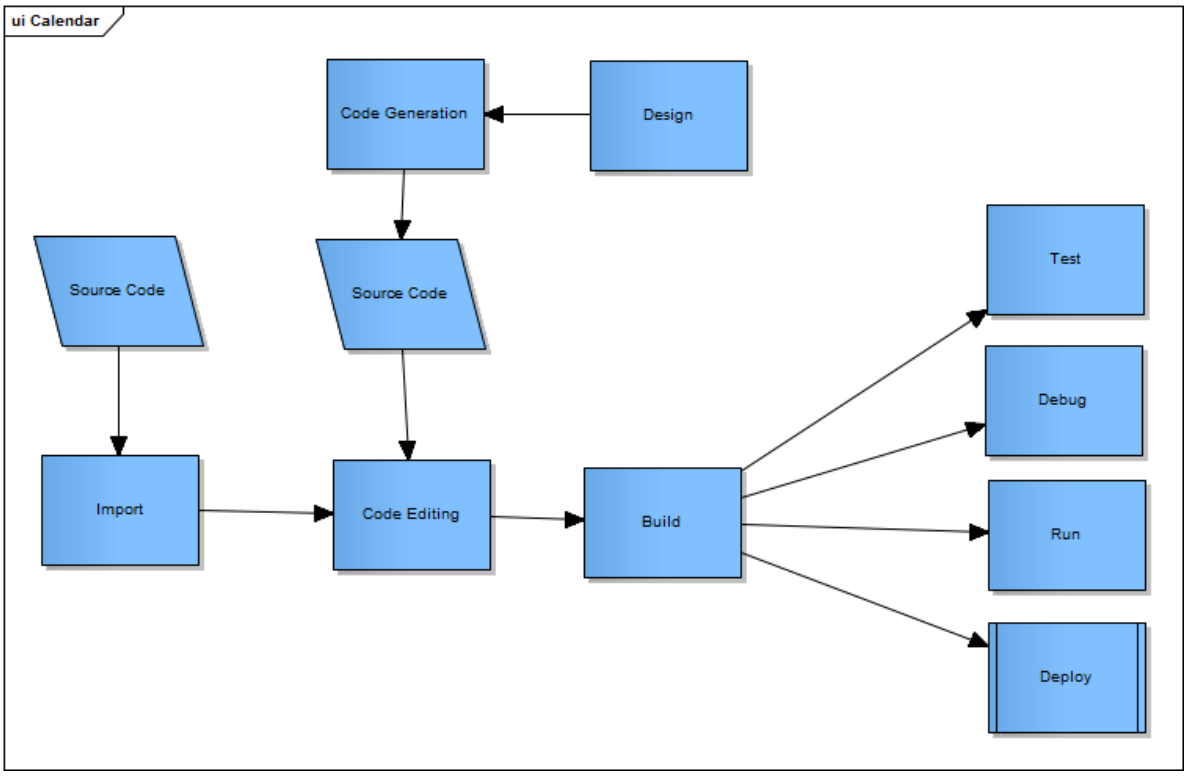
- Record sequence diagrams of application activities 57
- Capture State Transitions for a particular State Machine 75
- Capture Stacktraces at points in execution 37
- Profiling tool to sample application activity 82
- Object Workbench 87
- Unit Testing 80


Test
BookDB
«struct»
Book
PriceTotaller
Main()
BookDB()
AddBooks(BookDB)
loop 1 (4)
AddBook(string, string, decimal, bool)
Book(string, string, decimal, bool)
ProcessPaperbackBooks(ProcessBookDelegate)
loop 2 (3)
PrintTitle(Book)
Print(string)
PriceTotaller()
ProcessPaperbackBooks(ProcessBookDelegate)
loop 3 (3)
AddBookToTotal(Book)
AveragePrice() :decimal

3 Model Driven Development Environment

The Model Driven Development Environment (MDDE) provides tools to design, build and debug an application.

The MDDE integrates code and model by providing options to either generate source code from the model or reverse engineer existing source code into a model. Source code and model can be synchronized in either direction.



The MDDE provides development environments for popular languages including:

- C++
- C
- Java
- Microsoft .NET family
- ADA
- Python
- Perl

Toolboxes provide for different modeling technologies.

Note:

Although you can generate and reverse engineer code in a range of languages, Execution Analysis debugging and recording are supported for the following platforms / languages only:

- Microsoft Windows Native C
- Microsoft Windows Native C++
- Microsoft Windows Visual Basic
- Microsoft .NET Family (C#, J#, VB)
- Sun Microsystems Java.

To use the MDDE, work through the following sections:

3.1 Getting Started With The MDDE

To quickly start development in the Model Driven Development Environment, check through the following topics:

3.1.1 Prerequisites

Before using the Model Driven Development Environment:

- You should be using the correct edition: Enterprise Architect Professional, Corporate or Suite Editions.
- You should be connected to the required model.
- Relevant source code should be imported into the model.
- Basic Setup 8 should be complete.

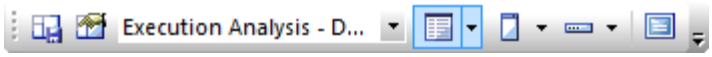
3.1.2 Available Tools

This section describes the tools available in the Model Driven Development Environment:

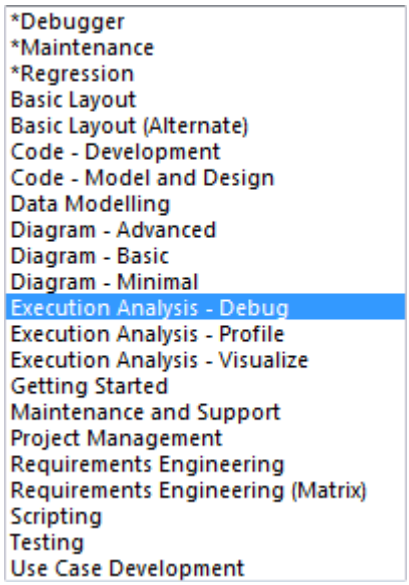
3.1.3 Workspace Layout

You can choose from many predefined workspace layouts (see *Using Enterprise Architect - UML Modeling Tool*) depending on the tasks you perform. When you are familiar with the environment and controls available to you, you can create your own.

Workspace Toolbar



Predefined Workspace Layouts



3.1.4 General Workflow

In working with the Model Driven Development Environment, you apply the following workflow as a circular process, refining as necessary in each iteration.

- Configure and set up scripts 9
- Model - Edit - Build - Debug - Test - Profile - Deploy - Document and Analyze.

3.2 Basic Setup

To use the execution tools of the Model Driven Development Environment - debugging, build and recording - it is necessary to record information about the application.This is achieved in Enterprise Architect through the use of Package Scripts.

A Package Script, when created, is naturally associated with the package that is currently selected.

A Package Script houses all the information the MDDE requires in order to provide support for tasks such as building the application, debugging and performing unit testing. A model can contain many Package Scripts. Each can build a separate application, or perhaps the same application but with different compilation options.

When you select a package or child Class in the Project Explorer, the Debug Management window displays any Package Scripts associated with that package. When you select a package root, the Debug Management window displays the scripts for the first package it finds under the root that has Scripts.

When you selected another package, the scripts displayed in the Debug Management window change also. You can force the scripts for a particular package to remain visible at all times by 'pinning 11' the package in the Debug Management window.

External Tools and Environment

If you plan on using any of the debugging features of the MDDE, you must have the appropriate Framework installed on your machine:

- The Java Runtime Environment for Java
- The .NET Framework for managed applications

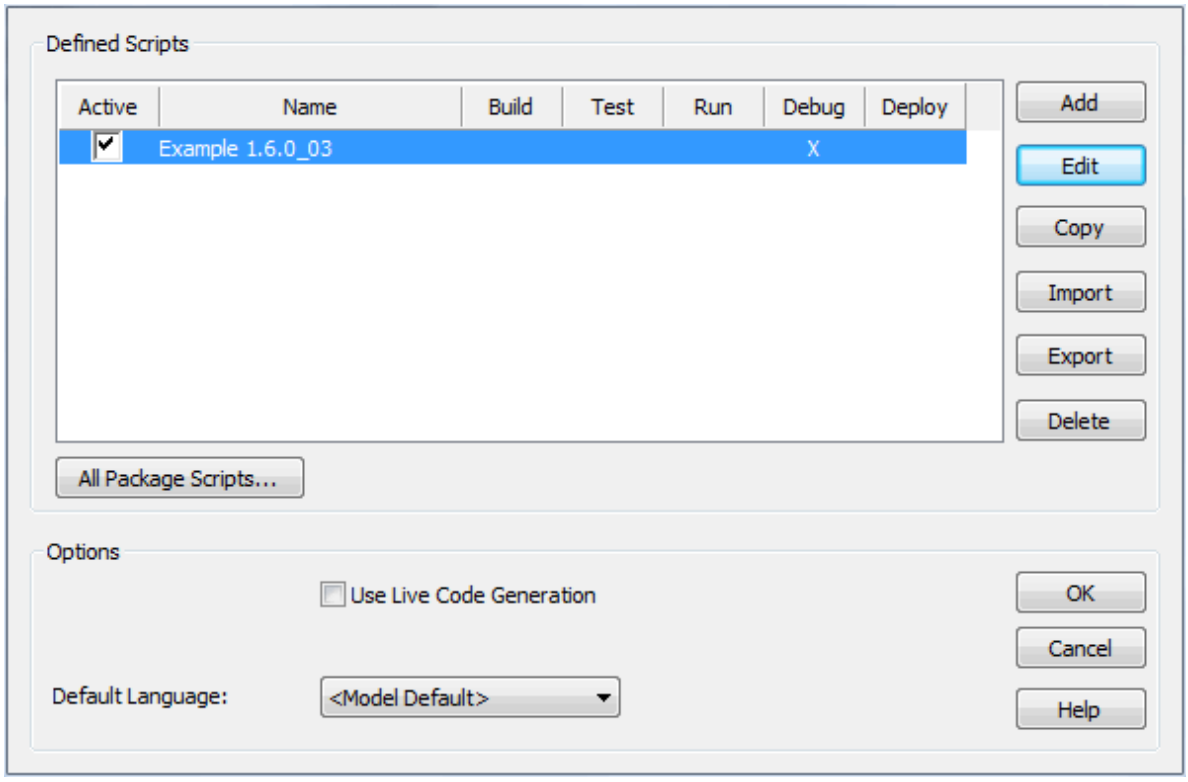
Any Operating System Environment Variables such as $PATH required by these kits should also be established.

3.2.1 Managing Scripts

In Enterprise Architect, any package within the UML Model can be configured to act as the 'root' of a source code project. By setting compilation scripts, xUnit commands, debuggers and other configuration settings for a package, all contained source code and elements can be built, tested or debugged according to the currently active configuration. Each package can have multiple scripts, but only one is active at any one time. The Package Build Scripts dialog enables you to create and manage those scripts.

To access the Package Build Scripts dialog, either:

- Press **[Shift]+[F12]**
- On the Debug toolbar, click on the drop-down arrow on the**Scripts** icon (the first icon on the left) and select the **Package Build Scripts** option
- Select the **Project | Execution Analyzer | Package Build Scripts** menu option, or
- Right-click on a package in the Project Browser, and select the **Execution Analyzer | Package Build Scripts** context menu option.



The Package Build Scripts dialog shows which script is active for the current package, and whether or not the script contains Build, Test, Run, Debug and Deploy components. The current package is as selected in the Project Browser; if a different package is selected, different scripts are available and different breakpoints and markers are applied.

Note that you must close the Package Build Scripts dialog to select a different package in the Project Browser. However, if the Debug window is open (**[Alt]+[8]**) you can see which debugging configuration is available and selected, and which breakpoints and markers are displayed, as you change packages in the Project Browser.

- To create a new script, click on the **Add** button; the Build Script dialog 10 displays.
- To modify an existing script, highlight the script name in the list and click on the **Edit** button.
- To copy a script with a new name, highlight the script name to copy and click on the **Copy** button; Enterprise Architect prompts you to enter a name for the new copy. Enter the new name in the dialog and click on the **OK** button. The new copy appears in the list and can be modified as usual.
- To delete a script, highlight the script name to delete, click on the **Delete** button, and click on the **OK** button.
- To export your scripts, click on the **Export** button to choose the scripts to export for this package.
- To import build scripts, click on the **Import** button to choose a .xml file of the scripts to import.

The **Default Language** field enables you to set the default language for generating source code for all new elements within this package and its descendents.

Select the **Use Live Code Generation** checkbox to update your source code instantly as you make changes to your model.

Click on the **All Package Scripts** button to open a new window that displays all scripts in the current project.

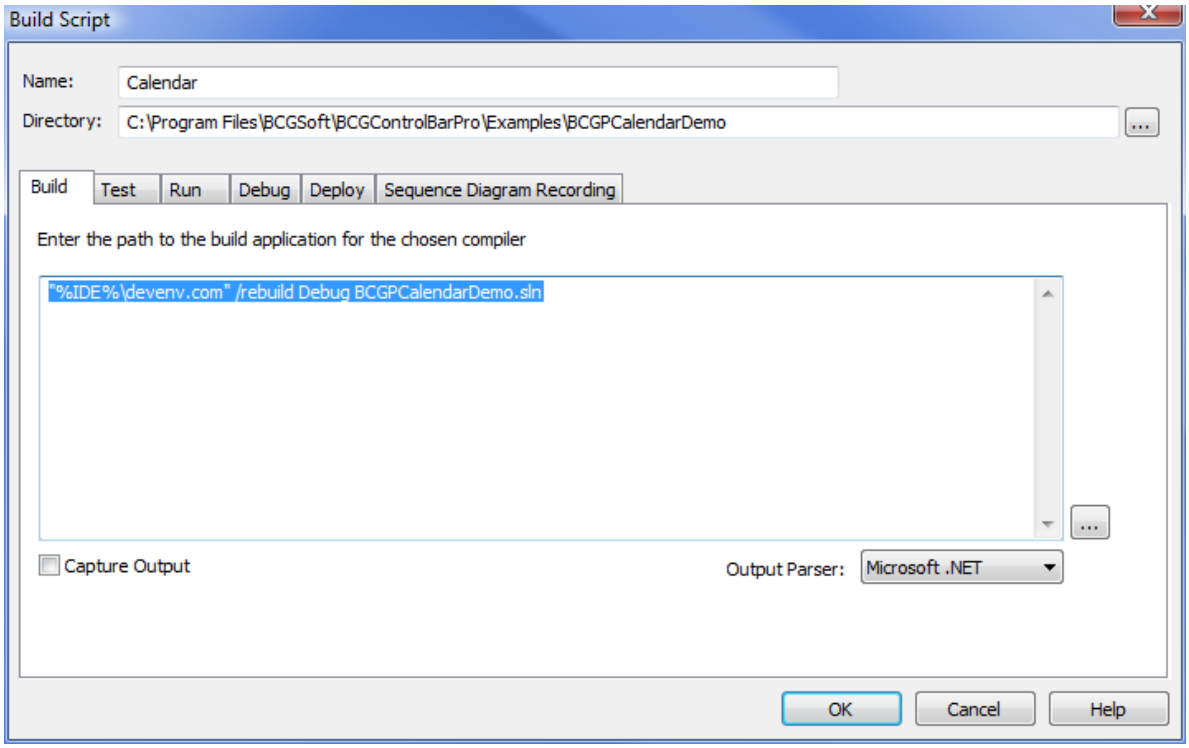
Once you have created new scripts or made changes to existing ones, click on the **OK** button to confirm the changes, otherwise click on the **Cancel** button to quit the Package Build Scripts dialog without saving any changes.

3.2.2 Defining Script Actions

Scripts are associated with a Package. When you create a Package Script you can define a number of actions.

If you plan to use any of the features of the Execution Analyzer, you must complete at least the Build and Debug tabs.

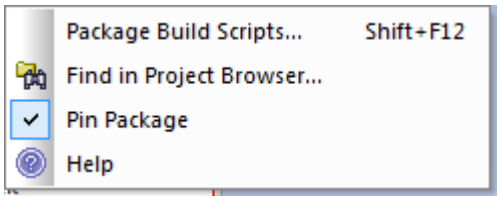
- Build 12
- Debug 15
- Test 53
- Run 55
- Deploy 56



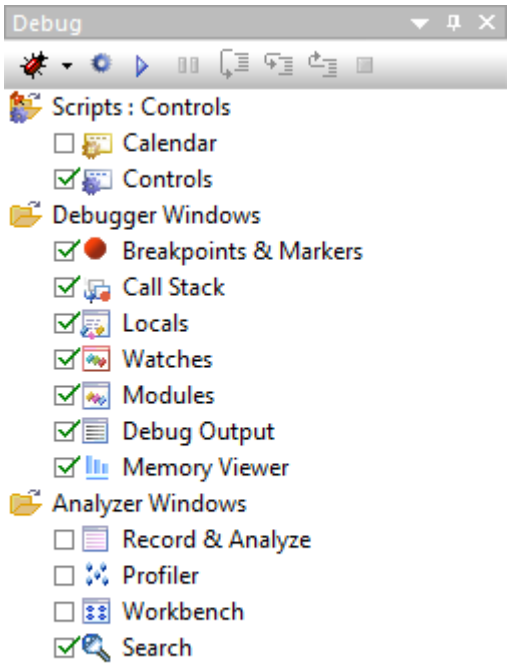
3.2.3 Setting the Default Script

Normally the target for any debugging session changes, tracking the package selected in the Project Explorer.

You can change this behaviour so that the scripts for a package remain selected in the Debug window. Use the context menu on the *Scripts* folder in the Debug window to either Pin or Unpin the currently-selected package.



When a package is pinned, the Debug window always displays the scripts defined for that package, and the debugger always uses the selected Package Script.



3.3 Code Generation and Synchronization - Safeguards

It is important that the model and source code are kept synchronized for the Visual Execution Analyzer to produce useful results.

Use the Code Generation tools to synchronize your model after any design changes or code editing (see *Code Engineering Using UML Models*).

Always build the application prior to any Execution Analysis session - debugging, recording or profiling.

3.4 Code Editing For MDDE

See the Code Editors topic in *Using Enterprise Architect - UML Modeling Tool*.

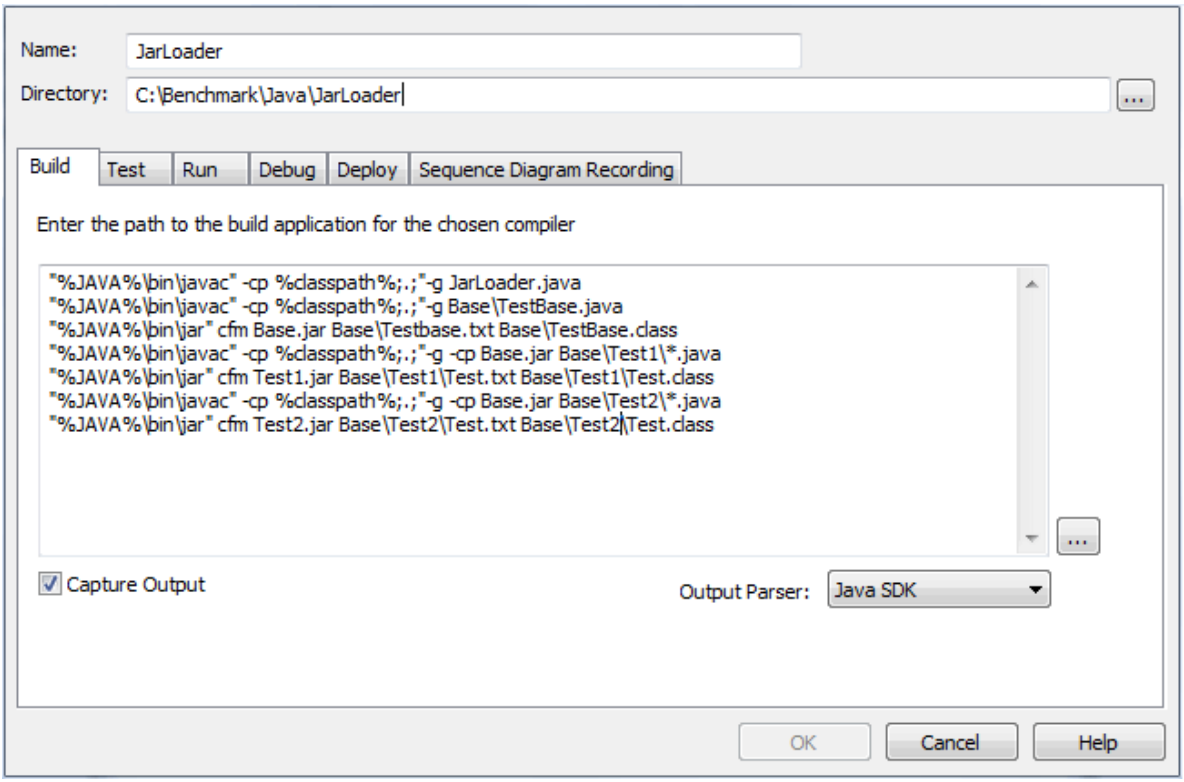
3.5 Build

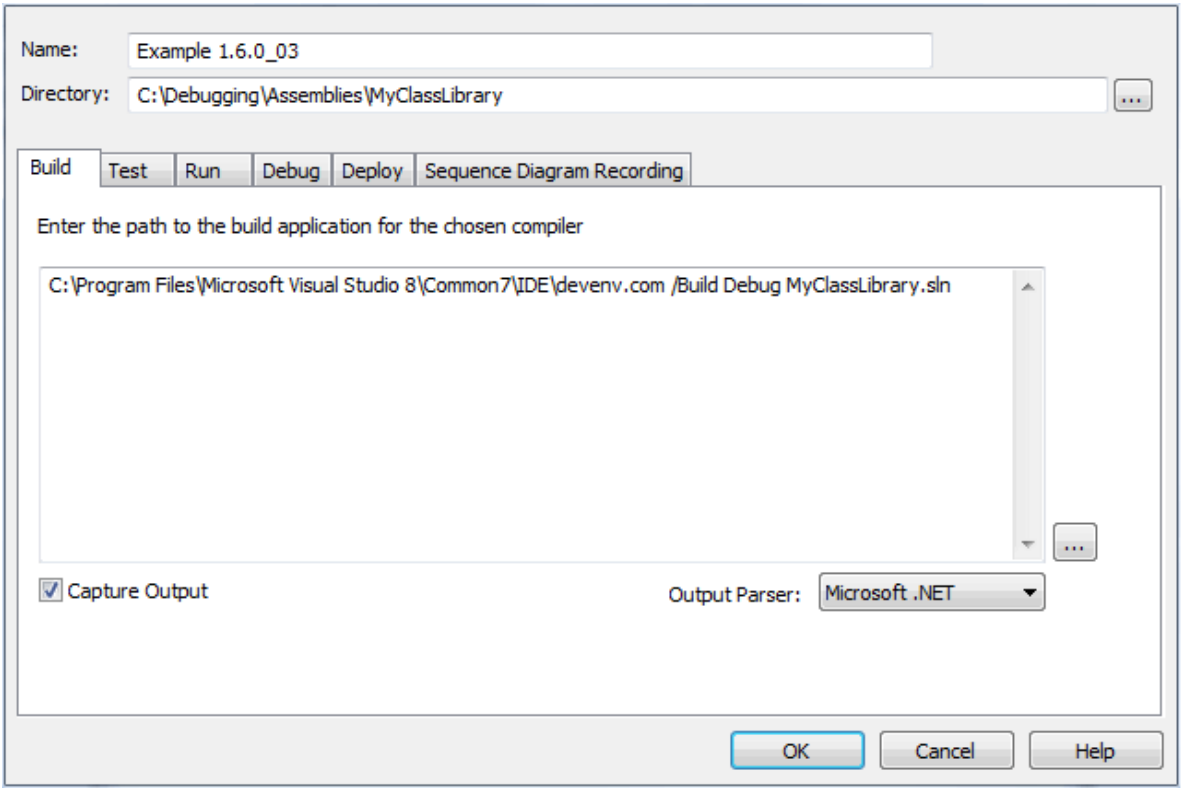
The topics in this section describe how you specify the commands to build the project / package:

- Add Comands 12
- Recursive Builds 14

3.5.1 Add Commands

The Build tab enables you to enter multiple commands for building the current package. These commands are executed when you select the **Project | Execution Analyzer | Build** menu option. The following examples are for Java and .NET respectively.





Write your script in the large text box using the standard *Windows Command Line* commands. You can specify, for example, compiler and linker options, and the names of output files. The format and content of this section depends on the actual compiler, make system, linker and so on that you use to build your project. You can also wrap up all these commands into a convenient batch file and call that here instead.

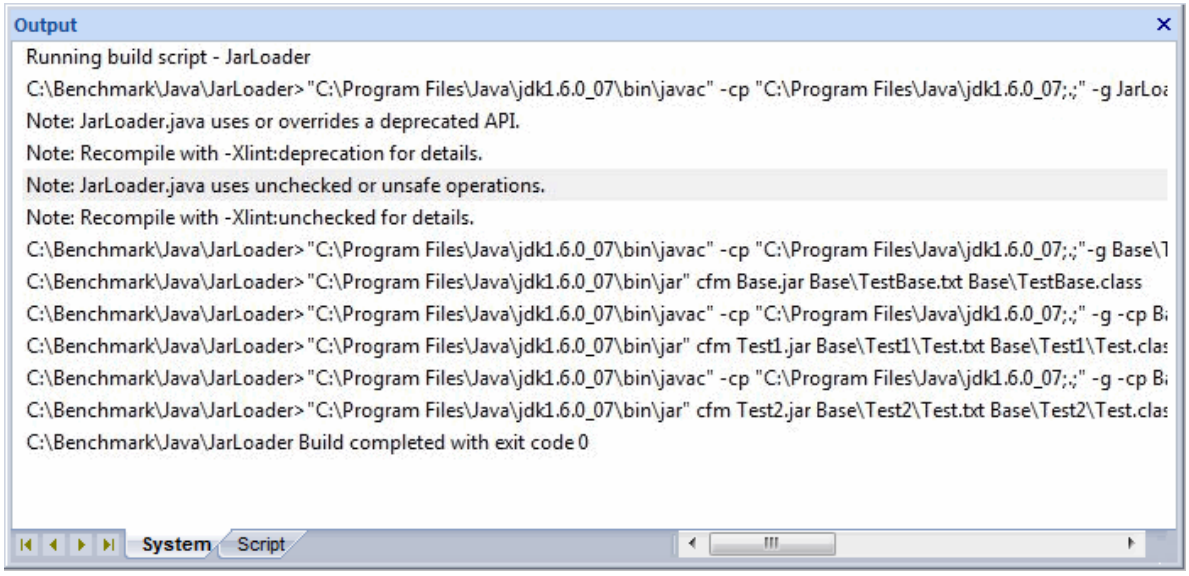
If you select the **Capture Output** checkbox, output from the script is logged in Enterprise Architect's Output window. This can be activated by selecting the **View | System Output** menu option.

The **Output Parser** field enables you to define a method for automatically parsing the compiler output. If you have selected the **Capture Output** checkbox, Enterprise Architect parses the output of the compiler so that by clicking on an error message in the Output window, you directly access the corresponding line of code.

Notes:

- The command listed in this field is executed as if from the command prompt. Therefore, if the executable path or any arguments contain spaces, they must be surrounded by quotes.
- Throughout this dialog, you can use Local Paths in specifying paths to executables; see the *Code Engineering Settings* section in *Code Engineering Using UML Models*.

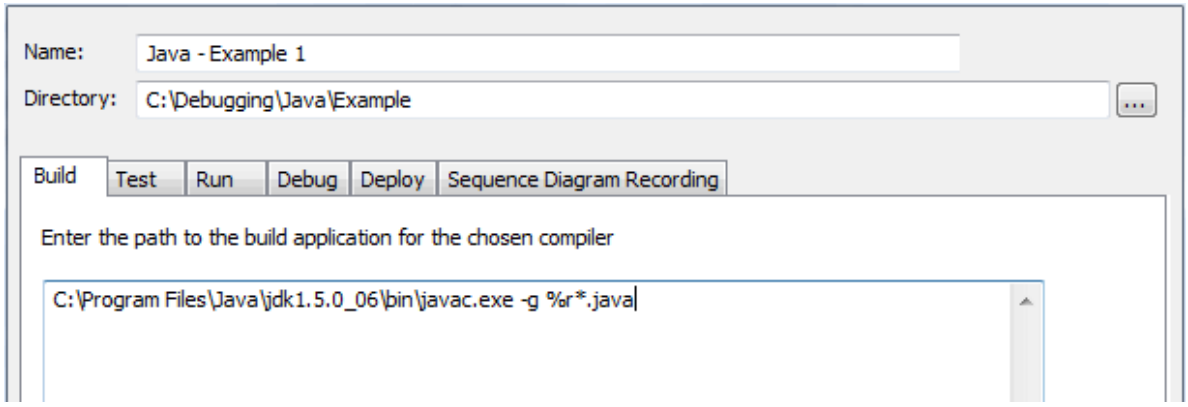
When you run the compile command inside Enterprise Architect, output from the compiler is piped back to the Output window and displayed as in the following illustration:



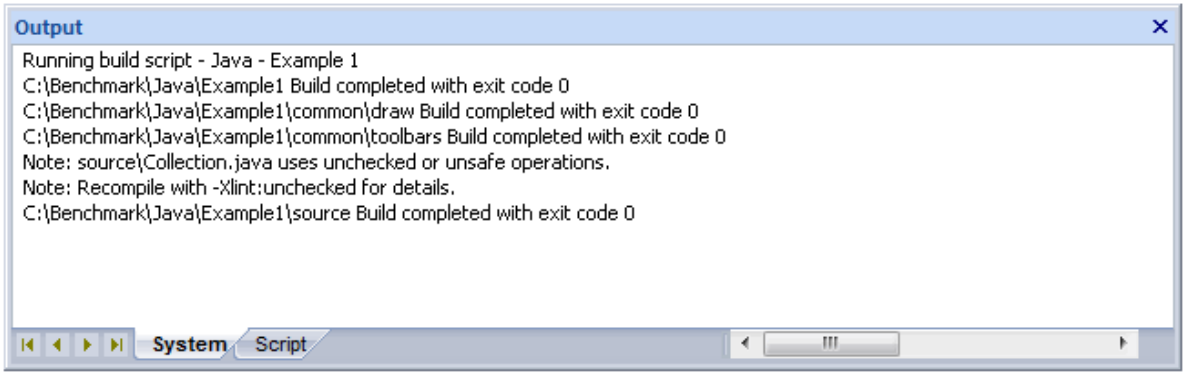
If you double-click on an error line, Enterprise Architect loads the appropriate source file and positions the cursor on the line where the error has been reported.

3.5.2 Recursive Builds

For any project you can apply the command entered in the build script to all sub folders of the initial directory by specifying the token *%r* immediately preceding the files to be built. The effect of this is Enterprise Architect iteratively replaces the token with any subpath found under the root and executes the command again.



The output from this Java example is shown below:



Note:

The path being built is displayed along with the exit code.

3.6 Debugging

This section describes how you define the debugging actions:

- How it works 15
- Setup for Debugging 15
- Breakpoint and Marker Management 37
- Debugging Actions 39
- Recording Actions 49

3.6.1 How it Works

The Model Driven Development Environment provides Debuggers for the following frameworks:

- Microsoft Native Code applications
- Microsoft .NET applications
- Java applications

To begin debugging:

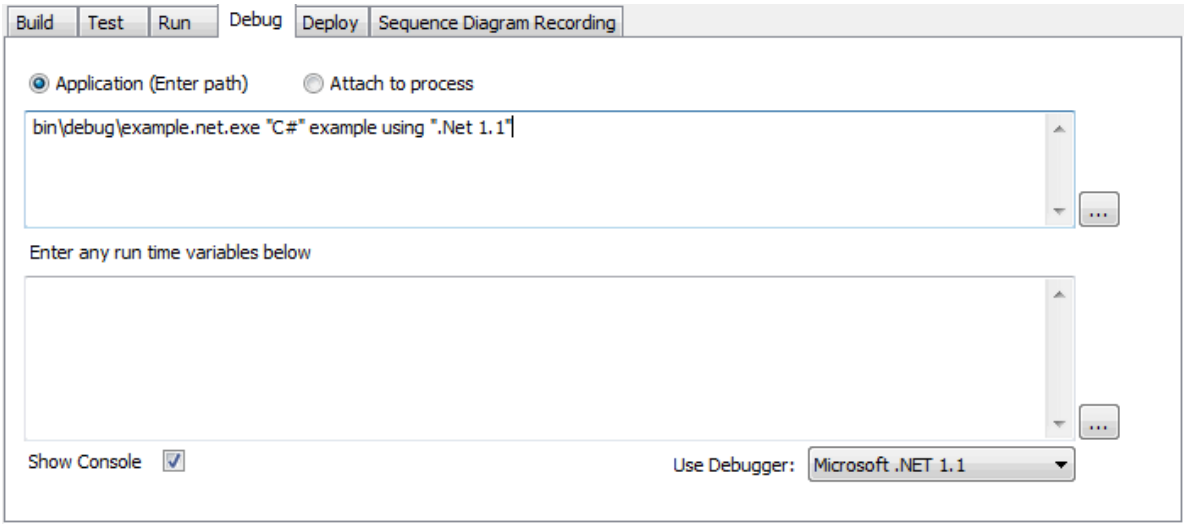
1. Ensure the model is open.
2. Ensure Basic Setup 8 has been completed for the Package or Project.
3. Ensure any source code for the areas of interest have been generated, or imported into the Model.
4. Ensure the application has been built. You can do this internally using the Build 12 Script or you can build the application externally. The important thing is that the application is built on the latest versions of the source.
5. Ensure that the model and source are synchronized (see *Code Engineering Using UML Models*).
6. Set breakpoints 38.
7. Start 39 the Debugger.

3.6.2 Setup for Debugging

To begin debugging you must specify

- The Debugger to use
- The Application path
- Runtime options, if applicable

The following example shows a .NET Debug script.



3.6.2.1 Operating System Specific Requirements

Important:

Please read the information provided in this topic.

Prerequisites

Creation of a Package Build Script and configuration of the **Debug** command in that script.

Supported Platforms

Enterprise Architect supports debugging on these platforms:

.Net

- Microsoft™ .NET Framework 1.1 and later
- Language support: C, C#, C++, J#, Visual Basic

Note:

Debugging under Windows Vista (x64) - If you encounter problems debugging with Enterprise Architect on a 64-bit platform, you should build a Win32 platform configuration in Visual Studio; that is, do not specify **ANY-CPU**, specify **WIN32**.

Java

- Java 2 Platform Standard edition (J2SE) version 5.0
- J2EE JDK 1.4 and above
- Requires previous installation of the Java Runtime Environment and Java Development Kit from Sun Microsystems™.

Debugging is implemented through the Java Virtual Machine Tools Interface (JVMTI), which is part of the Java Platform Debugger Architecture (JPDA). The JPDA is included in the J2SE SDK 1.3 and later.

Windows for Native Applications

Enterprise Architect supports debugging native code (C, C++ and Visual Basic) compiled with the Microsoft™ compiler where an associated PDB file is available. Select **Microsoft Native** from the list of debugging platforms in your package script.

You can import native code into your model, and record the execution history for any Classes and methods. You can also generate Sequence diagrams from the resulting execution path.

Note:

Enterprise Architect currently does not support remote debugging.

3.6.2.1.1 UAC-Enabled Operating Systems

The Microsoft operating systems *Windows Vista* and *Windows 7* provide User Account Control (UAC) to manage security for applications.

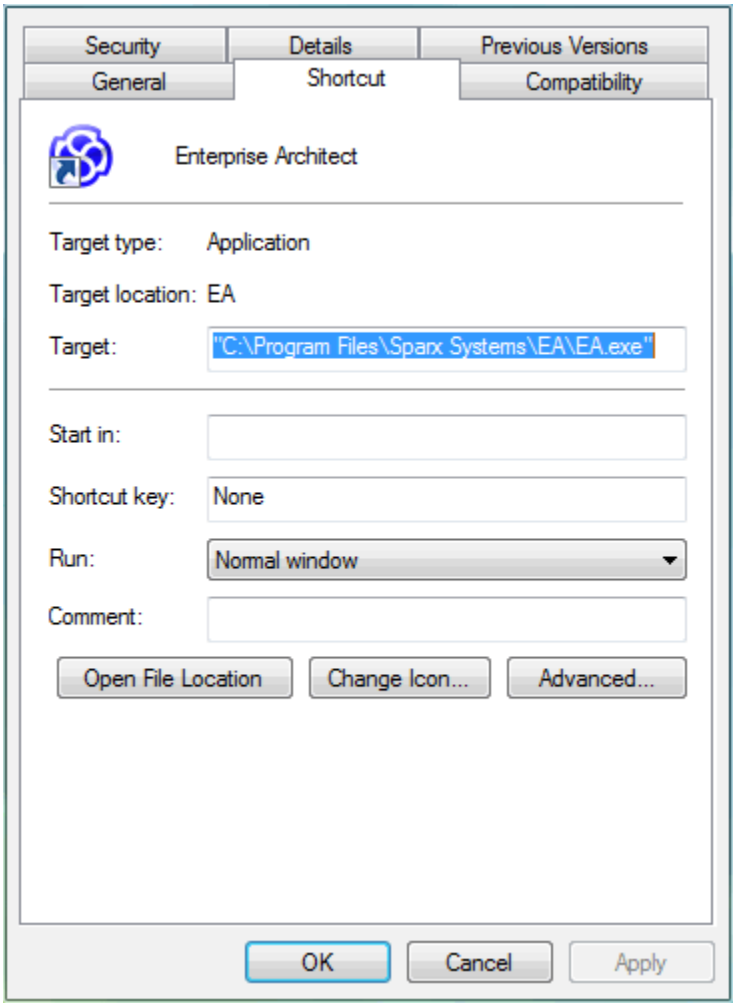
The Enterprise Architect Visual Execution Analyser is UAC-compliant, and users of UAC-enabled systems can perform operations with the Visual Execution Analyser and related facilities under accounts that are members of only the *Users* group.

However, when attaching to processes running as services on a UAC-enabled operating system, it might be necessary to log in as an Administrator. To do this, follow the step below:

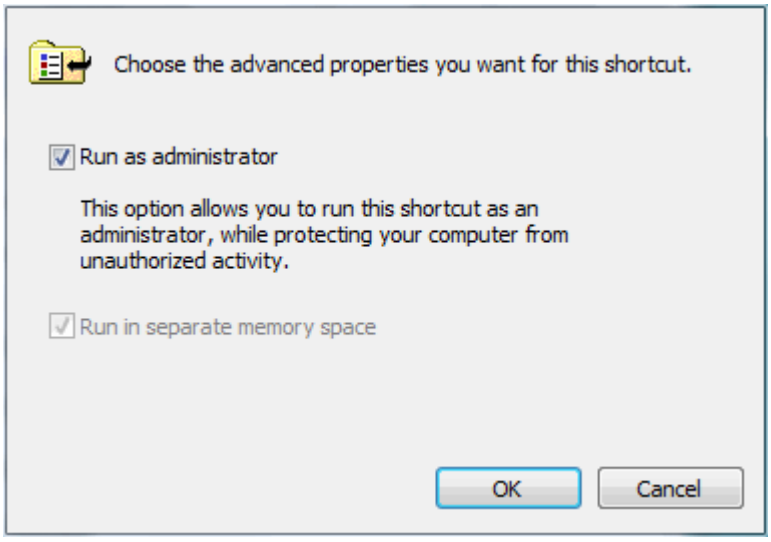
1. Before you run Enterprise Architect, right-click on the Enterprise Architect icon on the desktop and select the **Run as administrator** option.

Alternatively, edit or create a link to Enterprise Architect and configure the link to run as an administrator; follow the steps below:

1. Right-click on the Enterprise Architect icon and select the **Properties** menu option. The Enterprise Architect Properties dialog displays.



2. Click on the **Advanced** button. The Advanced Properties dialog displays.

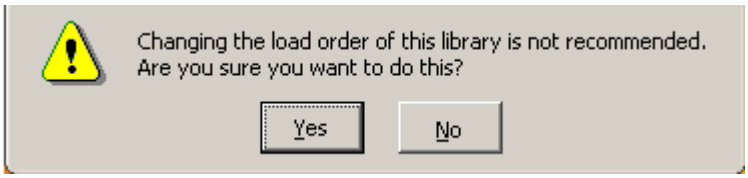


3. Select the **Run as administrator** checkbox.
4. Click on the **OK** button, and again on the Enterprise Architect Properties dialog.

3.6.2.1.2 WINE Debugging

At the command line, run $ winecfg.

Set the library overrides for dbghelp to (*native, builtin*), and accept the warning about overriding this DLL:



Note:

If WINE crashes, the back traces may not be correct.

1. Set *dbghelp* to **native** by using *winecfg*.
2. Copy the application source code plus executable(s) to your bottle. (The path must be the same as the compiled version; that is:

   If Windows source = C:\Source\SampleApp, under Crossover it must be C:\Source\SampleApp.)
3. Copy any Side-By-Side assemblies that are used by the application.
4. Import the source code into Enterprise Architect. (Optional.)
5. Create a build script 12 on a package.

   Set the path of the application on the Debug tab, and set **Use Debugger** to **Microsoft Native**.
6. Open the Profiler 82 (**View | Execution Analyzer | Profiler**).
7. Click on the **Launch** button (first button on the Profiler window).

   If the sample didn't start, click on the **Sampling** button (third button on the Profiler window).
8. Once you have finished profiling, shut down the application (not Enterprise Architect).
9. View the Sampler report by clicking the **View Report** button (fifth button on the Profiler window).

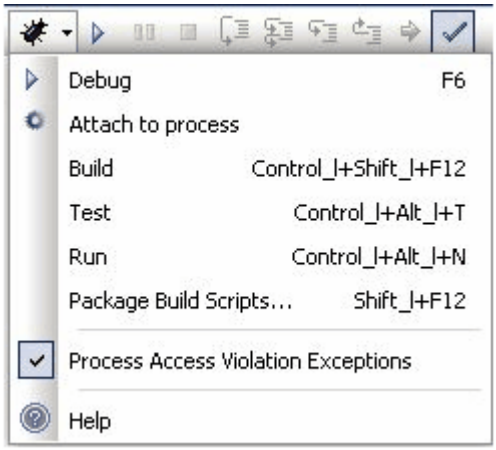
Tips:

- If you are using MFC remember to copy the debug side-by-side assemblies to the *C:\window\winsxs* directory.
- To add a windows path to WINE, modify the Registry entry:

  HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Environment

Access Violation Exceptions

Due to the manner in which WINE handles direct drawing and access to DIB data, an additional option is provided on the Debug window toolbar drop-down menu to ignore or process access violation exceptions thrown when your program directly accesses DIB data.



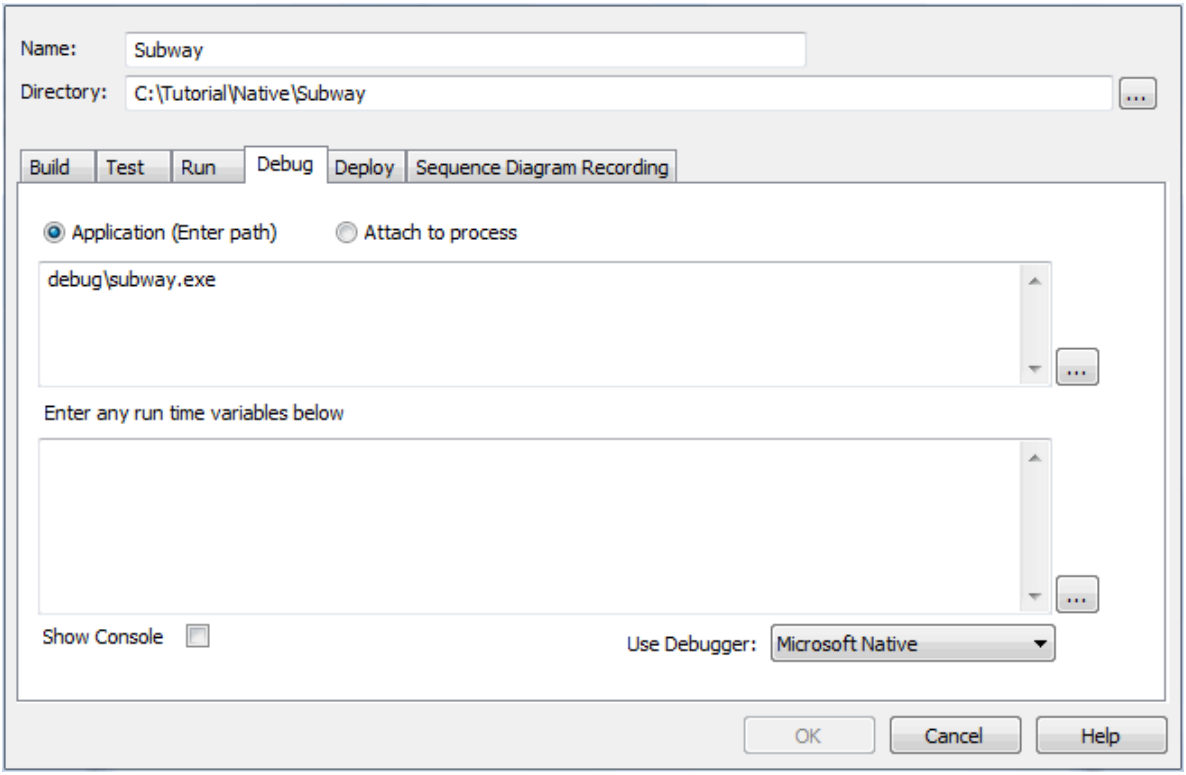
Select this option to catch genuine (unexpected) access violations; deselect it to ignore expected violations. As the debugger cannot distinguish between expected and unexpected violations, you might have to use trial and error to capture and inspect genuine program crashes.

3.6.2.2 Microsoft C++ and Native (C, VB)

The example script below is configured to enable debugging of a **C++** project built in Microsoft Visual Studio 2005 or 2008.

You can debug native code only if there is a corresponding PDB file for the executable. You normally create the PDB file as a result of building the application.

The build should include full debug information and there should be no optimizations set.



The script must specify two things to support debugging:

- The path to the executable
- **Microsoft Native** as the debugging platform.

3.6.2.2.1 Debug Symbols

For Applications built using *Microsoft Platform SDK* Debug Symbols are written to an Application PDB file when the Application is built.

The *Debugging Tools for Windows*, an API used by the Visual Execution Debugger, uses these symbols to present meaningful information to Execution Analyzer controls.

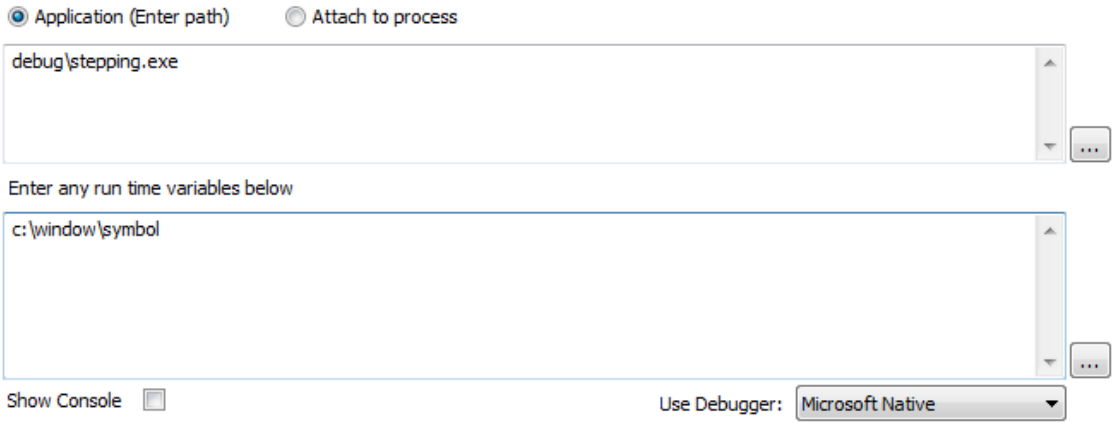
These symbols can easily get out of date and cause errant behaviour. The debugger might highlight the wrong line of code in the editor whilst at a breakpoint. It is therefore best to ensure the application is built prior to any debugging or recording session.

The debugger must inform the API how to reconcile addresses in the image being debugged. It does this by specifying a number of paths to the API that tell it where to look for PDB files. The API automatically picks up the path to the main image PDB from the image itself.

For system DLLs (kernel32, mfc90ud ...) for which no debug symbols are found, the Call Stack shows some frames with module names and addresses only .

You can supplement the symbols translated by passing additional paths to the API. To do this there must be a Package Script selected and it must have the Native debugger specified.

You pass additional symbol paths in a semi-colon separated list in the **Enter any runtime variables...** field of the Debug tab, as illustrated below.

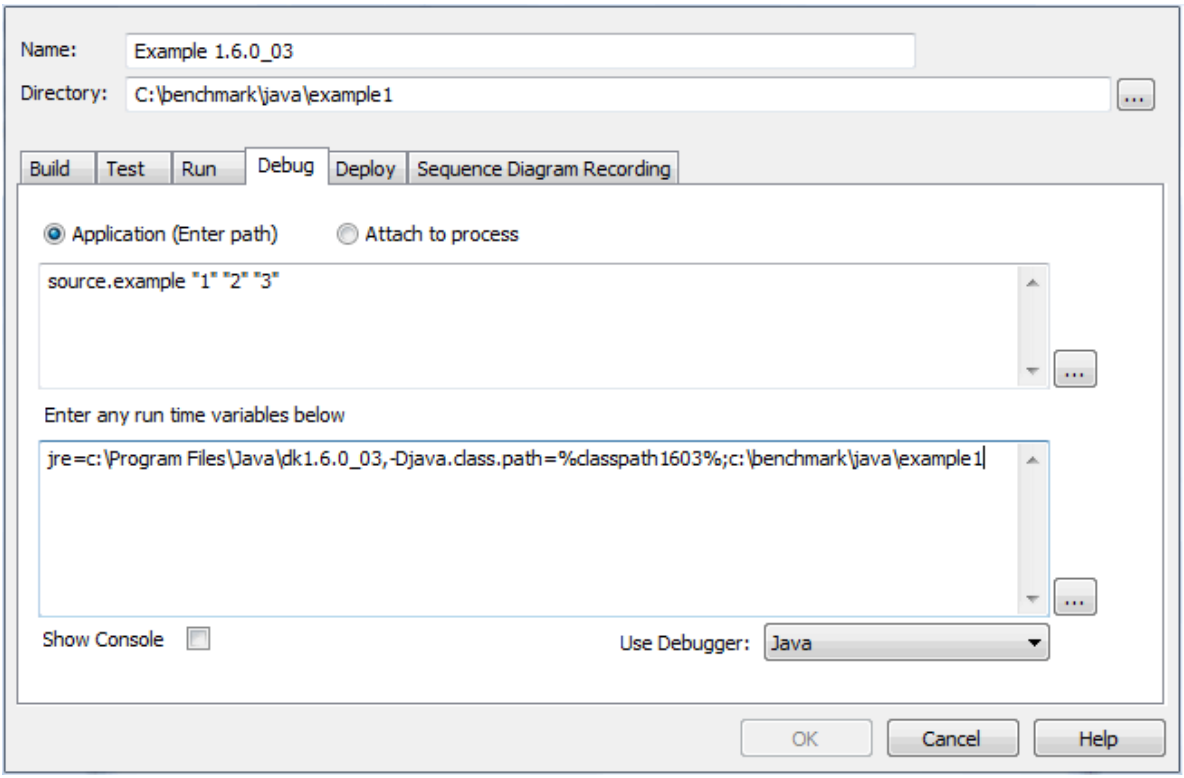


3.6.2.3 Java

This section describes how to configure Enterprise Architect for debugging Java applications and Web Servers.

3.6.2.3.1 General Setup for Java

This is the general setup for debugging Java applications:



Option	Use to
Application (Enter path)	Identify the fully qualified Class name to debug, followed by any arguments. The Class must have a method declared with the following signature: public static void main(String[]); The debugger calls this method on the Class you name. In the example above, the parameters **1**, **2** and **3** are passed to the method. You can also debug a Java application by attaching to an existing Java process 22.
Enter any run time variables below	Type any required command line options to the Java Virtual Machine. You also must provide a parameter (**jre**) that is a path to be searched for the jvm.dll. This is the DLL supplied as part of the Java runtime environment or Java JDK from Sun Microsystems™ (see Debugging 57). In the example above, a virtual machine is created with a new Class path property that comprises any paths named in the environment variable *classpath 1603* plus the single path "C:\benchmark\java\example1". If no Class path is specified, the debugger always creates the virtual machine with a Class path property equal to any path contained in the environment variable plus the path entered in the default working directory of this script. **Note:** If source files and .class files are located under different directory trees, the Class path property MUST include both root path(s) to the source and root path(s) to binary class files.
Show Console	Create a console window for Java. If no console window is required, leave blank.
Use Debugger	Select **Java**.

3.6.2.3.2 Advanced Techniques

In addition to the standard Java debugging techniques, you can also:

- Attach to a Virtual Machine 22
- Debug Internet Browser Java Applets. 23

You can debug a Java application by attaching to an existing Java process.

However, the Java process requires a specific startup option specifying the Sparx Systems Java Agent. The format of the command line option is:

```
-agentlib:SSJavaProfiler80
```

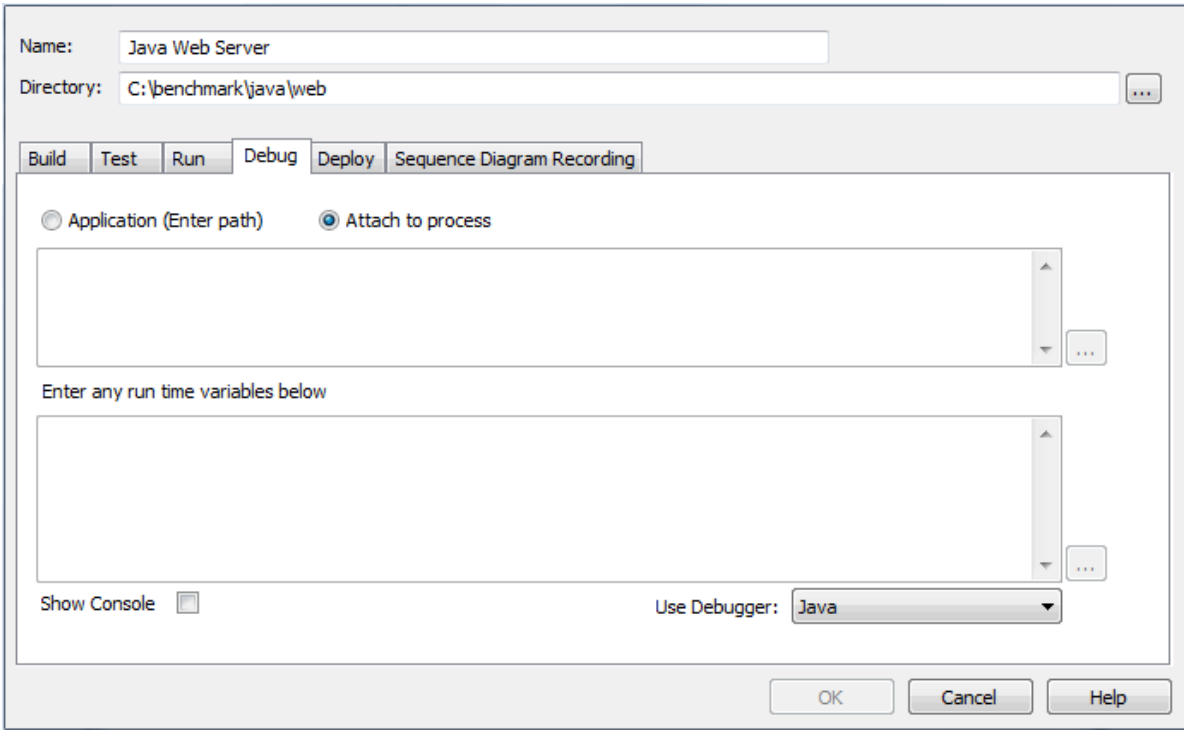
or:

```
-agentpath:"c:\program files\sparx systems\ea\SSJavaProfiler80"
```

The example below is for attaching to the *Tomcat Webserver*. Select the **Attach to process** radio button, and then the keyword **Attach** is all that you have to enter. This keyword causes the debugger to prompt you for a process at runtime.

Note:

The **Show Console** checkbox has no effect when attaching to an existing virtual machine.

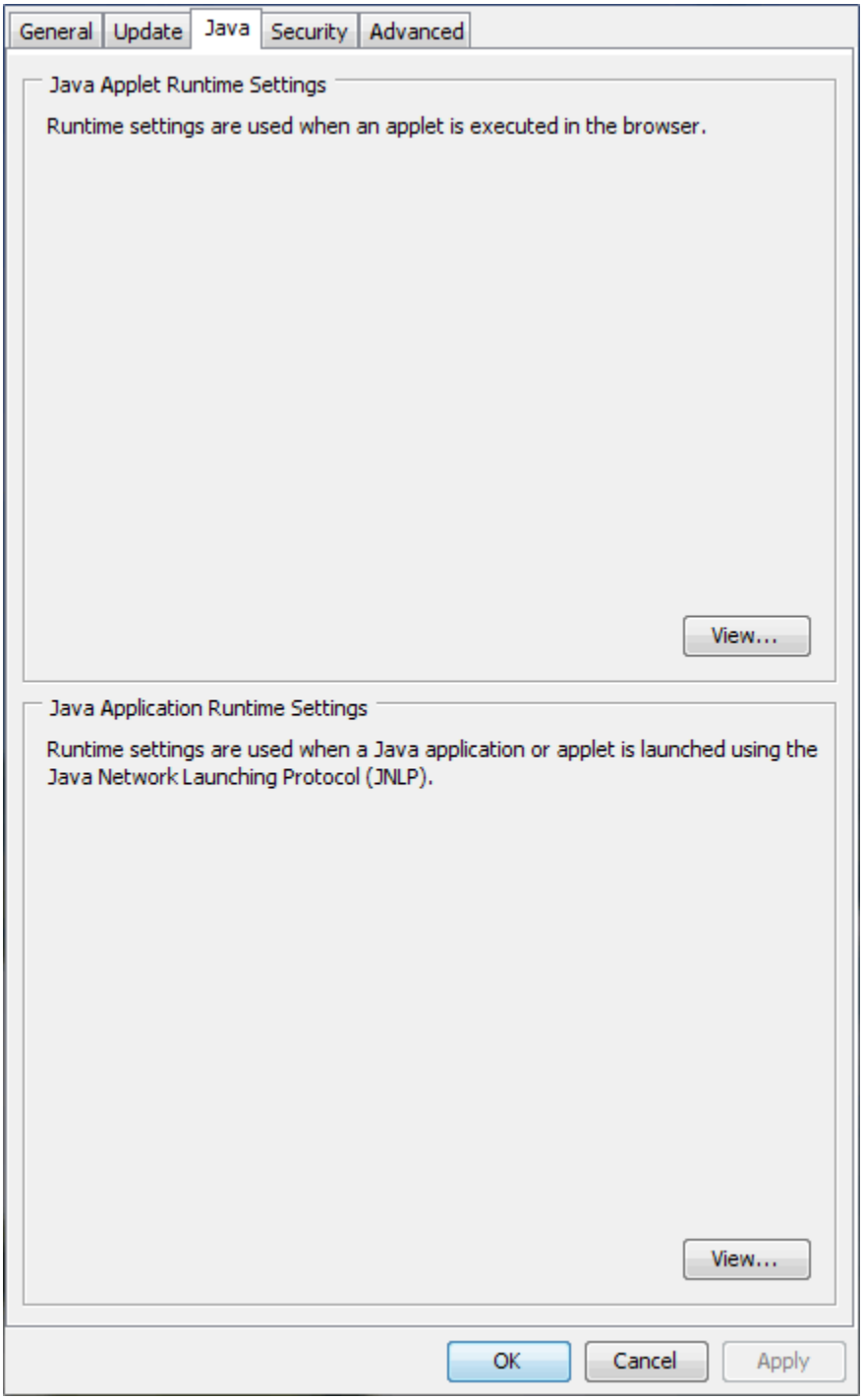


No run time variables are necessary when attaching as these are specified as startup parameters to the process.

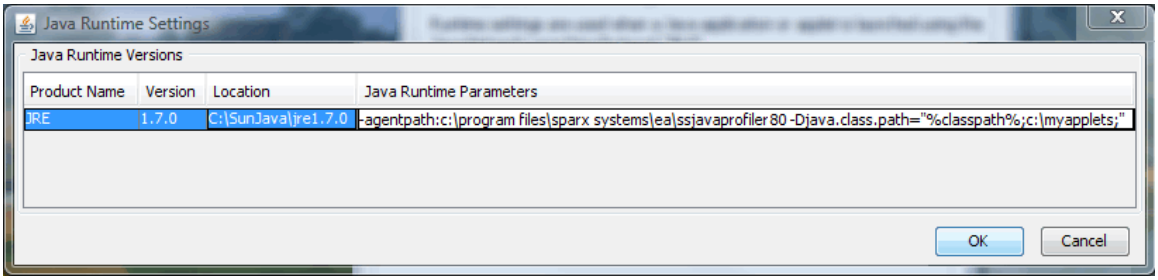
This topic describes the configuration requirements and procedure for debugging Java Applets running in a browser from Enterprise Architect.

The procedure requires you to attach to the browser process hosting the Java Virtual Machine (JVM) from Enterprise Architect, as summarized below:

1. Ensure binaries for the applet code to be debugged have been built with debug information.
2. Configure the JVM using the Java Control Panel.



3. In the Java Applet Runtime Settings panel, click on the **View** button. The Java Runtime Settings dialog displays.



4. Click on the appropriate entry and click on the **OK** button to load the Sparx Systems Agent.
5. Import source code into the Enterprise Architect model, or synchronize existing code. (See *Code Engineering Using UML Models*.)
6. Create or modify the Package Build Script 8 to specify the option for attaching to the process.
5. Set breakpoints 37.
6. Launch the browser.
7. Attach to the browser process from Enterprise Architect.

Note that the *class.path* property specified for the JVM includes the root path to the applet source files. This is necessary for the Enterprise Architect debugger to match the execution to the imported source in the model.

3.6.2.3.3 Working with Java Web Servers

This topic describes the configuration requirements and procedure for debugging Java web servers such as JBOSS 27 and Apache Tomcat (both Server 28 configuration and Windows Service 29 configuration) in Enterprise Architect.

The procedure involves attaching to the process hosting the Java Virtual Machine from Enterprise Architect, as summarized below:

1. Ensure binaries for the web server code to be debugged have been built with debug information.
2. Launch the server with the Virtual Machine startup option 25 described in this topic.
3. Import source code into the Enterprise Architect Model, or synchronize existing code.
4. Create or modify the Package Build Script 25 to specify the **Debug** option for attaching to the process.
5. Set breakpoints. 37
6. Launch the client.
7. Attach to the process from Enterprise Architect.

Server Configuration

The configuration necessary for the web servers to interact with Enterprise Architect must address the following two essential points:

- Any VM to be debugged, created or hosted by the server must have the Sparx Systems Agent SSJavaProfiler80 command line option specified in the VM startup option (that is: -agentlib:SSJavaProfiler80)
- The CLASSPATH, however it is passed to the VM, must specify the root path to the package source files.

The Enterprise Architect debugger uses the java.class.path property in the VM being debugged, to locate the source file corresponding to a breakpoint occurring in a Class during execution. For example, a Class to be debugged is called:

a.b.C

This is located in physical directory:

C:\source\a\b

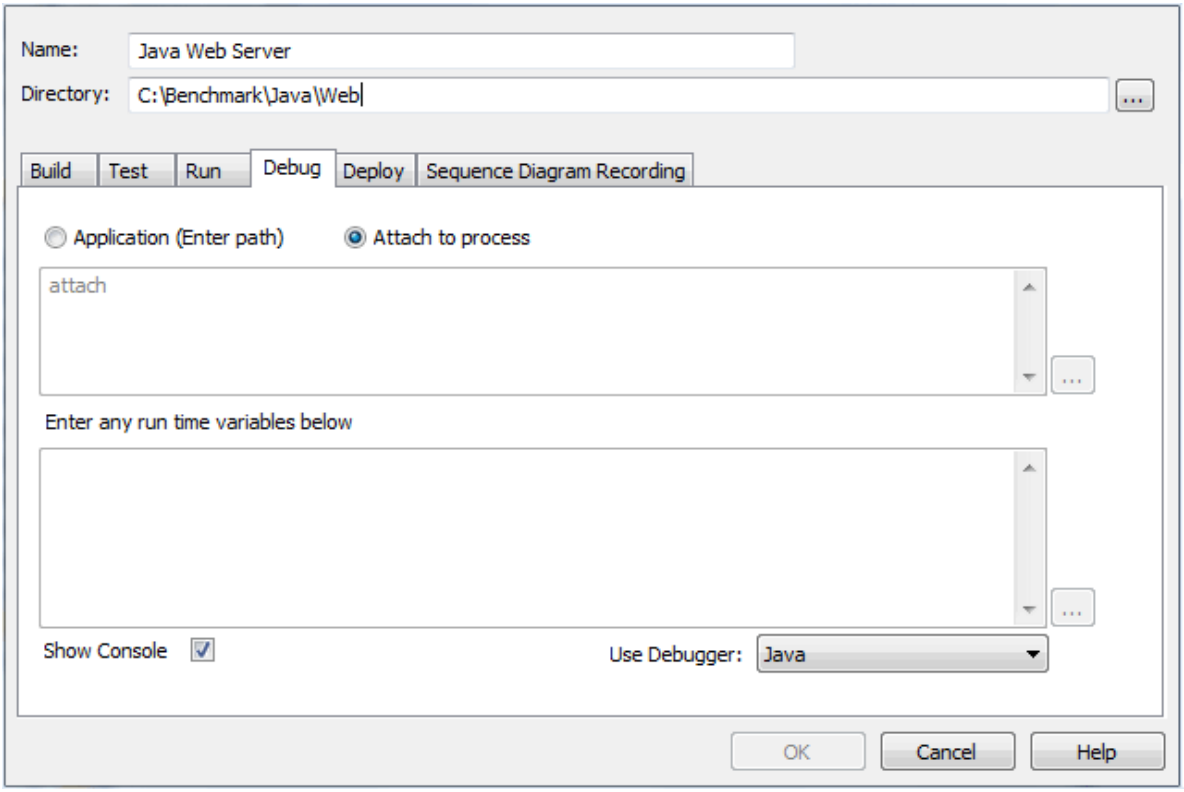
So, for debugging to be successful, the CLASSPATH must contain the root path:

c:\source.

Package Script Configuration

Using the Debug tab 21 of the Build Script 2 dialog, create a script for the code you have imported and specify the following:

- Select the **Attach to process** radio button, and in the field below type **attach**.
- In the **Use Debugger** field, click on the drop-down arrow and select **Java**.



All other fields are unimportant. The **Directory** field is normally used in the absence of any Class path property.

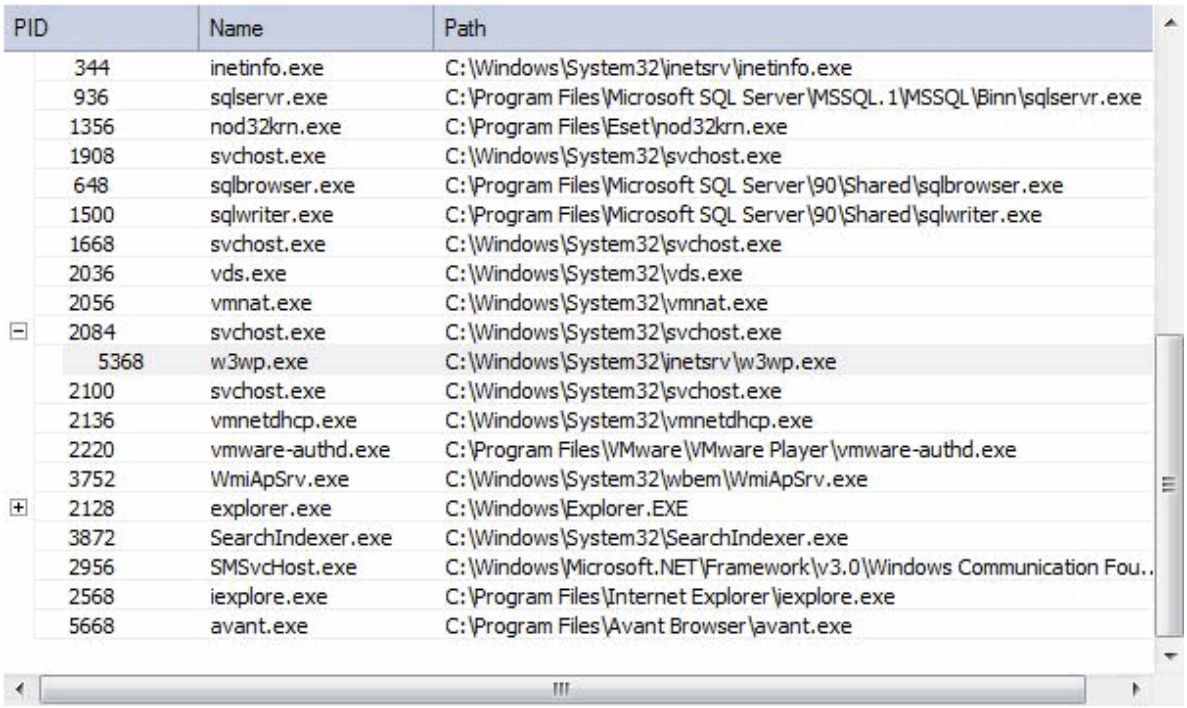
Debugging

First ensure that the server is running, and that the server process has loaded the Sparx Systems Agent DLL SSJavaProfiler80.DLL (use *Process Explorer* or similar tools to prove this).

Launch the client and ensure the client executes. This must be done before attaching to the server process in Enterprise Architect.

After the client has been executed at least once, return to Enterprise Architect, open the source code you imported and set some breakpoints 37.

Click on the **Run Debugger** 15 button in Enterprise Architect. The Attach To Process dialog displays.

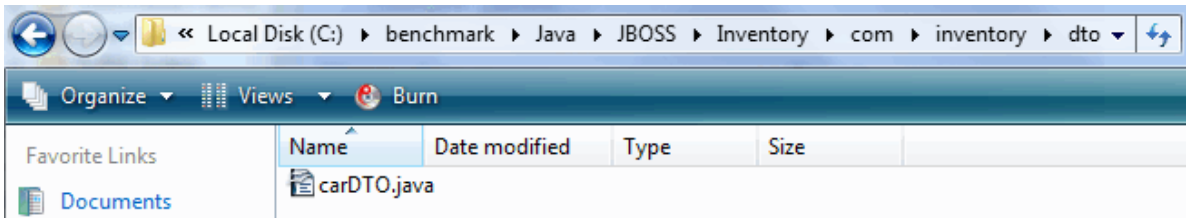


PID	Name	Path
344	inetinfo.exe	C:\Windows\System32\inetsrv\inetinfo.exe
936	sqlservr.exe	C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Binn\sqlservr.exe
1356	nod32krn.exe	C:\Program Files\Eset\nod32krn.exe
1908	svchost.exe	C:\Windows\System32\svchost.exe
648	sqlbrowser.exe	C:\Program Files\Microsoft SQL Server\90\Shared\sqlbrowser.exe
1500	sqlwriter.exe	C:\Program Files\Microsoft SQL Server\90\Shared\sqlwriter.exe
1668	svchost.exe	C:\Windows\System32\svchost.exe
2036	vds.exe	C:\Windows\System32\vds.exe
2056	vmnat.exe	C:\Windows\System32\vmnat.exe
2084	svchost.exe	C:\Windows\System32\svchost.exe
5368	w3wp.exe	C:\Windows\System32\inetsrv\w3wp.exe
2100	svchost.exe	C:\Windows\System32\svchost.exe
2136	vmnetdhcp.exe	C:\Windows\System32\vmnetdhcp.exe
2220	vmware-authd.exe	C:\Program Files\VMware\VMware Player\vmware-authd.exe
3752	WmiApSrv.exe	C:\Windows\System32\wbem\WmiApSrv.exe
2128	explorer.exe	C:\Windows\Explorer.EXE
3872	SearchIndexer.exe	C:\Windows\System32\SearchIndexer.exe
2956	SMSvcHost.exe	C:\Windows\Microsoft.NET\Framework\v3.0\Windows Communication Fou..
2568	iexplore.exe	C:\Program Files\Internet Explorer\iexplore.exe
5668	avant.exe	C:\Program Files\Avant Browser\avant.exe

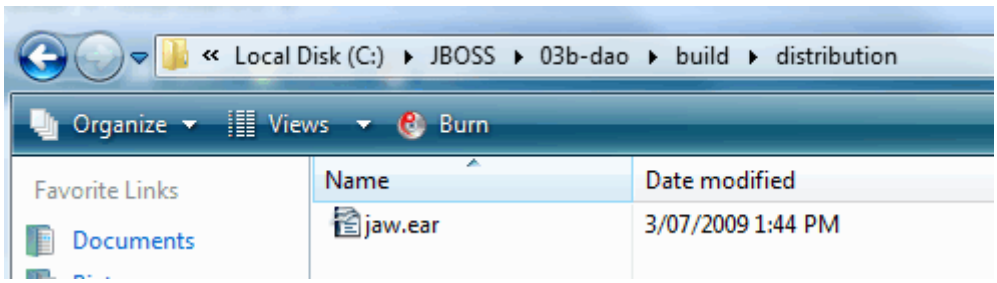
Click on the **OK** button. A confirmation message displays in the Debug Output window, stating that the process has been attached.

The breakpoints should remain enabled (bright red). If the breakpoints fail, and contain either an exclamation mark or a question mark, then either the process is not hosting the SSJavaprofiler80 Agent or the binaries being executed by the server are not based on the source code. If so, check your configuration.

Consider the JBoss example below. The source code for a simple servlet is located in the directory location:



The binaries executed by JBOSS are located in the JAW.EAR file in this location:



The Enterprise Architect debugger has to be able to locate source files during debugging. To do this it also uses the CLASSPATH, searching in any listed path for a matching JAVA source file, so the CLASSPATH must include a path to the root of the package for Enterprise Architect to find the source during debugging.

The following is an excerpt from the command file that executes the JBOSS server. Since the Class to be

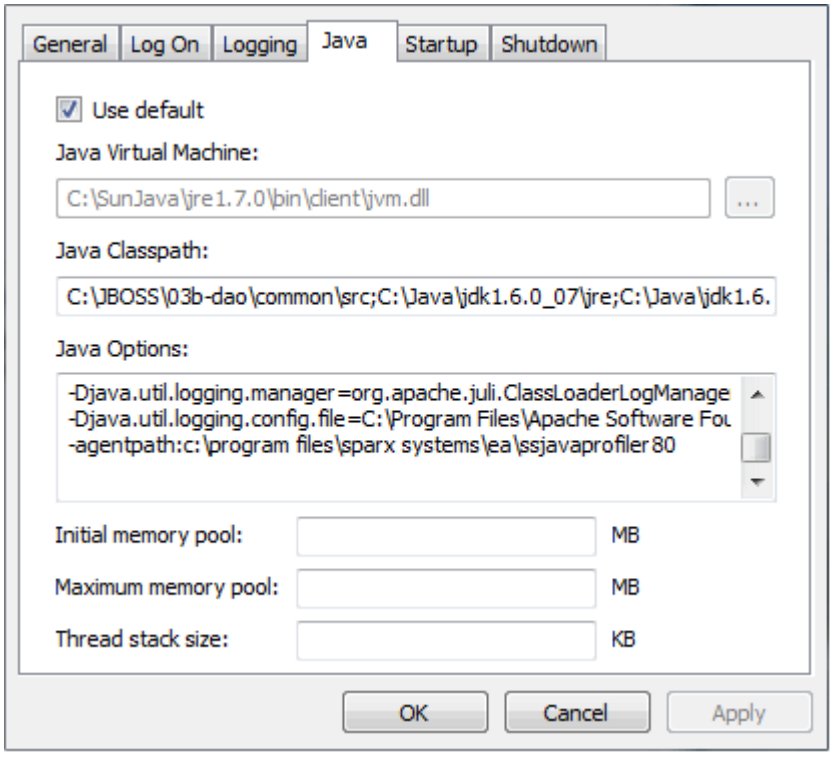
debugged is at com/inventory/dto/carDTO, the root of this path is included in the *JBOSS* classpath.

```
RUN.BAT
-------------
set SOURCE=C:\Benchmark\Java\JBOSS\Inventory

set JAVAC_JAR=%JAVA_HOME%\lib\tools.jar

if  "%JBOSS_CLASSPATH%" == ""
(
      set JBOSS_CLASSPATH=%SOURCE%;%JAVAC_JAR%;%RUNJAR%;
)
else
(
      set JBOSS_CLASSPATH=%SOURCE%;%JBOSS_CLASSPATH%;%JAVAC_JAR%;%RUNJAR%;
)

set JAVA_OPTS=%JAVA_OPTS% -agentpath:"c:\program files\sparx systems\ssjavaprofiler80"
```

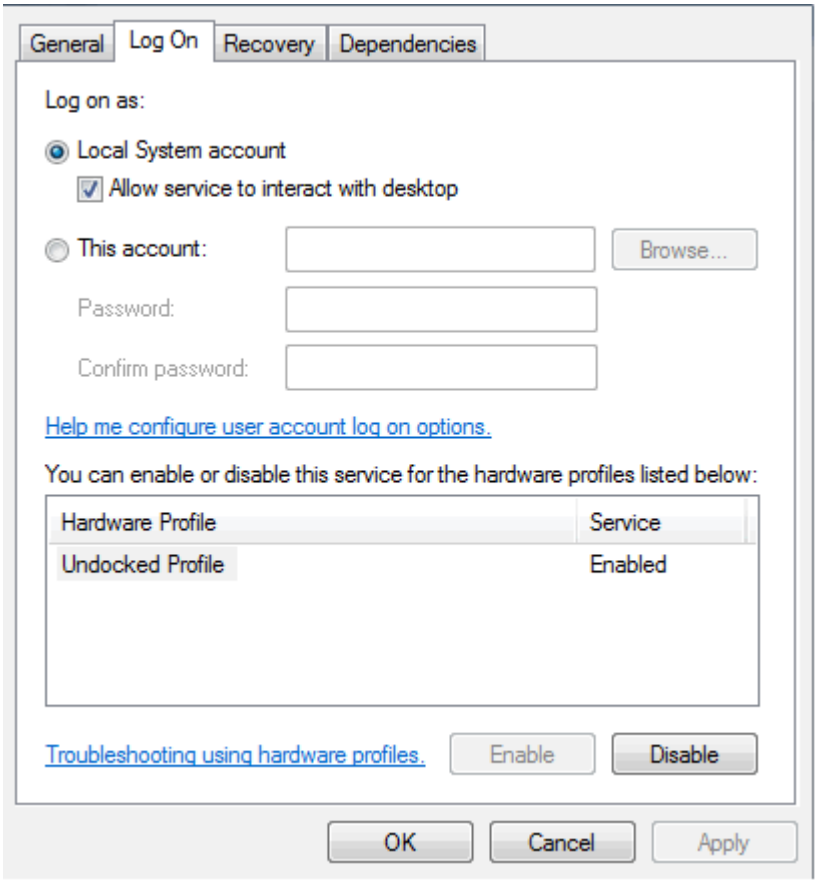
This configuration is for the same application as outlined in the *JBOSS server* 27 *configuration* topic.

There are two things to notice of importance.

- The Java VM option: -agentpath:c:\program files\sparx systems\ea\ssjavaprofiler80
- The addition to the Class path property of the path to the source code: C:\JBOSS\03b-dao\common\src;



For users running Apache Tomcat as a Windows™ service, it is important to configure the service to enable interaction with the Desktop. Failure to do so causes debugging to fail within Enterprise Architect.



Select the **Allow service to interact with desktop** checkbox.

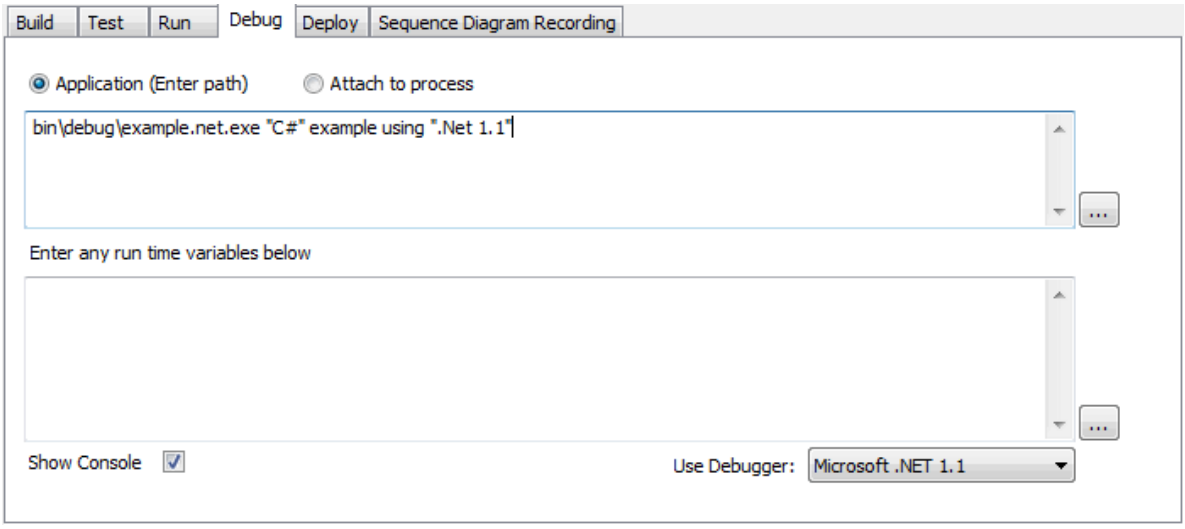
3.6.2.4 .NET

This section describes how to configure Enterprise Architect for debugging .NET applications. It covers:

- General Setup 30
- Debug Assemblies 30
- Debug CLR Versions 31
- Debug COM Interop 32
- Debug ASP .NET 32

3.6.2.4.1 General Setup for .NET

This is the general setup for debugging .NET applications:



Option	Use to
Application (Enter path)	Select and enter either the full or the relative path to the application executable, followed by any command line arguments.
Enter any runtime variables below	Type any required command line options, if debugging a single .NET Assembly 30.
Show Console	Create a console window for the debugger. Not applicable for attaching to a process.
Use Debugger	Select the debugger to suit the .NET Framework under which your application runs.

Note:

If you intend to debug managed code using an unmanaged application, please see the *Debug - CLR Versions* 31 topic.

3.6.2.4.2 Debug Assemblies

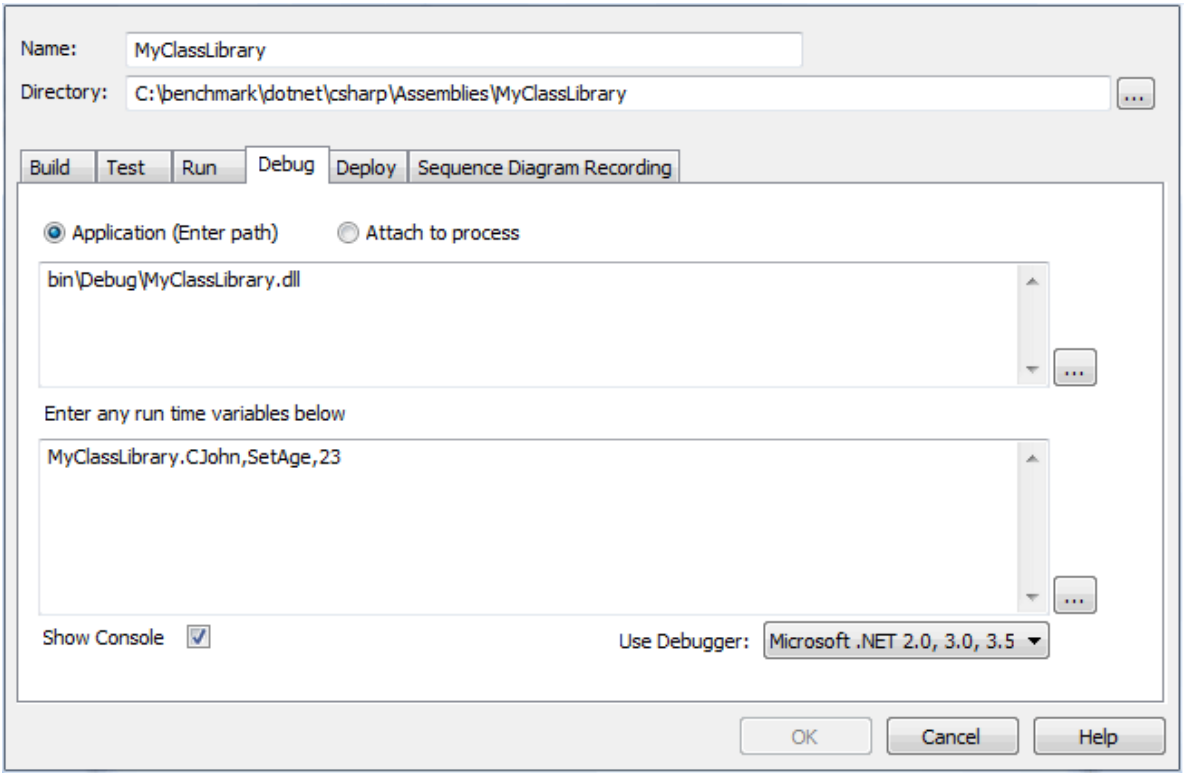
Enterprise Architect permits debugging of individual assemblies.

The assembly is loaded and a specified method invoked. If the method takes a number of parameters, these can be passed.

Constraints

Debugging of assemblies is only supported for .NET version 2.

The following image is of a Build Script configured for debugging a .NET assembly.



Notice the **Enter any run time variables below** field. This field is a comma-delimited list of values that must present in the following order:

type_name, method_name, { method_argument_1, method_argument2,....}

where:

- *type_name* is the qualified type to instantiate
- *method_name* is the unqualified name of the method belonging to the type that is invoked
- the *argument list* is optional depending on the method invoked.

The information in this field is passed to the debugger.

3.6.2.4.3 Debug - CLR Versions

Please note that if you are debugging managed code using an unmanaged application, the debugger might fail to detect the correct version of the Common Language Runtime (CLR) to load. You should specify a config file if you don't already have one for the debug application specified in the *Debug* command of your script. The config file should reside in the same directory as your application, and take the format:

```
name.exe.config
```

where *name* is the name of your application.

The version of the CLR you should specify should match the version loaded by the managed code invoked by the debuggee.

Sample config file:

```
<configuration>
        <startup>
            <requiredRuntime version="version "/>
        </startup>
</configuration>
```

where *version* is the version of the CLR targeted by your plugin or COM code.

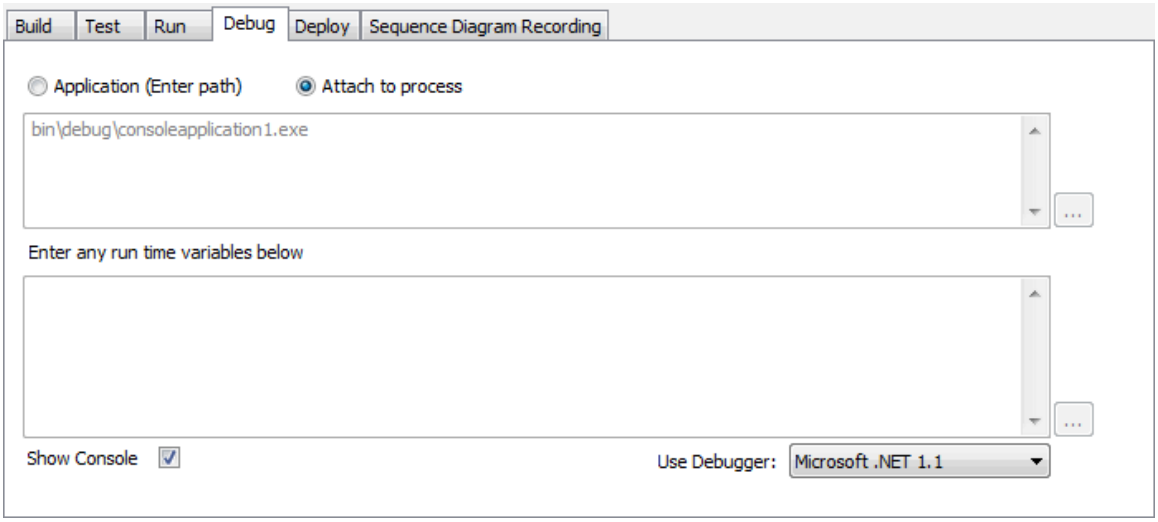
For further information, see http://www.msdn2.microsoft.com/en-us/library/9w519wzk.aspx.

3.6.2.4.4 Debug COM Interop

Enterprise Architect enables you to debug .NET managed code executed using COM in either a Local or an In-Process server.

This feature is useful for debugging Plugins and ActiveX components.

1. Create a package in Enterprise Architect and import the code to debug. See *Code Engineering Using UML Models*.
2. Ensure the COM component is built with debug information.
3. Create a Script for the Package.
4. In the Debug tab, you can elect to either attach to an unmanaged process or specify the path to an unmanaged application to call your managed code.



5. Add breakpoints in the source code to debug.

Attach to an Unmanaged Process

- If an In-Process COM server, attach to the client process or
- If a Local COM Server, attach to the server process.

Click on the Debug window **Run** button (or press **[F6]**) to display a list of processes from which you can choose.

Important:

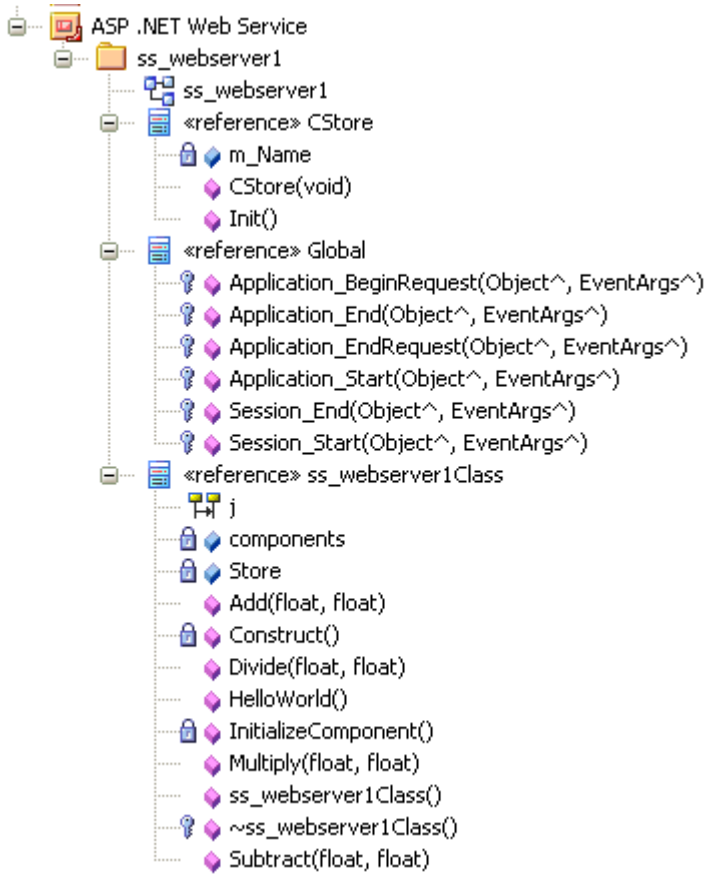
Detaching from a COM interop process you have been debugging terminates the process. This is a known issue for Microsoft .NET Framework, and information on it can be found on many of the MSDN .NET blogs.

3.6.2.4.5 Debug ASP .NET

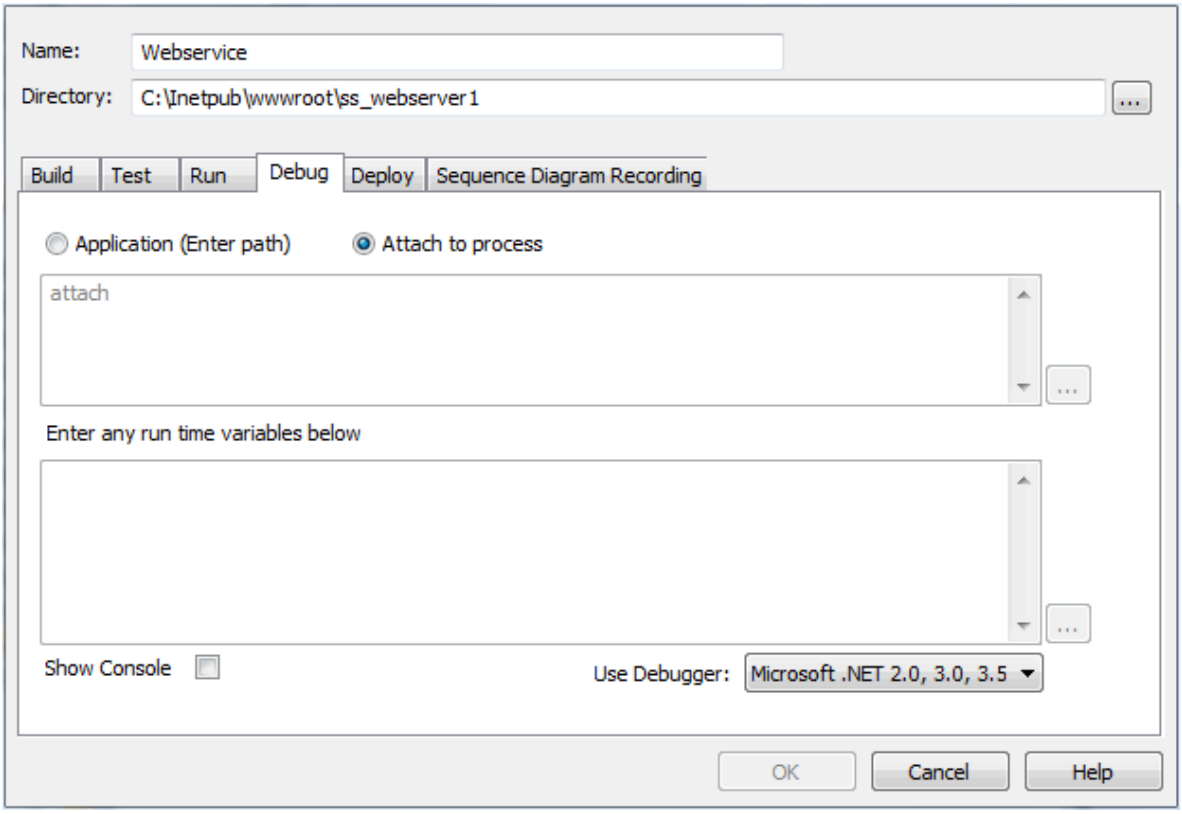
Debugging for web services such as ASP requires that the Enterprise Architect debugger is able to attach to a running service.

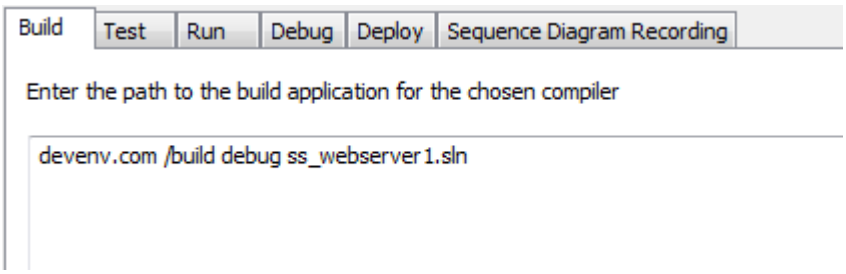
Begin by ensuring that the directory containing the ASP .NET service project has been imported into Enterprise Architect and, if required, the web folder containing the client web pages. If your web project directory resides under the website hosting directory, then you can import from the root and include both ASP code and web pages at the same time.

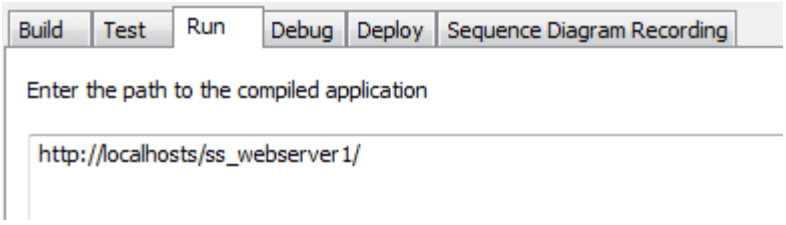
The following image shows the project tree of a web service imported into Enterprise Architect.



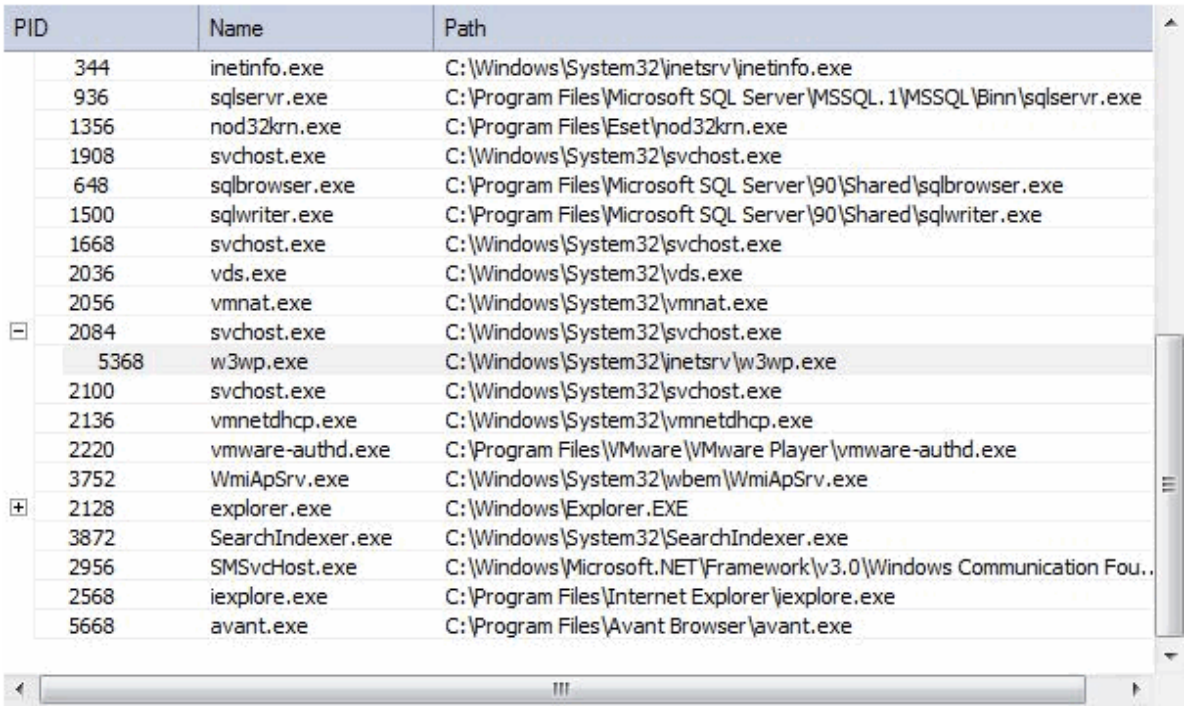
It is necessary to launch the client first, as the ASP .NET service process might not already be running. Load the client by using your browser. This ensures that the web server is running. The only difference to a debug script for ASP is that you specify the **attach** keyword in your script, as follows:







When you start the debugger (click on the Debug window 15 **Run** button) the Attach To Process dialog displays.



PID	Name	Path
344	inetinfo.exe	C:\Windows\System32\inetsrv\inetinfo.exe
936	sqlservr.exe	C:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Binn\sqlservr.exe
1356	nod32krn.exe	C:\Program Files\Eset\nod32krn.exe
1908	svchost.exe	C:\Windows\System32\svchost.exe
648	sqlbrowser.exe	C:\Program Files\Microsoft SQL Server\90\Shared\sqlbrowser.exe
1500	sqlwriter.exe	C:\Program Files\Microsoft SQL Server\90\Shared\sqlwriter.exe
1668	svchost.exe	C:\Windows\System32\svchost.exe
2036	vds.exe	C:\Windows\System32\vds.exe
2056	vmnat.exe	C:\Windows\System32\vmnat.exe
2084	svchost.exe	C:\Windows\System32\svchost.exe
5368	w3wp.exe	C:\Windows\System32\inetsrv\w3wp.exe
2100	svchost.exe	C:\Windows\System32\svchost.exe
2136	vmnetdhcp.exe	C:\Windows\System32\vmnetdhcp.exe
2220	vmware-authd.exe	C:\Program Files\VMware\VMware Player\vmware-authd.exe
3752	WmiApSrv.exe	C:\Windows\System32\wbem\WmiApSrv.exe
2128	explorer.exe	C:\Windows\Explorer.EXE
3872	SearchIndexer.exe	C:\Windows\System32\SearchIndexer.exe
2956	SMSvcHost.exe	C:\Windows\Microsoft.NET\Framework\v3.0\Windows Communication Fou..
2568	iexplore.exe	C:\Program Files\Internet Explorer\iexplore.exe
5668	avant.exe	C:\Program Files\Avant Browser\avant.exe

Note that the name of the process varies across Microsoft operating systems.; check the *ASP .NET SDK* for more information. The image above shows the IIS process *w3wp.exe*, which is the name of the process that runs under Windows Vista.

On Windows XP, the name of the process is something like *aspnet_wp.exe*, although the name could reflect the version of the .NET framework that it is supporting. There can be multiple ASP.NET processes running under XP; you must ensure that you attach to the correct version, which would be the one hosting the .NET framework version that your application runs on. Check the *web.config* file for your web service to verify the version of .NET framework it is tied to.

The Debug window **Stop** button should be enabled and any breakpoints 37 should be red, indicating they have been bound.

Note:

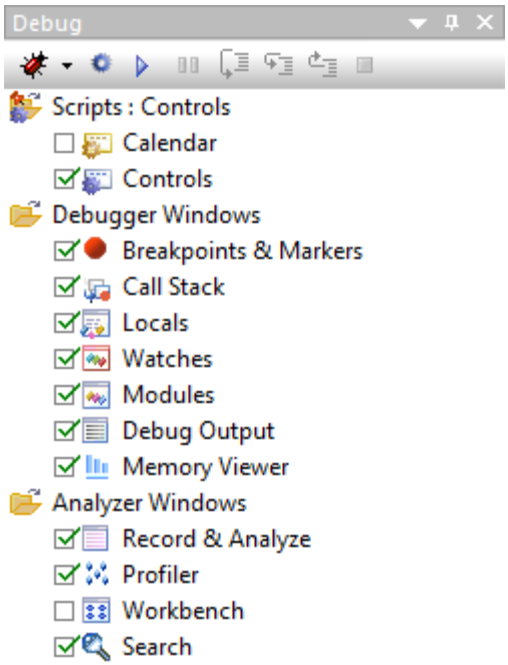
Some breakpoints might not have bound successfully, but if none at all are bound (indicated by being dark red with question marks) something has gone out of sync. Try rebuilding and re-importing source code.

You can set breakpoints at any time in the web server code. You can also set breakpoints in the ASP web page(s) if you imported them.

3.6.3 The Debug Window

The Debug window gives access to the scripts and windows of the debug facility.

To access the Debug window, select the **View | Execution Analyzer | Debugger** menu option.



The Debug Window has three top-level folders:

- Scripts - The *Scripts <Package Name>* folder lists the scripts available for the currently-selected package, the first in the list being, by default, the active script that is executed when you start debugging, as indicated by the selected checkbox. If you are using recording markers, this is also the script that determines what recording options are applied 59. If you want to execute a different script, select the appropriate checkbox. The context menu for each script provides further scripting options, such as **Debug**, **Build**, **Test** and **Edit**.

  You can pin the package scripts so that they remain listed in the Debug window even if you select a different package. To do this, right-click on the folder title and select the **Pin Package** context menu option; the Scripts folder icon changes. To unpin the scripts, right-click on the folder title and deselect the **Pin Package** option.
- The *Debugger Windows* folder lists the debug windows, which you can display or hide by selecting or deselecting the checkbox against each one. If the window is docked, you can bring it to the front by clicking on the window name:
  - Breakpoints & Markers 37 - lists any breakpoints placed in the package source code, along with their status (enabled/disabled), line number, and the physical source file in which they are located
  - Call Stack 42 - shows the position of the debugger in the code; clicking on the > button advances the stack through the code until the next breakpoint is reached
  - Locals 43 - shows the local variables defined in the current code segment, their type and value
  - Watches 44 - shows the values of static and globally scoped expressions you have entered
  - Modules 47 - displays all the modules loaded during a debug session
  - Debug Output 47 - displays output from the debugger including any messages output by the debugged process, such as writes to standard output.
- The *Analyzer Windows* folder lists the advanced control windows of the Execution Analyzer, which you can display or hide by selecting or deselecting the checkbox against each one:
  - Record & Analyze 74 - records any activity that takes place during a debug session; once the activity has been logged, Enterprise Architect can use it to create a new Sequence diagram
  - Profiler 82 - opens the Profiler window to sample an application
  - Workbench 87 - enables you to create instances of .NET and Java Classes
  - Search 51 - enables you to search for text in files.

You can dock and combine the windows to suit your working requirements; see the *Arrange Windows and Menus* section in *Using Enterprise Architect - UML Modeling Tool*.

3.6.4 Breakpoint and Marker Management

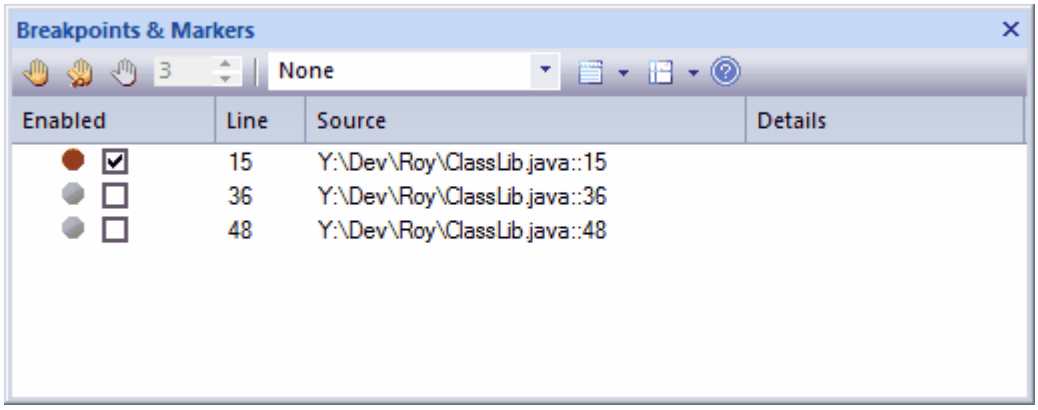
Breakpoints work in Enterprise Architect much like in any other debugger. Adding a breakpoint notifies the debugger to trap code execution at the point you have specified. When a breakpoint is encountered by a thread of the application being debugged, the source code is displayed in an editor window, and the line of code where the breakpoint occurred is highlighted.

Selecting a different package in the project affects which breakpoints are displayed.

Note:

The debugger does not stop automatically. It runs to completion unless it encounters a breakpoint.

An Enterprise Architect model maintains breakpoints for every package having a *Build Script - Debug* command. Breakpoints themselves are listed in their own Breakpoints & Markers window (**View | Execution Analyzer | Breakpoints & Markers**).



Breakpoint States

	DEBUGGER STATE	
	Running	**Not running**
	Bound	Enabled
	Disabled	Disabled
	Not bound - this usually means that the DLL is not yet loaded or was not built with debug information	N/a
	Failed - this usually means a break could not be set at this time, and can occur when the source file is newer or older than that used to build the application.	N/a

Delete, Disable and Enable Breakpoints

To delete a specific breakpoint, either:

- If the breakpoint is enabled, click on the red breakpoint circle in the left margin of the Source Code Editor
- Right-click on the breakpoint marker in the editor and select the appropriate context menu option, or
- Select the breakpoint in the Breakpoints & Markers tab and press **[Delete]**.

Whether you are viewing the *Breakpoints* folder or the Breakpoints & Markers window, you can right-click on an existing breakpoint and select a context menu option either to delete it or to convert it to a start recording marker or end recording marker 66.

You can also delete all breakpoints by clicking on the **Delete all breakpoints** button on the Breakpoints & Markers window toolbar ( ).

To disable a breakpoint, deselect its checkbox on the Breakpoints & Markers window or, to disable all breakpoints, click on the **Disable all breakpoints** button in the toolbar ( ). The breakpoint is then shown as an empty grey circle. Select the checkbox or use the **Enable all breakpoints** button to enable it again ( ).

3.6.4.1 How Markers are Stored

Breakpoints created that are not part of any set are maintained in an external file for the current model.

The file format is as follows:

path\guid.brkpt

where:

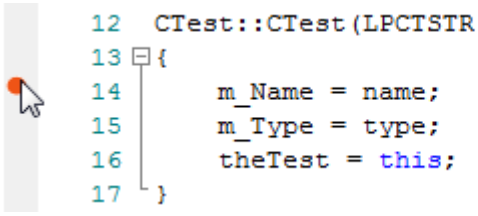
- *path* = The O/S application data directory for each user
- *guid* = model Guid.

Marker Sets are stored in the model and are available to all users of the Model.

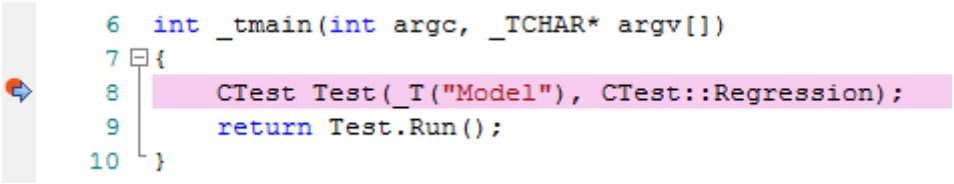
3.6.4.2 Setting Code Breakpoints

To set breakpoints for a code segment:

1. Open the model code to debug.
2. Find the appropriate code line and click in the left margin column. A solid red circle in the margin indicates that a breakpoint has been set at that position.

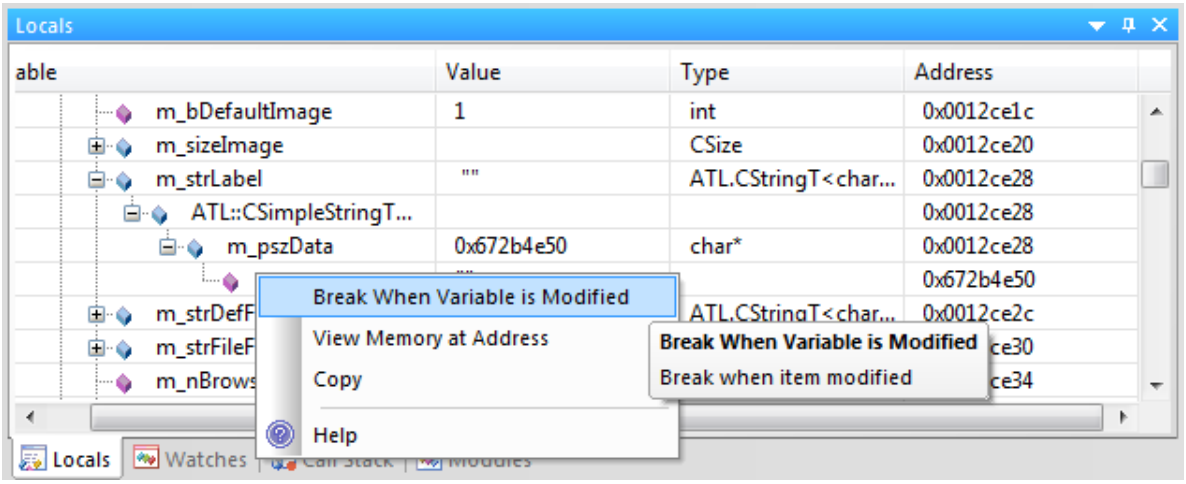


If the code is currently halted at a breakpoint, that point is indicated by a blue arrow next to the marker.



3.6.4.3 Setting Data Breakpoints

Data breakpoints can currently only be set by right-clicking on the variable in the Locals 43 window and selecting the **Set Data Breakpoint** context menu option. This means that in order to establish a data breakpoint you must first set a normal breakpoint 37 at a point in the code that presents the required scope of local variables to choose from.



3.6.5 Debugging Actions

This section describes the actions you perform in running a debug session. It covers:

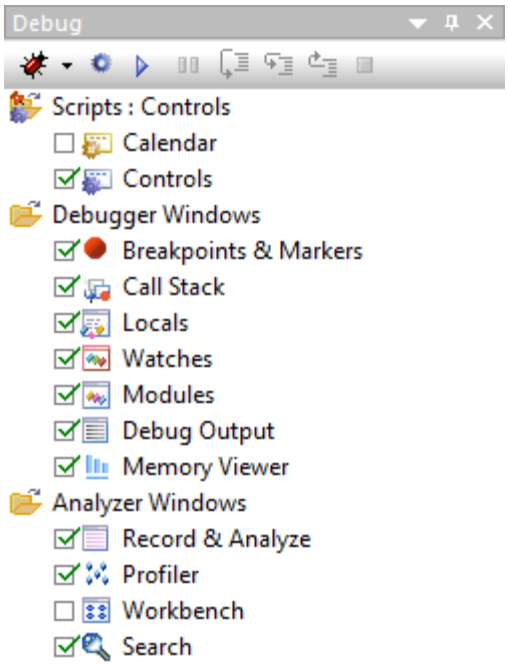
- Displaying Windows 39
- Starting and Stopping the Debugger 40
- Debugging a Subsequent Process 40
- Stepping Over Lines of Code 41
- Stepping Into Function Calls 41
- Stepping Out of Functions 42
- Viewing the Call Stack 42
- Viewing the Local Variables 43
- Viewing the Content of Long Strings 43
- Viewing Variables in Other Scopes 44
- Inspecting Process Memory 45
- Setting Breaks for When a Variable Changes Value 46
- Showing Loaded Modules 47
- Showing Output from the Debugger 47
- Debugging Tooltips in Code Editors. 48

3.6.5.1 Displaying Windows

Debugger Actions - Displaying Windows

The Debugger windows 35 are available from the **View | Execution Analyzer** menu options.

These windows can also be displayed and hidden from the debug management control checkboxes shown below:



3.6.5.2 Start & Stop Debugger

Debugging Actions - Start & Stop

If Basic Setup 8 has been completed, pressing **[F6]** starts the application using the configured Debugger.

If not, debugging is still possible by using the **Attach** button on either one of the Debugger toolbars.

To stop debugging, click on the **Stop** button  in the Debug window toolbar, or press **[Ctrl]+[Alt]+[F6]**.

Notes:

In most situations, the debugger ends:

- when it encounters breakpoints (which should be set beforehand)
- when the debug process terminates or
- when the Java Class thread exits.

However, due to the nature of the Java Virtual Machine, it is necessary at times for Java developers to stop the debugger manually with the **Stop** button.

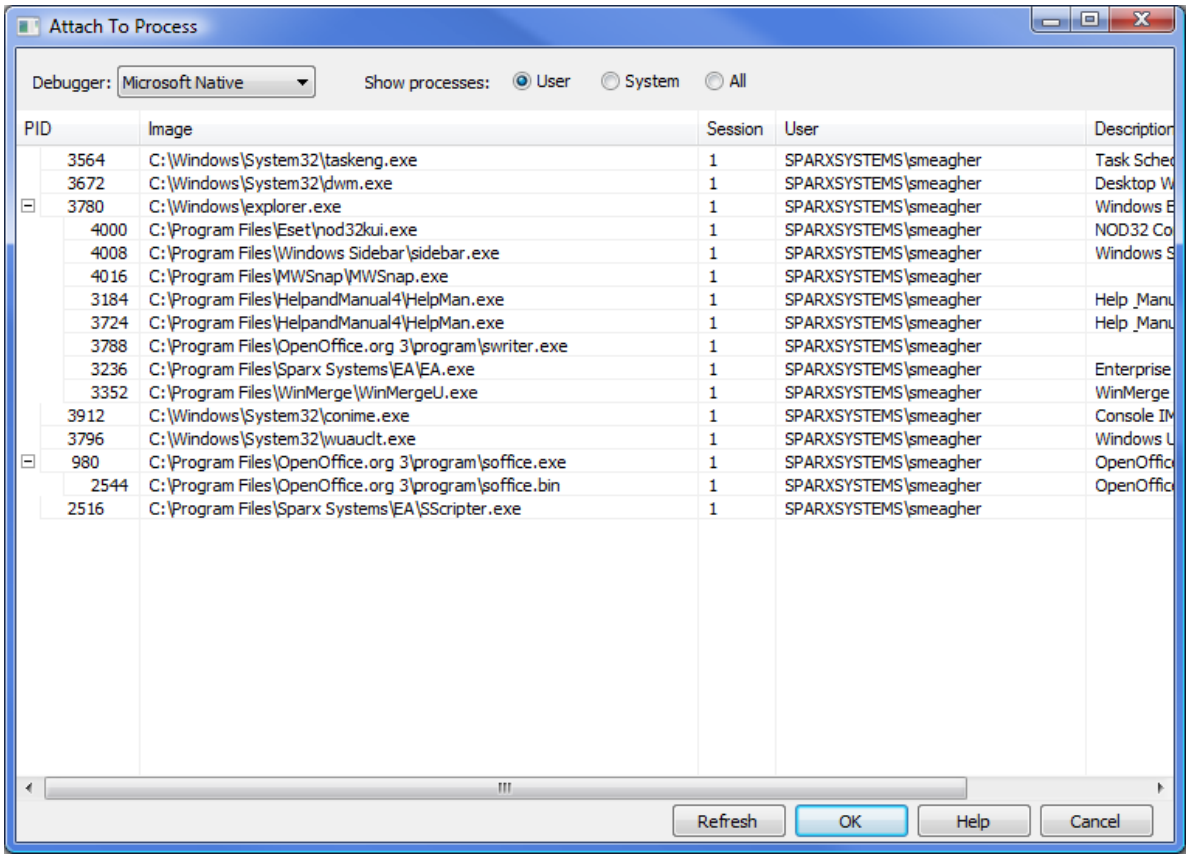
3.6.5.3 Debug Another Process

Debugging Actions - Debug Another Process

When debugging another process, the Attach To Process dialog is displayed.

You can limit the processes displayed using the radio buttons at the top of the dialog. To find a service such as Apache Tomcat or ASP.NET, select the **System** radio button.

You must choose the debugger when you select a process. However, if the selected Package has already been configured for debugging then the Debugger listed is the one specified in the Script.



Once Enterprise Architect is attached to the process, any breakpoints encountered are detected by the debugger and the information is available in the Debugger windows.

To detach from a process, click on the **Debug Stop** button.

3.6.5.4 Step Over Lines of Code

Debugging Actions - Step Over

You can only step over the lines of a function using the Debug toolbar buttons.

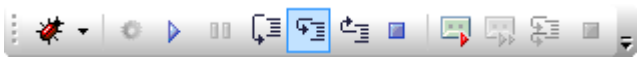
When you step to the end of the function, you step back to the caller.

Alternatively, press **[Alt]+[F6]** or select the **Project | Execution Analyzer | Step Over** context menu option.

3.6.5.5 Step Into Function Calls

Debugging Actions - Step In

The *Step In* function is executed by clicking on the **Step In** button.



Alternatively, press **[Shift]+[F6]** or select the **Project | Execution Analyzer | Step In** context menu option.

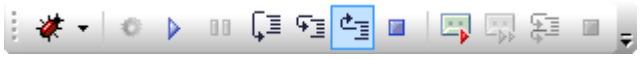
If no source is available for the function then the debugger continues stepping till it either enters a new

function or reaches the next line of the current one.

3.6.5.6 Step Out of Functions

Debugging Actions - Step Out

The Step Out function is executed by clicking on the **Step Out** button.



Alternatively, press **[Ctrl]+[F6]** or select the **Project | Execution Analyzer | Step Out** context menu option.

If no source is available for the function then the debugger will continue stepping till it either enters a new function or reaches the

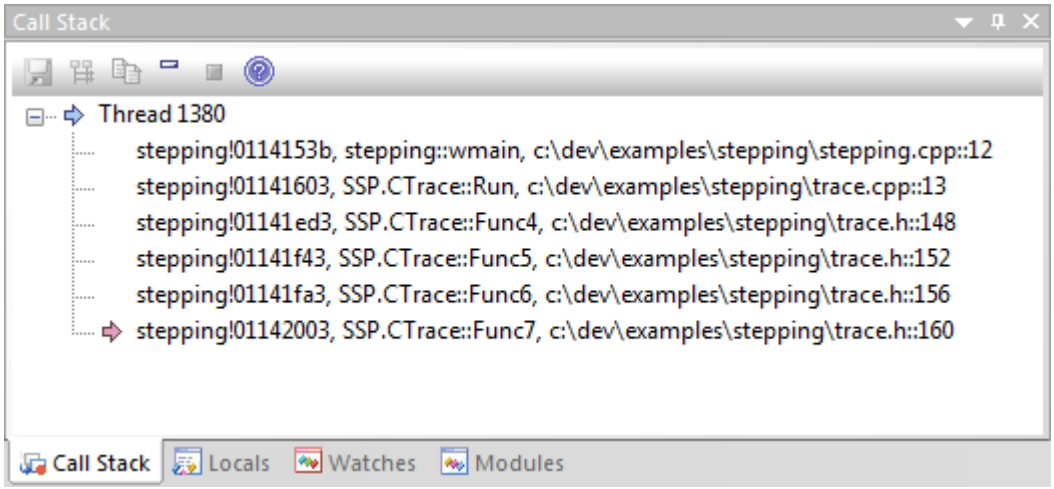
next line of the current one.

3.6.5.7 View the Call Stack

Debugging Actions - View the Call Stack

The Call Stack window shows all currently running threads. A Stack trace is displayed whenever a thread is suspended, through one of the step actions or through encountering a breakpoint 37.

- A green or yellow arrow highlights the current stack frame
- A blue arrow indicates a thread that is running
- A red arrow indicates a thread for which a stack trace history is being recorded
- Double-clicking a frame takes you to that line of code in the Source Code Editor; local variables are also refreshed for the selected frame.



Toolbar

 - Save Stack to file

 - Generate Sequence diagram from Stack

 - Copy Stack to recording history

 - Toggle Stack View

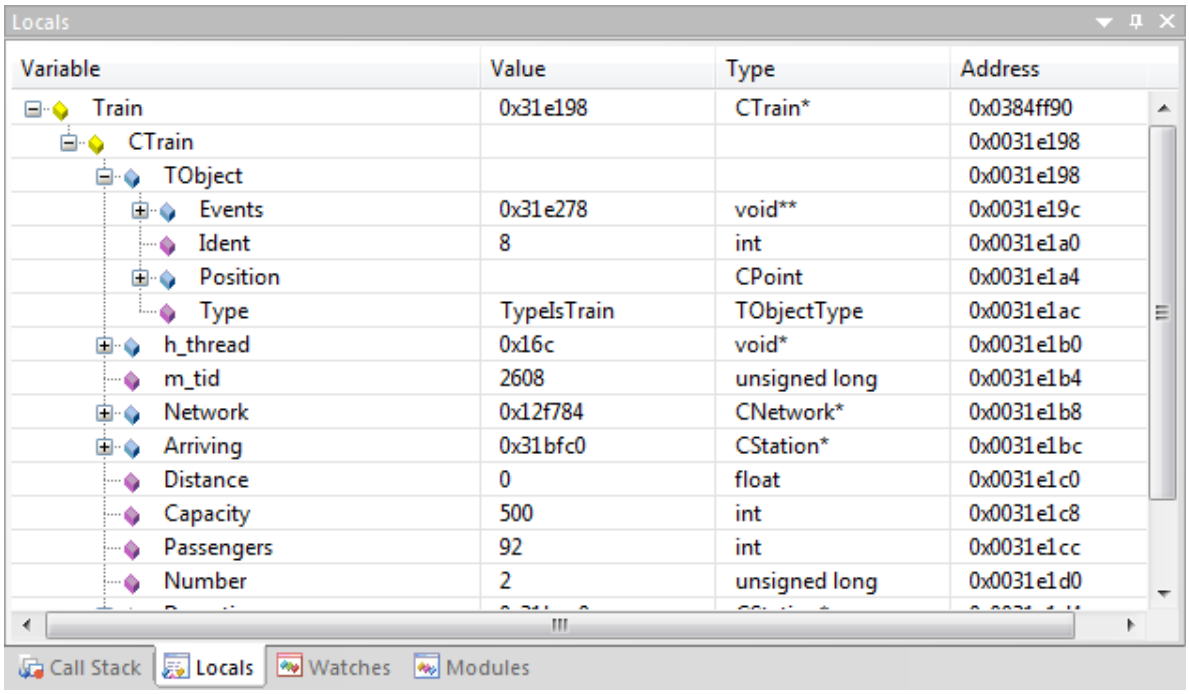
- Stop recording

3.6.5.8 View the Local Variables

Debugging Actions - Viewing Local Variables

Whenever a thread encounters a breakpoint 37, the Locals window displays all the local variables for the thread at its current stack 42 frame.

The value and the type of any in-scope variables are displayed in a tree, as illustrated below:



Local variables are displayed with colored box icons with the following meanings:

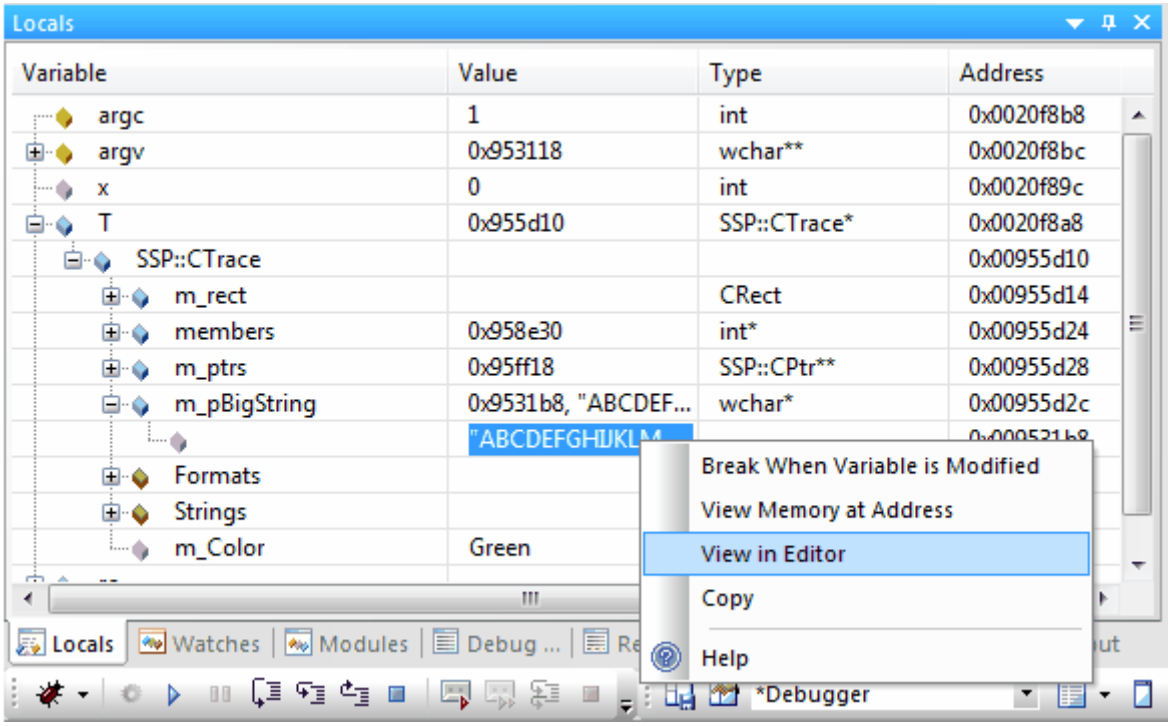
- Blue Object with members
- Green Arrays
- Pink Elemental types
- Yellow Parameters
- Red Workbench Instance

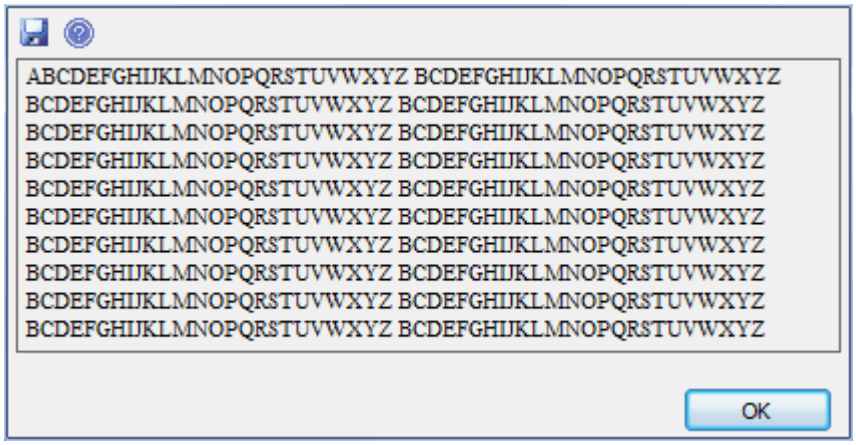
3.6.5.9 View Content Of Long Strings

Debugging Actions - View Entire Content Of Long Strings

For efficiency, the Locals window only shows partial strings. The size of any variable value displayed in the window can be up to 256 characters.

To view the entire value of a variable, right-click on it and select the **View in Editor** context menu option. The String Viewer dialog displays.



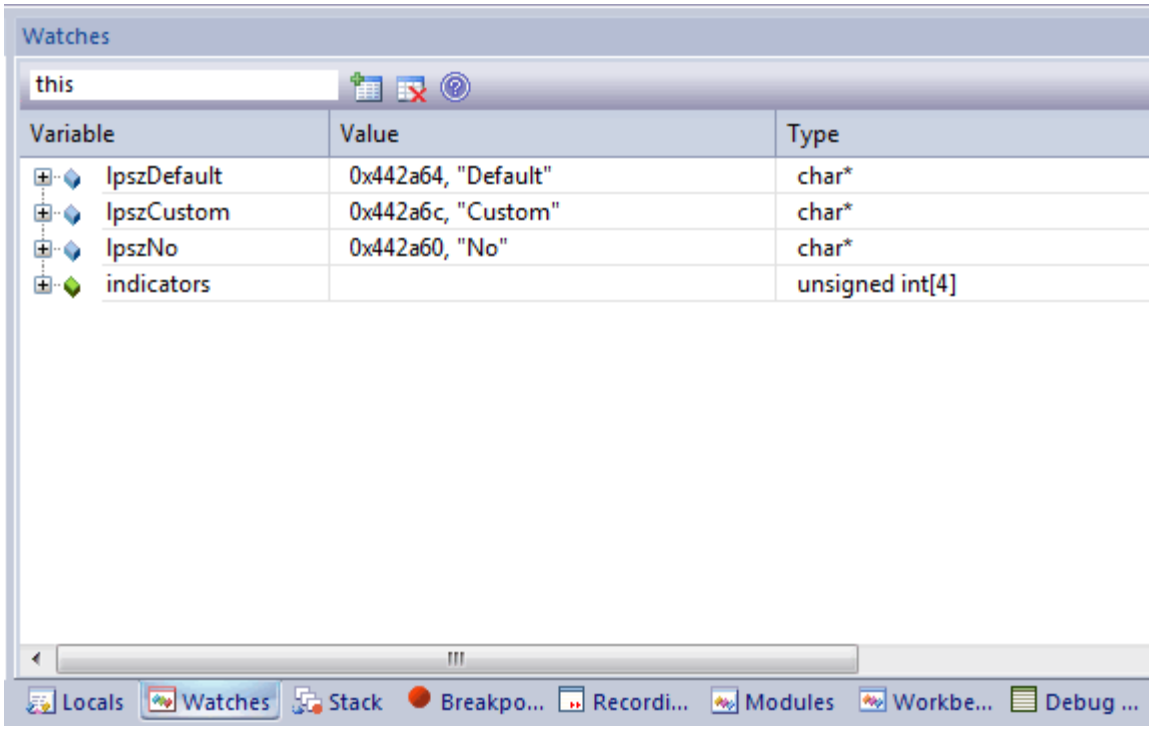


3.6.5.10 View Variables in Other Scopes

Debugging Actions - Viewing the Variables in Other Scopes

The Watches window is most useful for native code (C, C++, VB) where it can be used to evaluate data items that are not available as Local Variables 43 - data items with module or file scope and static Class member items.

You can also use the window to evaluate static Class member items in Java and .NET.



To use the Watches window, type the name of the variable to examine in the field in the window toolbar, and either press **[Enter]** or click on the **Add new watched item** icon.

To examine a static Class member variable in C++, Java or Microsoft .NET enter its fully qualified name. For example:

CMyClass::MyStaticVar

To examine a C++ data symbol with module or file scope, just enter its name. Note, items are evaluated only if the package in whose scope the item resides is currently loaded by the process being debugged. If the debugger is not running, no items are listed.

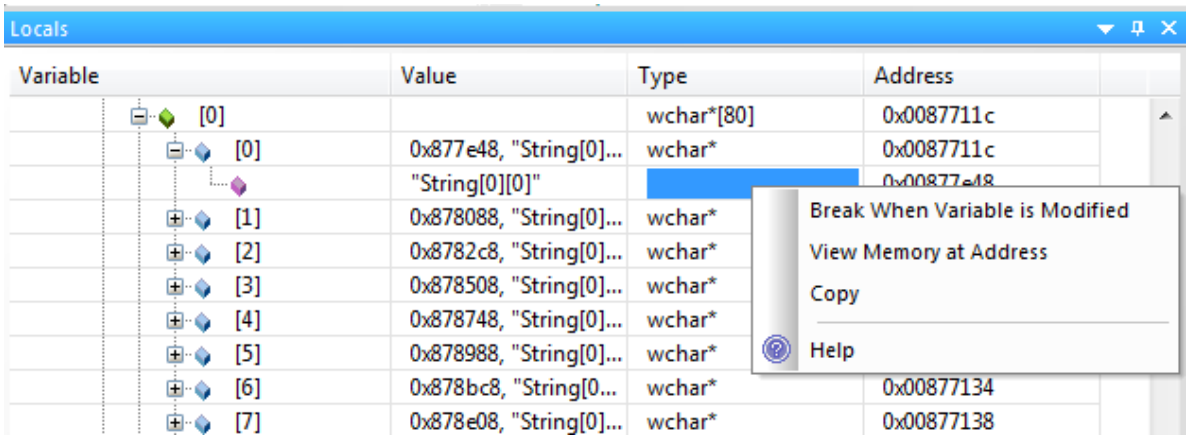
The names of the items to evaluate persist for the package and user ID, so the next time *you* debug the same project, the items evaluate automatically whenever a breakpoint occurs. They do not appear if another user debugs the same code.

If necessary, you can delete items using the **Delete all watched items** icon in the toolbar, or the right-click context menu options inside the window.

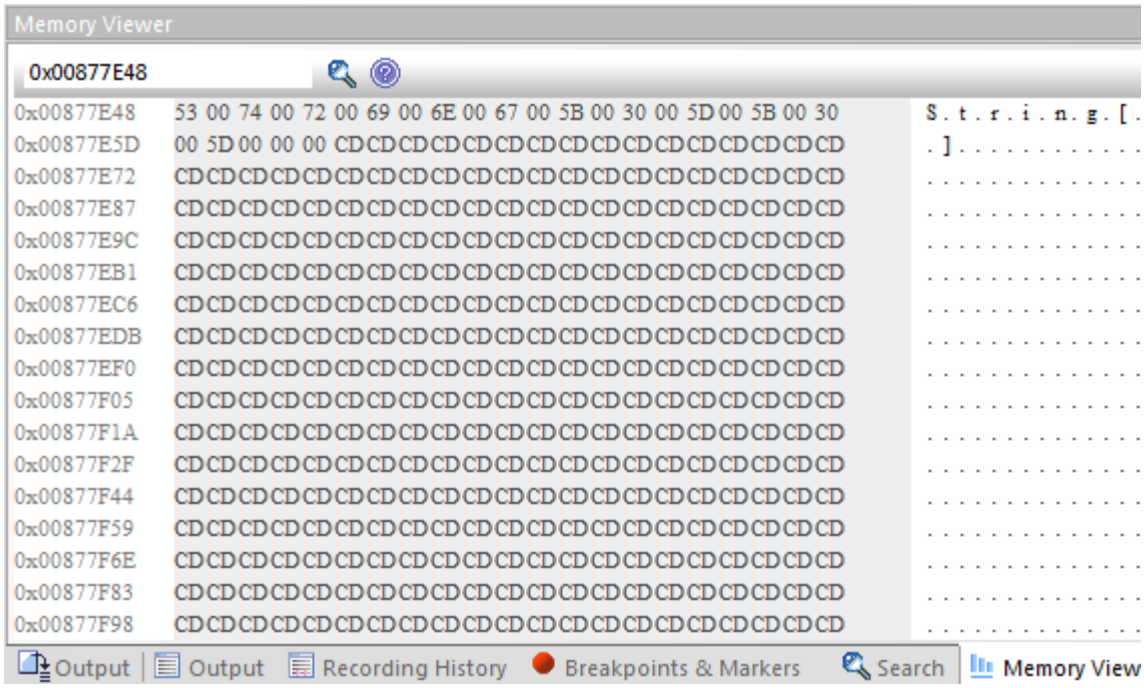
3.6.5.11 Inspect Process Memory

Debugging Actions - Inspecting Process Memory Debugging

You can display the raw values at a memory address or for a variable in a window using the Memory Viewer - select the **View Memory at Address** context menu option.



The Memory Viewer displays the raw values at a memory address



The Memory Viewer is available for debugging Microsoft Native Code Applications (C,C++,VB) running on Windows or within WINE on Linux.

3.6.5.12 Break When a Variable Changes Value

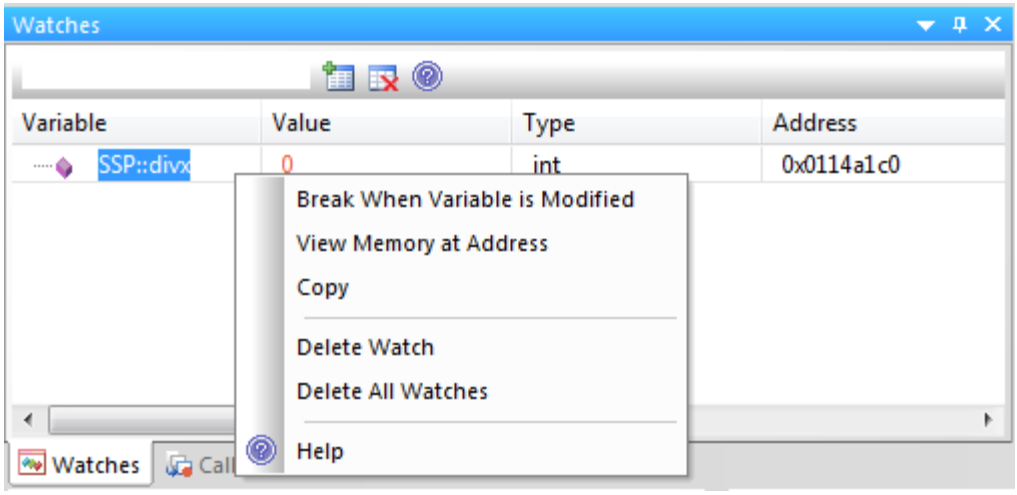
Debugging Actions - Break when a Variable Changes Value

An invalid or uninitialised variable can cause the program behaviour to differ from expected. This tool enables you to halt execution whenever a certain variable has its value changed.

Note:

This feature is not presently supported by the Microsoft .NET platform.

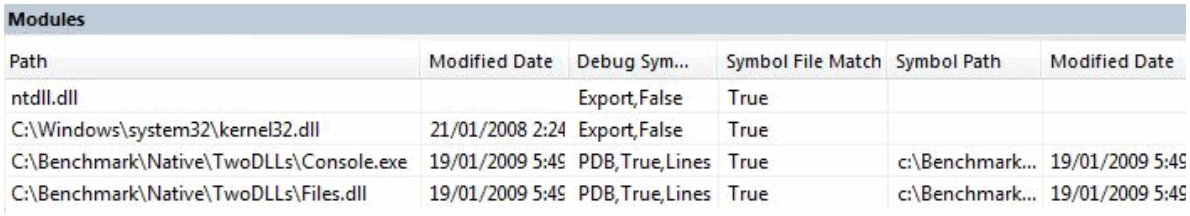
The example below creates a notification on a variable from the Watches window. The item being watched is an integer in the *SSP* namespace scope.



3.6.5.13 Show Loaded Modules

Debugging Actions - Show loaded modules

The debugger Modules window lists the modules loaded by the process being debugged.



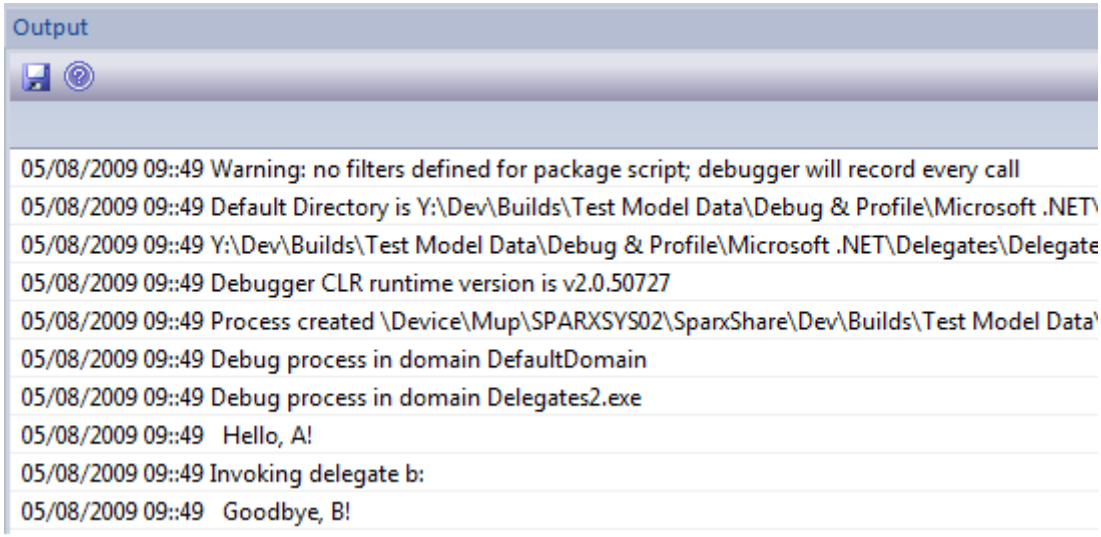
The columns on this window are described below:

Column	Use To
Path	Determine the file path of the loaded module.
Modified Date	Determine the local file date and time the module was modified.
Debug Symbols	Establish the debug symbols type, whether debug information is present in the module and whether line information is present for the module (required for debugging).
Symbol File Match	Check the validity of the symbol file; if the value is false, the symbol file is out of date.
Symbol Path	Determine the file path of the symbol file, which must be present for debugging to work.
Modified Date	Determine the local file date and time the symbol file was created.

3.6.5.14 Show Output from Debugger

Debugging Actions - Show Output From Debugger

During a debug session the Debugger emits messages detailing both startup and termination of session, to its Output tab. Details of exceptions and any errors are also output to this tab. Any trace messages such as those output using *Java System.out* or *.NET System.Diagnostics.Debug* are also captured and displayed here.

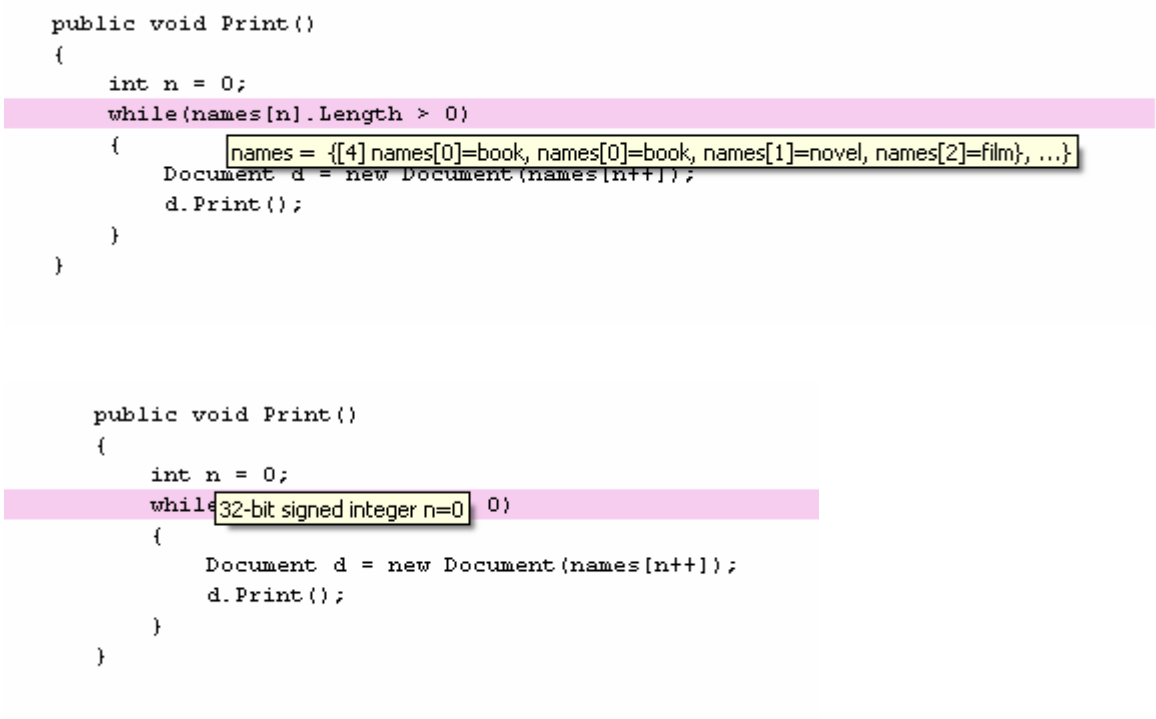


3.6.5.15 Debug Tooltips in Code Editors

Debugging Actions - Viewing Variable Values in Code Editors

During debugging, whenever a thread is suspended at a line of execution, you can inspect member variables in the Editor window.

To evaluate a member variable, use the mouse to move the cursor over the variable in the Editor window, as shown in the following examples.



3.6.6 Recording Actions

This section describes how to perform the following debug recording actions:

- Step through function calls 49
- Create a Sequence diagram of the Call Stack 49
- Save the Call Stack. 50

3.6.6.1 Step Through Function Calls

Debugging Actions - Step Through

The Step Through function can be executed by clicking on the **Step Through** button on the Record & Analyze window toolbar.

Alternatively, press **[Shift]+[F6]** or select the **Project | Execution Analyzer | Step Into** context menu option.

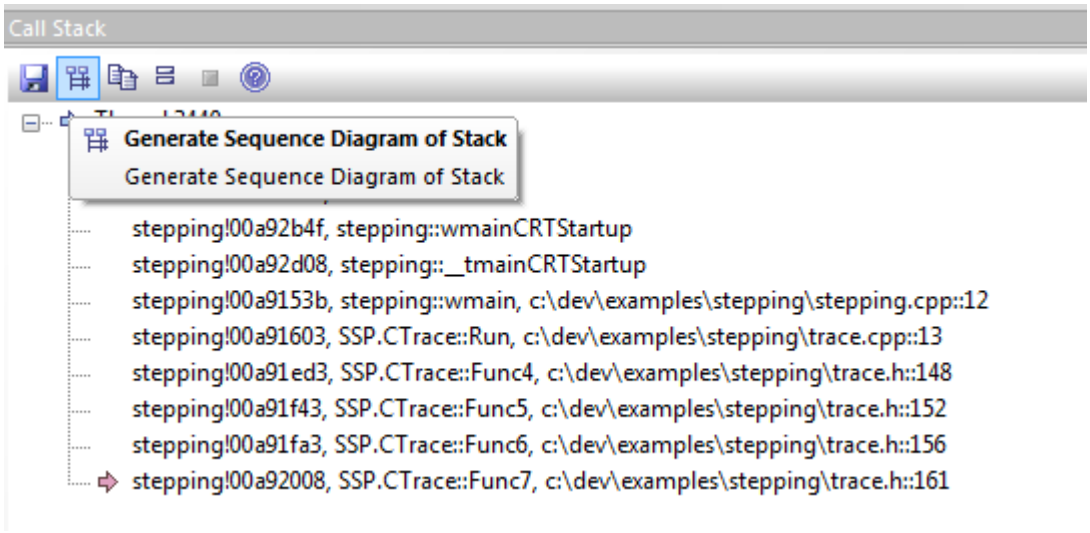
The *Step Through* command causes a *Step Into* command to be executed. If any function is detected, then that function call is recorded in the History window. The debugger then steps out, and the process can be repeated.

This button enables you to record a call without having to actually step into a function. The button is only enabled when at a breakpoint and in manual recording mode.

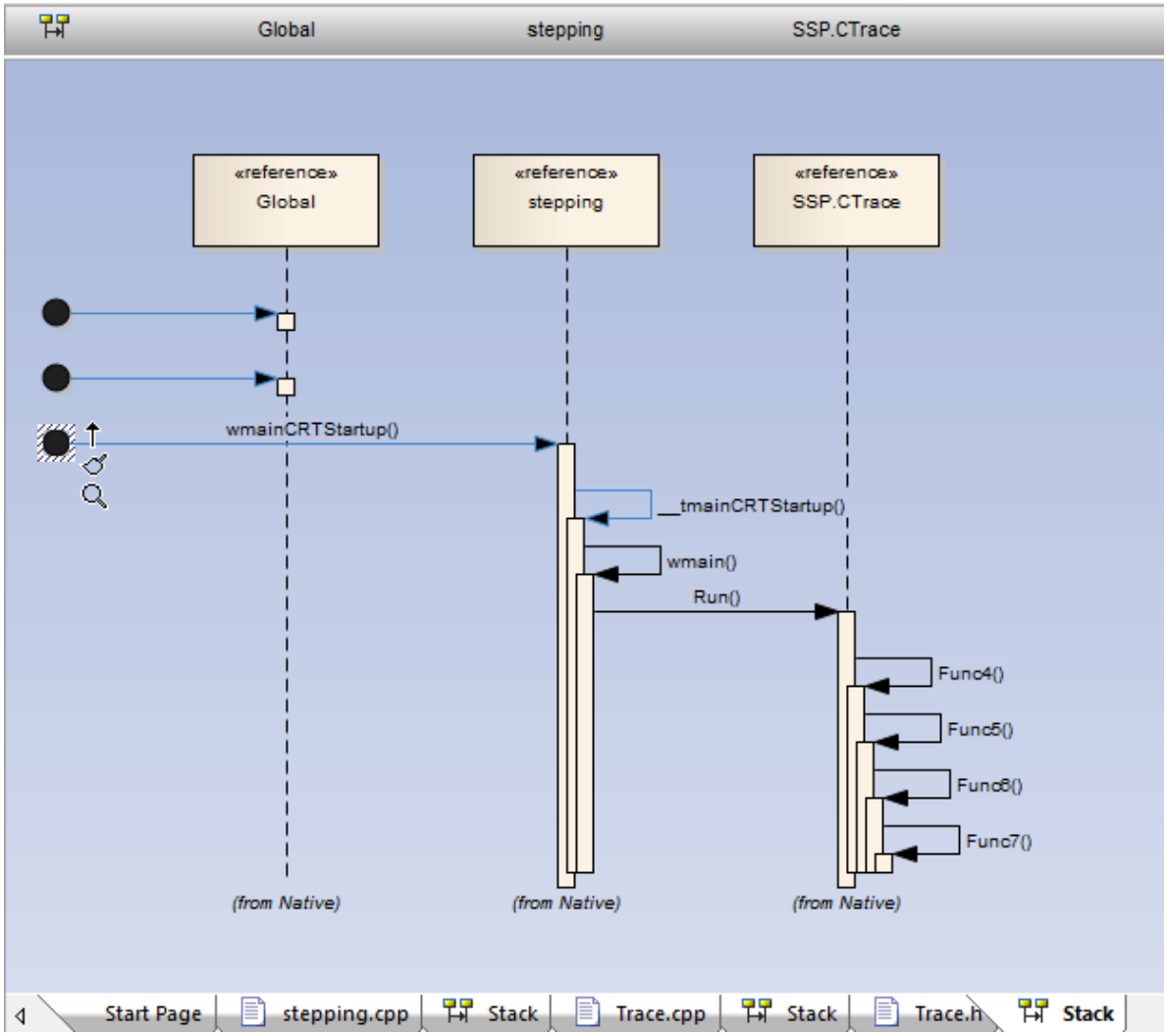
3.6.6.2 Create Sequence Diagram of Call Stack

Debugging Actions - Create Sequence Diagram from Current Call Stack

To generate a Sequence diagram from the current Stack, click on the **Generate Sequence Diagram of Stack** button on the Call Stack window toolbar.



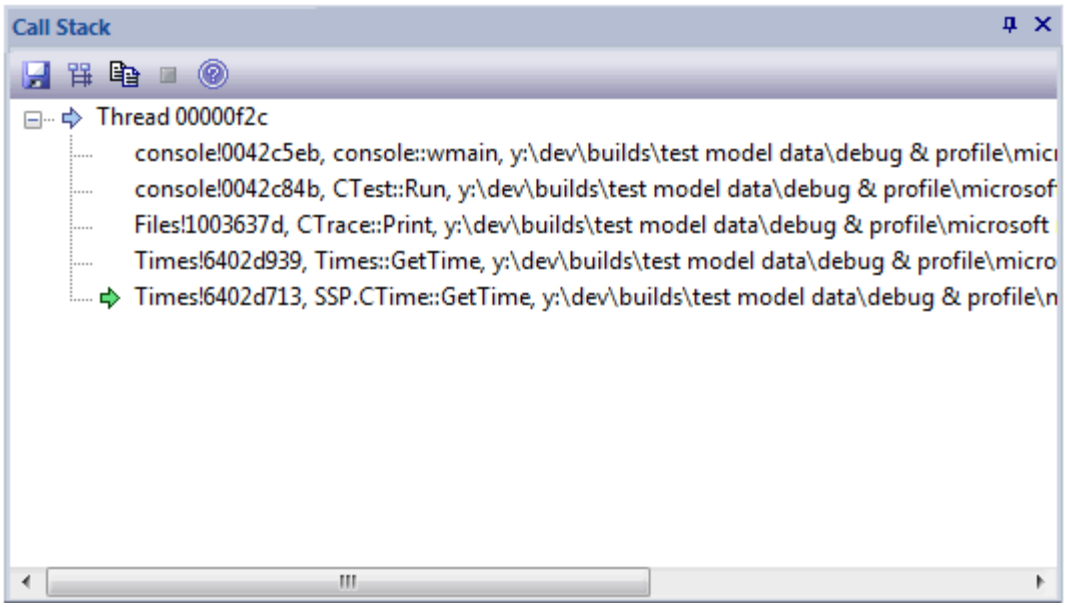
This immediately generates a Sequence diagram in the Diagram View.



3.6.6.3 Saving the Call Stack

Debugging Actions - Saving the Call Stack

On the Call Stack window, you can save the current Stack to file or copy the Stack to the recording history.



Toolbar

 - Save Stack to file

 - Copy Stack to recording history

3.7 Searching in Files

This topic describes how to use the File Search 51 facility.

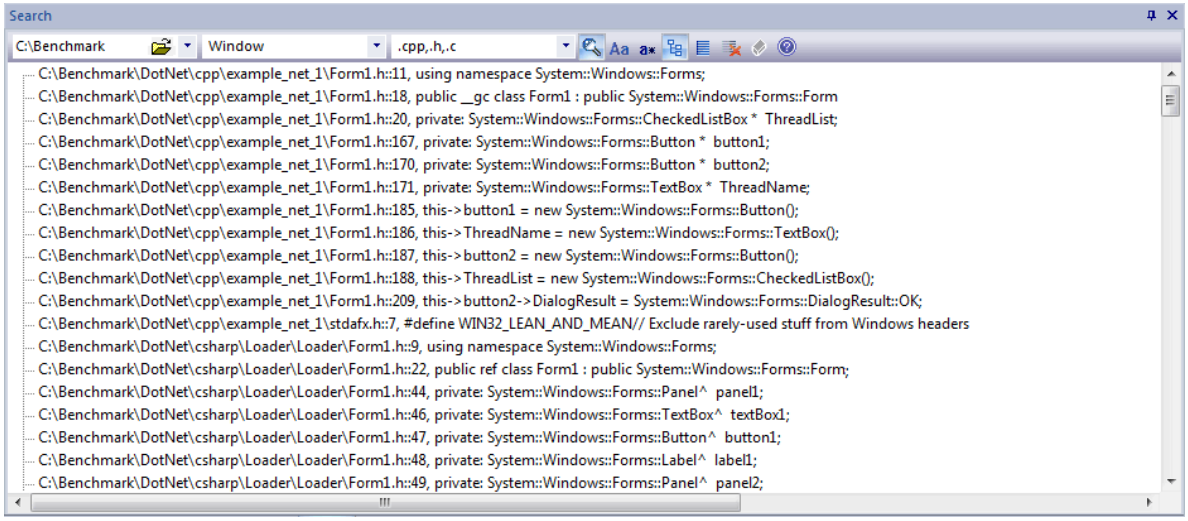
3.7.1 Search in Files

This topic describes the **File Search** control.

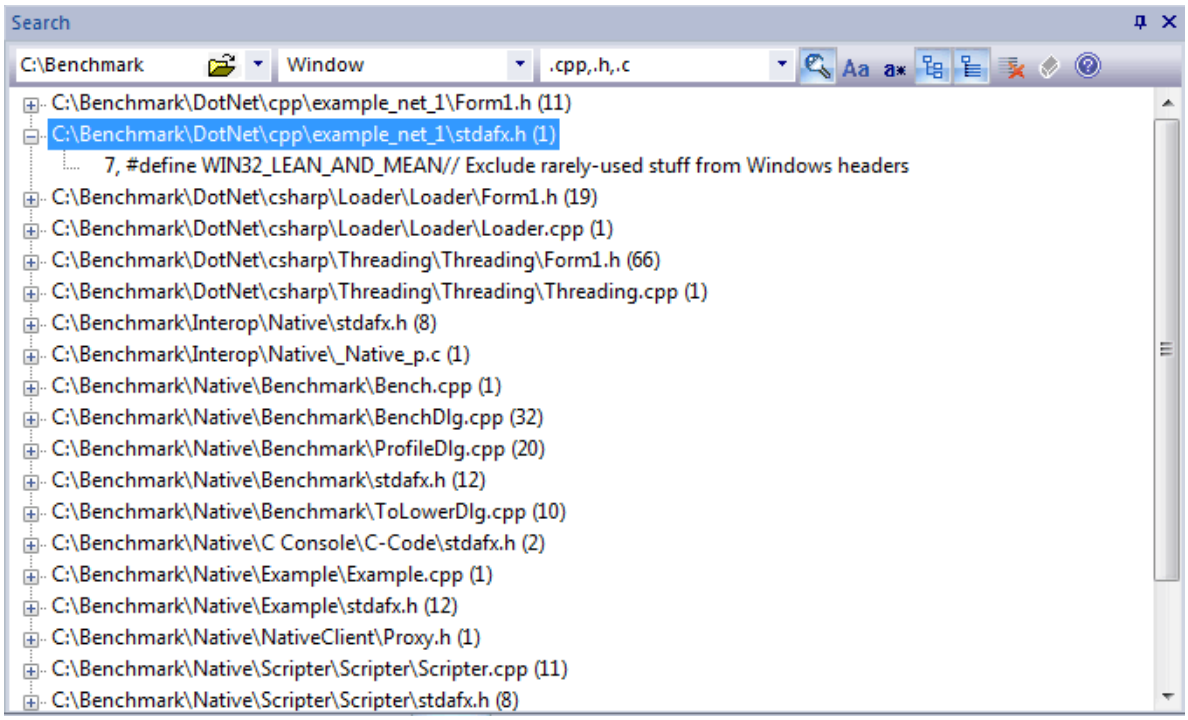
File Text Searches are provided by the Search Window and from within the Code Editors.

The Search window enables you to search for text in code files and scripts. You can select to display the results of the search in one of two formats:

- *List View* - each result line consists of the file path and line number, followed by the line text; multiple lines from one file are listed as separate entries



- *Tree View* - each result line consists of the file path that matches the search criteria, and the number of lines matching the search text within that file; you can expand the entry to show the line number and text of each line.



Search Toolbar

You can use the toolbar options in the Search window to control the search operation. The state of all buttons persists over time to always reflect your previous search criteria.

The options, from left to right, are as follows:

Option	Use to
Search Path list box	Specify the folder to search.

Option	Use to
	You can type the path to search directly into the text box, or click on the folder icon to browse for the path. Any paths you enter are automatically saved in the drop-down list, up to a maximum of ten; paths added after that overwrite the oldest path in the list. A fixed option in the drop-down list is **Search in Scripts**, which sets the search to operate on all local and user-defined scripts in the Scripts tab of the Scripter window. This option disables the **Search File Types** list box.
Search Text list box	Specify the text to look for. You can type the text directly into the text box or click on the drop-down arrow to select from a previous entry in the list. The search text you enter is automatically saved in the list when you click on the **Search** button. The list box saves up to ten search queries. Search queries added after that overwrite the oldest query in the list.
Search File Types list box	Limit the search to specific types of files. You can select multiple file types in a string, separated by either a comma or a semi-colon as shown in the image above.
Search button	Begin the search. During the course of the search all other buttons in the toolbar are disabled. You can cancel the search at any time by clicking on the **Search** button again. If you switch any of the toggle buttons below, you must run the search again to change the output.
Case Sensitivity button	Toggle the case sensitivity of the search. The tooltip message identifies the current status of the button.
Word Match button	Toggle between searching for any match and searching for only those matches that form an entire word. The tooltip message identifies the current status of the button.
SubFolders button	Toggle between limiting the search to a single path and including all subfolders under that path. The tooltip message identifies the current status of the button.
Result View button	Select the presentation format of the search results - List View or Tree View format.
Clear Results button	Clear the results.
Clear Search Criteria button	Remove all the entries in the **Search Path**, **Search Text** and **Search File Types** list boxes, if required.
Help button	Display this Help topic.

3.8 Testing Command

This section describes how to create a command for performing unit testing 53 on your code.

3.8.1 Add Testing Command

This topic explains how you enter a command for performing unit testing on your code.

The command is entered in the text box using the standard *Windows Command Line* commands. A sample script would contain a line to execute the testing tool of your choice, with the filename of the executable produced by the **Build** command as the option. To execute this test select the **Project | Execution Analyzer | Test** menu option.

Testing could be integrated with any test tool using the command line provided, but in these examples you can see how to integrate *NUnit* and *JUnit* testing with your source code. Enterprise Architect provides an inbuilt

MDA Transform from source to Test Case, plus the ability to capture *xUnit* output and use it to go directly to a test failure. xUnit integration with your model is now a powerful means of delivering solid and well-tested code as part of the complete model-build-test-execute-deploy life-cycle.

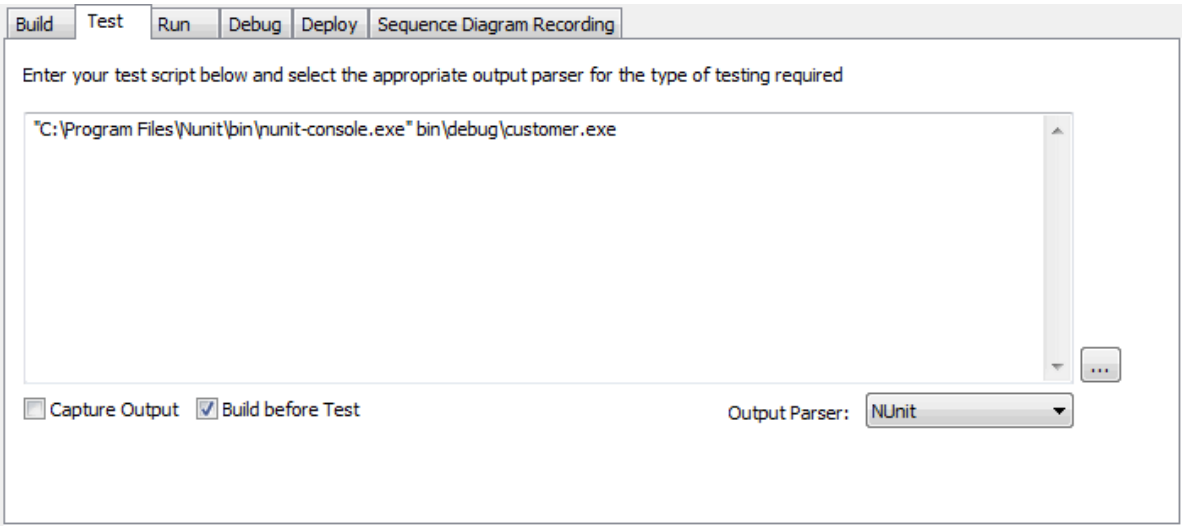
Note:

NUnit and JUnit must be downloaded and installed prior to their use. Enterprise Architect does not include these products in the base installer.

The **Capture Output** checkbox enables Enterprise Architect to show the output of the program in the Output window, while the **Output Parser** field specifies what format output is expected. When parsing is enabled, double-clicking on a result in the Output window opens the corresponding code segment in Enterprise Architect's code window.

Selecting the **Build before Test** checkbox ensures that the package is recompiled each time you run the test.

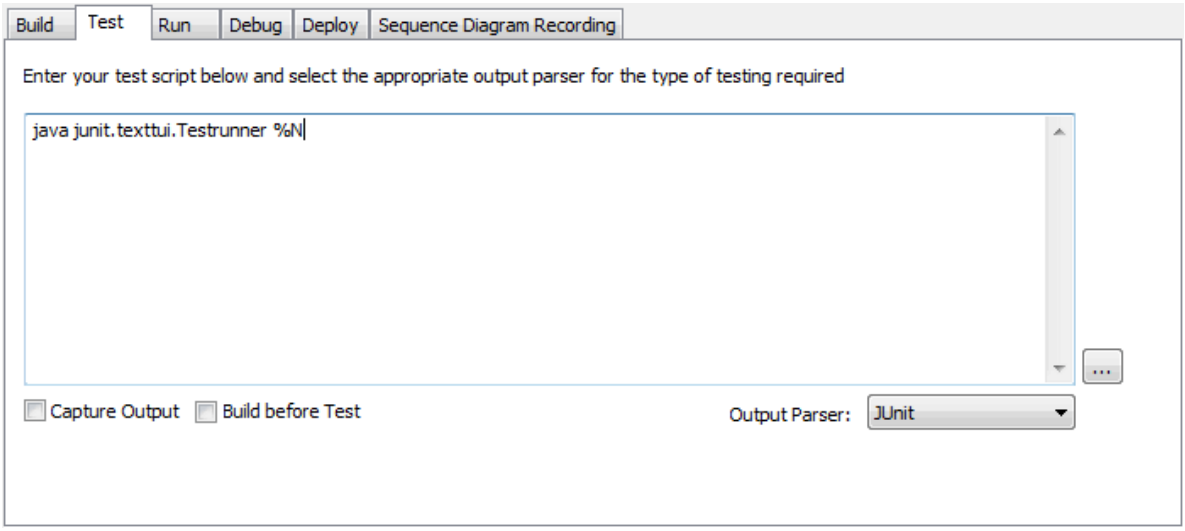
Two example test scripts are included below. The first is an NUnit example that shows the **Build before Test** checkbox selected. As a result, every time the test command is given it runs the build script first.



Note:

The command listed in this field is executed as if from the command prompt. As a result, if the executable path or any arguments contain spaces, they must be surrounded in quotes.

The second example is for JUnit. It doesn't have the **Build before Test** checkbox selected, so the build script won't be executed before every test, but as a result it could test out of date code. This also shows the use of *%N,* which is replaced by the fully namespace-qualified name of the currently selected Class when the script is executed.



3.9 Run Command

This section describes how to create a command for running 55 your executable code.

3.9.1 Add Run Command

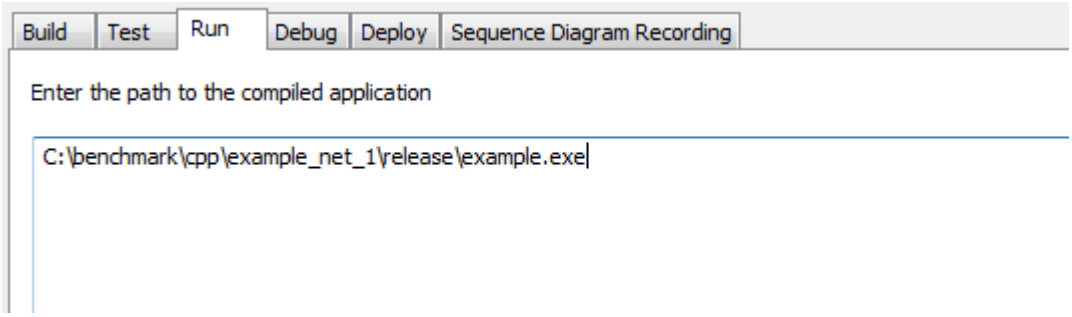
This topic explains how you enter a command for running your executable.

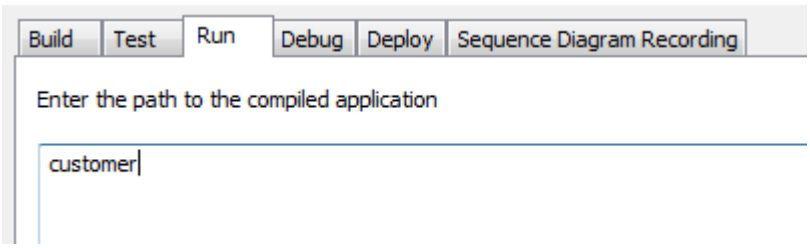
This is the command that is executed when you select the **Project | Execution Analyzer | Run** menu option. At its simplest, the script would contain the location and name of the file to be run.

Note:

Enterprise Architect provides the ability to start your application normally OR with debugging from the same script. The **Execution Analyzer** menu has separate options for starting a normal run and a debug run.

The following two examples show scripts configured to run a .Net and a Java application in Enterprise Architect.





Note:

The command listed in this field is executed as if from the command prompt. As a result, if the executable path or any arguments contain spaces, they must be surrounded in quotes.

3.10 Deploy Command

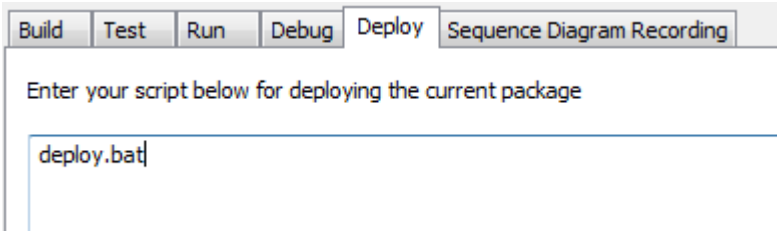
This section describes how to create a command for deploying 55 the current package.

3.10.1 Add Deploy Command

This topic enables you to create a command for deploying the current package.

These are the commands that are executed when you select the **Project | Execution Analyzer | Deploy** menu option.

Write your script in the large text box using the standard *Windows Command Line* commands.



4 Execution Analysis

This section describes the Visual Analysis of executing applications by recording application execution and generating:

- Sequence Diagrams
- Sequence/State Diagrams
- Profile (execution) Reports

Execution analysis is configured by creating a debug script 15 for the packages to be tested. One of the primary objectives of this feature is to enable you to perform a debug walk-through executing code, and capture your stack trace for direct conversion into a Sequence diagram. This is a great way to document and understand what your program is doing during its execution phase.

Execution Analysis debugging and recording are supported for the following platforms / languages:

- Microsoft Windows Native C
- Microsoft Windows Native C++
- Microsoft Windows Visual Basic
- Microsoft .NET Family (C#, J#, VB)
- Sun Microsystems Java.

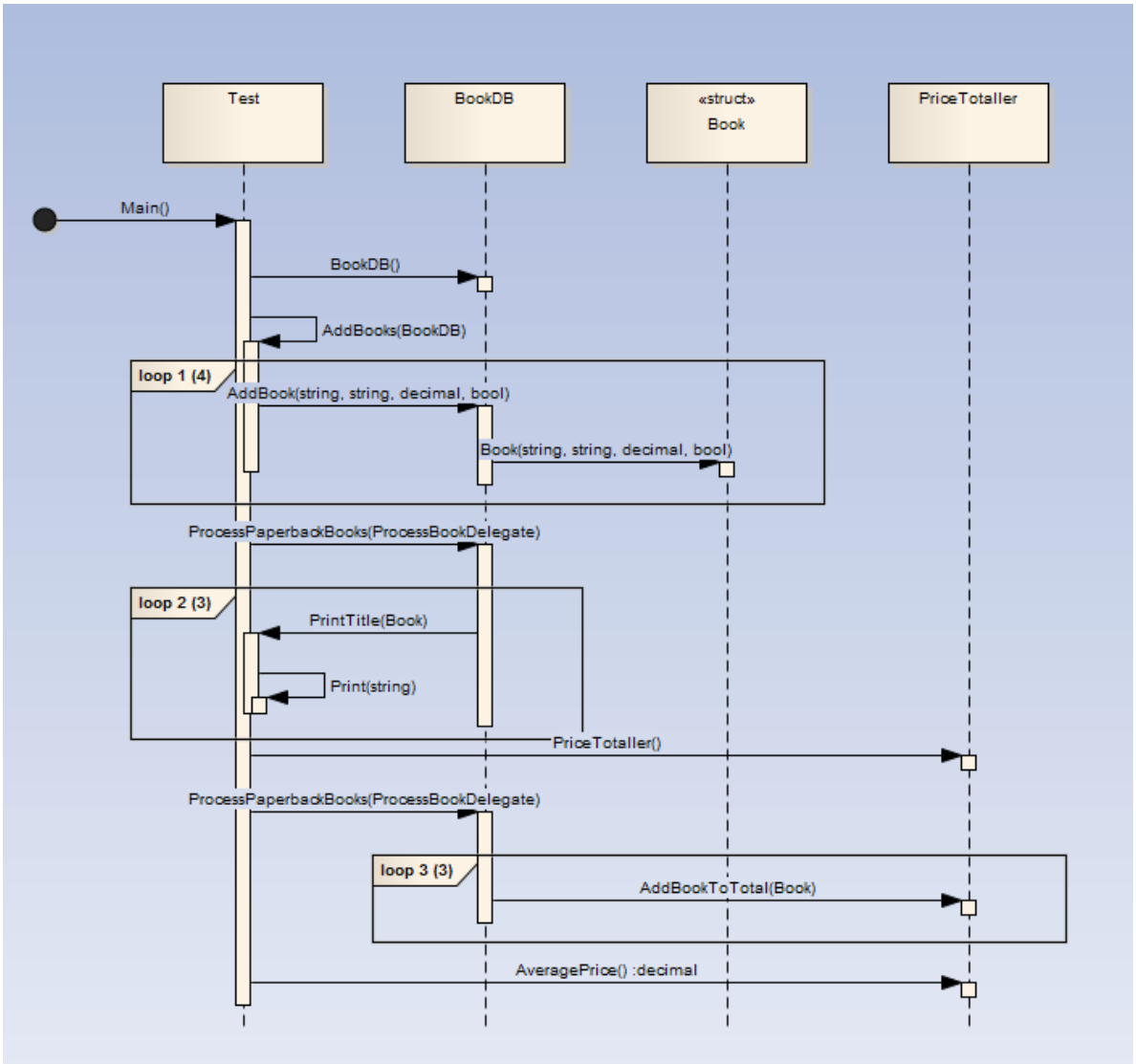
4.1 Recording Sequence Diagrams

This section explains how to use the Visual Execution Analyzer to record execution data in the form of a Sequence Diagram. It covers:

- An overview of how the process works 57
- Setup for recording 59
- Placing recording markers 66
- Controlling the recording session 72
- Generating Sequence diagrams 74
- Adding State Transitions. 75

4.1.1 How it Works

The Visual Execution Analyzer enables you to generate a Sequence Diagram. The diagram below illustrates the output of a Sequence Diagram for a program that calculates the price of books. The diagram creates a visual representation of the execution of an application, outlining what functions are being called, types of messages being sent, key data structure used and the relationships between different classes. The diagram makes it much simpler to Understand how information is moved throughout the system and what values are being passed by various functions. The first loop structure is executed four times and is being used to add four books to the book database. The arrows indicate information flow and demonstrate the change of states over time.



A Sequence diagram provides easy to understand visual information including:

- An understanding of how information is passed throughout a system.
- The sequence of various functions and their corresponding parameters.
- A clear understanding of how different classes interact to create behavior.
- A visual overview of how data structures are used to produce results.

If an application crashes, data corruption such as a stack overflow can prevent you from diagnosing and rectifying the problem. However the Visual Execution Analyzer allows you to record a given execution sequence and provide a reliable source of information that may further explain why a crash occurred. Enterprise Architect can record arguments to functions, record calls to external modules or capture state transitions based on any given constraint. This information can be integrated with existing system knowledge and test data to optimize code execution, reduce errors and understand why application failure and system crashes occur.

A Sequence Diagram extends traditional analysis to help identify errors in logic, explain unexpected system behavior and identify data flow inconsistencies. The Visual Execution Analyzer extends analysis through the use of a comprehensive array of reports that detail everything from state transitions through to the contents of the stack at a given time. A Sequence Diagram can convey more detail and provide greater understanding than reading unfamiliar code that has potentially been written by someone else. It also makes it easier to document existing code when a Sequence Diagram illustrates functions are being called and the specific sequence of events that occur to produce a particular type of system behavior.

4.1.2 Setup for Recording

This section explains how you prepare to record execution of the application. It covers:

- Prerequisites 59
- Configuring Recording Detail 59
- Advanced Techniques 64

4.1.2.1 Pre-Requisites

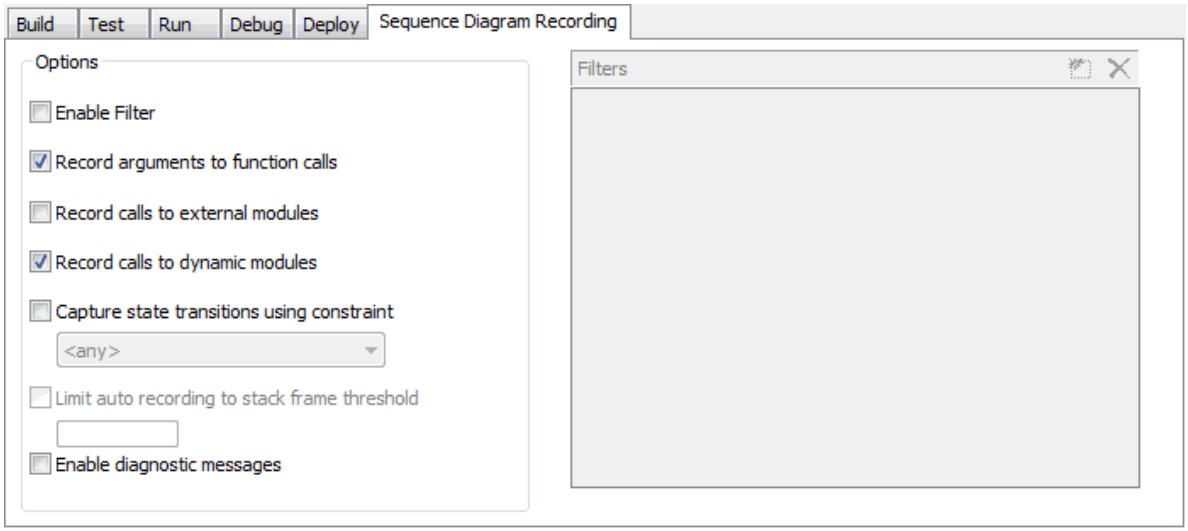
Recording is available to users of Enterprise Architect Professional and above.

Basic setup 8 must be completed.

You should first be able to successfully debug 15 the application.

4.1.2.2 Configure Recording Detail

The Sequence Diagram Recording tab enables you to set various options for generating Sequence diagrams from the debugger.



These options are not all available for each platform, as indicated in the following table:

Option	.NET	Java	Native
Enable Filter 60	X	X	X
Record arguments to function calls 61	X	X	X
Record calls to external modules 61	X	X	X
Record calls to dynamic modules 62	X	-	-
Capture state transitions using constraint 75	X	X	X
Limit auto recording to stack frame threshold 63	X	X	X
Enable diagnostic messages 64	X	X	X

4.1.2.2.1 Enable Filter

If the **Enable Filter** option is selected on the Sequence Diagram Recording tab, the debugger excludes calls to matching methods from the generated sequence history and diagram. The comparison is case-sensitive.

To add a value, click on the **New** (Insert) icon in the right corner of the **Filters** box, and type in the comparison string. Each filter string takes the form:

```
class_name_token::method_name_token
```

The *class_name_token* excludes calls to all methods of a Class or Classes having a name that matches the token. The string can contain the wildcard character * (asterisk). The token is optional.

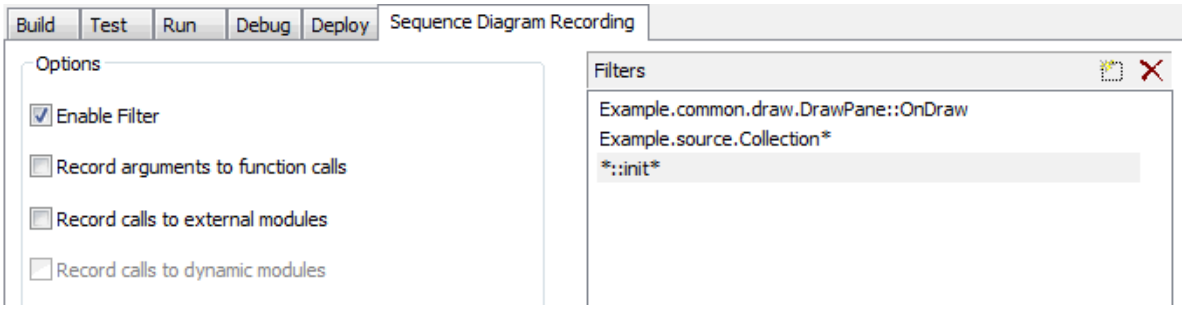
The *method_name_token* excludes calls to methods having a name that matches token. The string can contain the wildcard character *. The token is optional.

Where no Class token is present, the filter is applied only to global or public functions; that is, methods not belonging to any Class.

To Filter	Use Filter Entry
All public functions having a name beginning with **Get** from the recording session (*GetClientRect* for example in Windows API).	::Get*
All methods beginning with **Get** for every Class member method.	*::Get*
All methods beginning with **Get** from the Class *CClass.*	CClass::Get*
All methods for Class *CClass*.	CClass::*
All methods for Classes belonging to Standard Template and Active Template Libraries.	• ATL* • std*
The specific method *GetName* for Class *CClass.*	CClass::GetName

In the Java example in the screen below, the debugger would exclude:

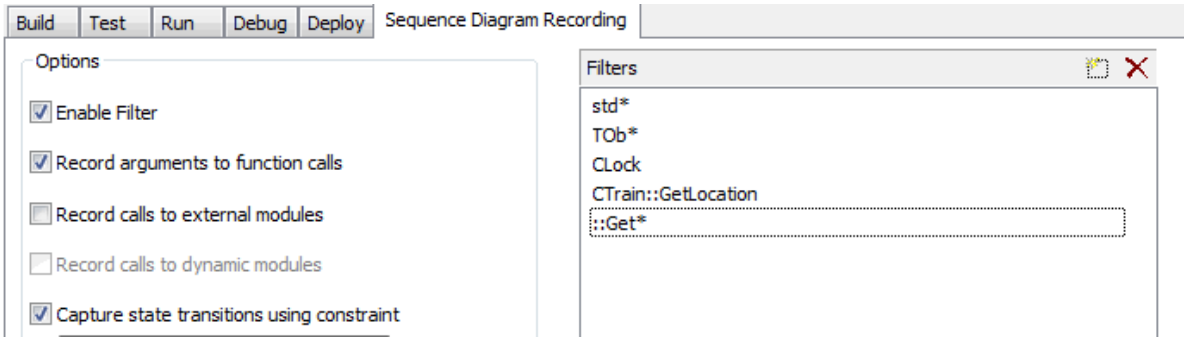
- Calls to *OnDraw* method for Class *Example.common.draw.DrawPane*
- Calls to any method of any Class having a name beginning with *Example.source.Collection*
- Calls to any constructor for any Class (ie: <*clint*> and <*init*>).



In the Native code example below, the debugger would exclude:

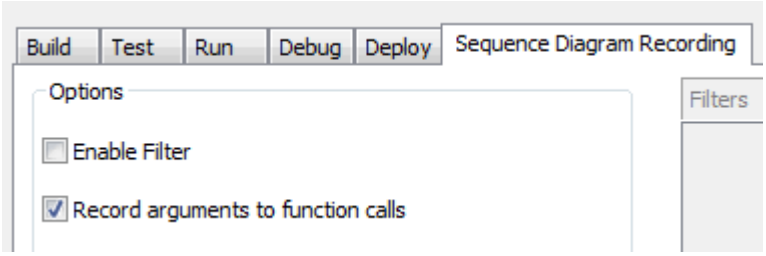
- Calls made to Standard Template Library namespace
- Calls to any Class beginning with **TOb**
- Calls to any method of Class *CLock*

- Calls to any Global or Public Function with a name beginning with **Get**
- Calls to the method *GetLocation* for Class *Ctrain.*

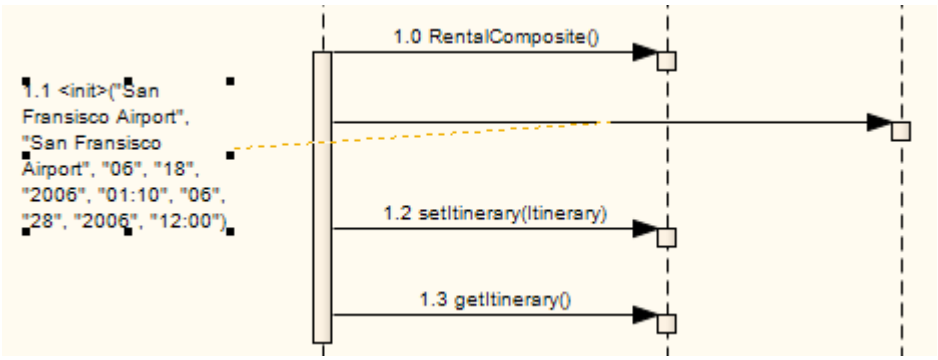


4.1.2.2.2 Record Arguments To Function Calls

When recording the sequence history, Enterprise Architect can record the arguments passed to method calls.



When the **Record Arguments to function calls** option is selected on the Build Script dialog Sequence Diagram Recording tab, the resulting Sequence diagram shows the values of elemental and string types passed to the method. See the following Java example.

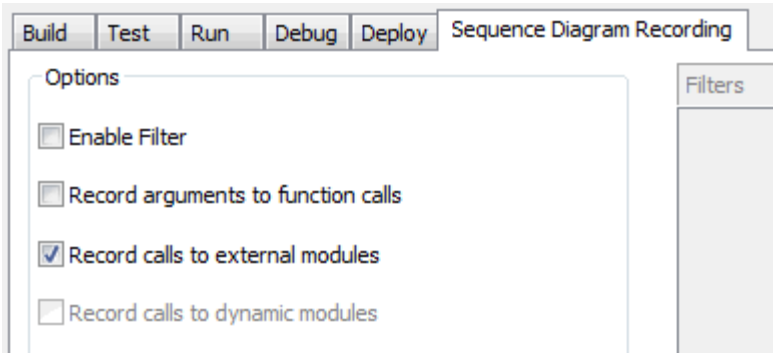


Where the argument is not an elemental type, the type name is recorded instead.

4.1.2.2.3 Record Calls To External Modules

On the Sequence Diagram Recording tab, the **Record calls to external modules** option causes function calls to external modules outside the model to be included in the sequence history and generated diagram.

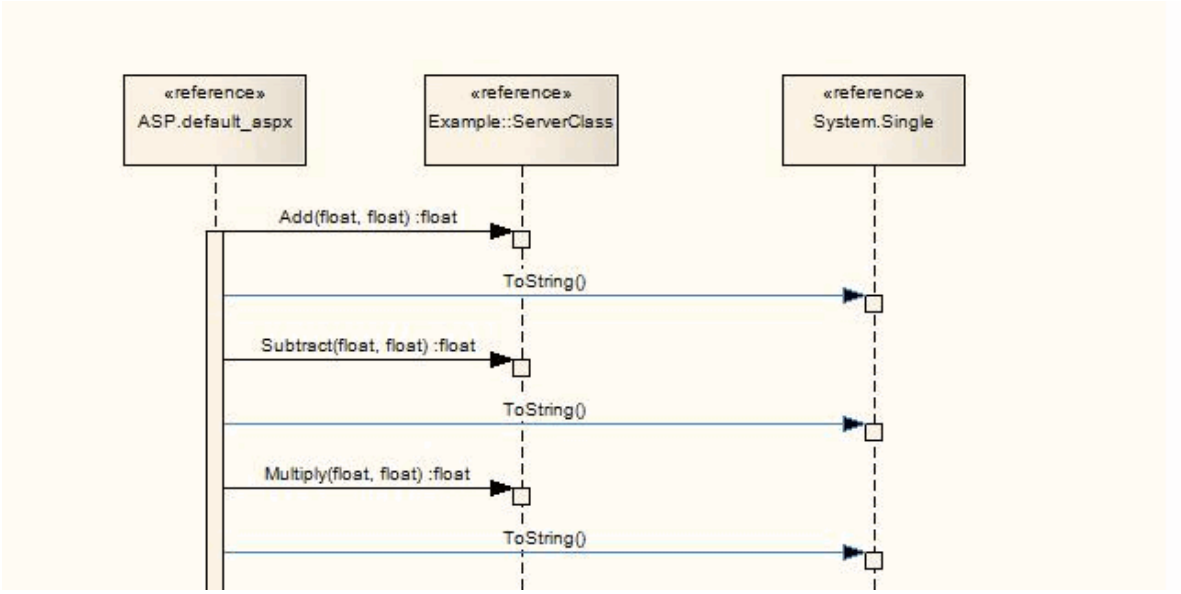
For applications built in a Microsoft Native code (C, C++) you can record calls to the WIN32 API if required, using the **Record calls to external modules** option. This option can also be used to record calls to functions in modules that have a PDB file but for which there is no source.



Only calls originating within the model to functions external to the model are recorded.

Note:

External calls are displayed with a blue connector, as shown below.

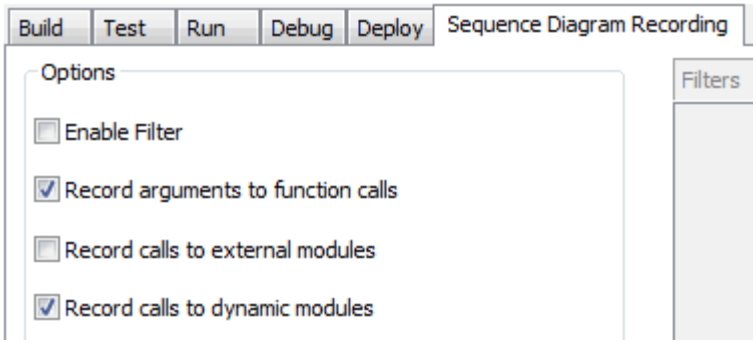


This example shows three external calls (*ToString()*) to the Microsoft .NET framework assembly function *System.Single*.

4.1.2.2.4 Record Calls to Dynamic Modules

(Available only for .NET platforms.)

On the Sequence Diagram Recording tab, the **Record calls to dynamic modules** option causes the debugger to record execution of dynamic or 'In Memory' function calls, in transitions between normal assemblies and those emitted dynamically.



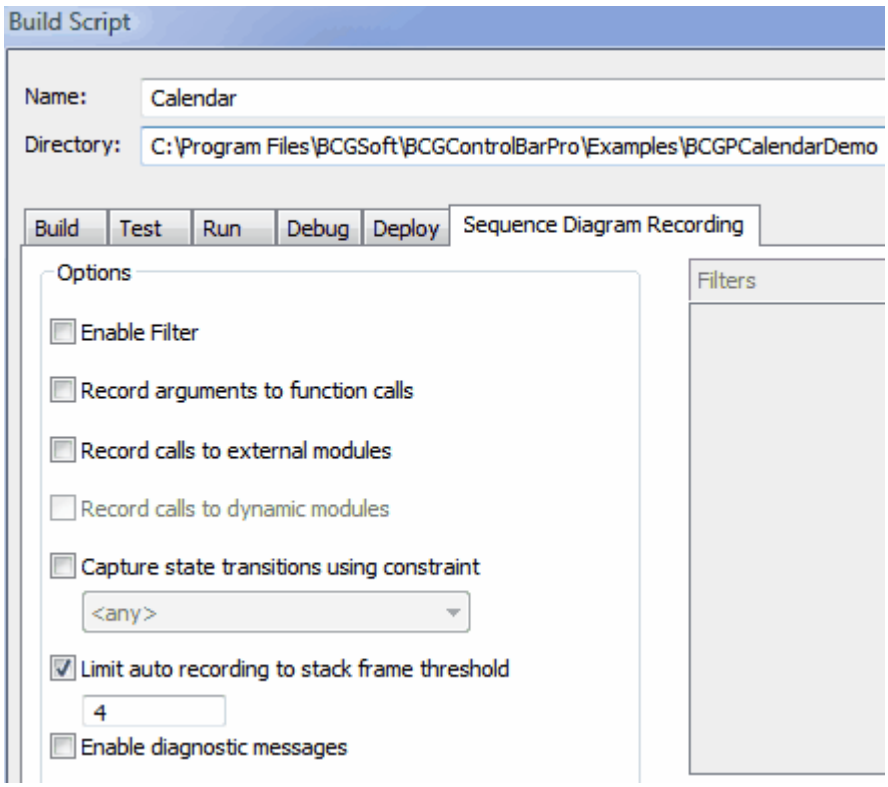
4.1.2.2.5 Limit Auto Recording

Where the Stack window shows recording to be involved in function calls that are not particularly useful, and that are not being excluded in a filter, you can achieve a quicker and more general picture of a sequence by limiting the stack depth being recorded. You can do this on the Sequence Diagram Recording tab, by selecting the **Limit auto recording to stack frame threshold:** option.

If you use this option, be aware that the threshold value you set is a relative frame count; that is, the count is relative to the frame at which recording begins. For example:

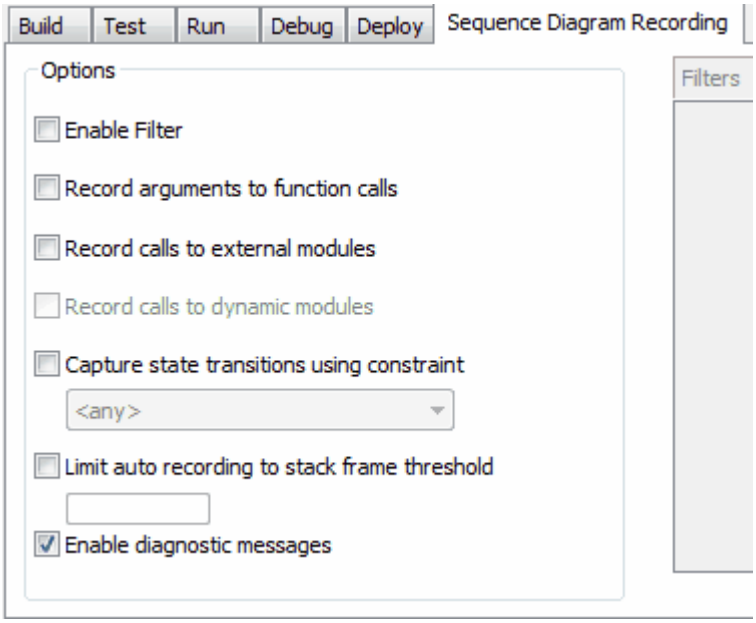
> A breakpoint has occurred, and the Stack window shows five frames. If the stack frame threshold is set to 3 and you begin auto-recording at this breakpoint, the debugger records all function calls between the current frame 5 and a maximum stack frame depth of 8 inclusive.

For situations during auto-recording where the stack is very large, it is recommended that you first use a low stack frame threshold of 2 or 3, gradually increasing it if necessary to expand the picture. You can also use the threshold to work out which filters you could add to the script in order to further clarify the Sequence diagram that is ultimately produced.



4.1.2.2.6 Enable Diagnostic Messages

The **Enable diagnostic messages** checkbox triggers the debugger to output more self-reporting, diagnostic messages as it executes. For example, the debugger might output messages about method calls that are being excluded from the recording history due to a filter also having been set in the Sequence Diagram Recording tab of the Build Script dialog.



4.1.2.3 Advanced Techniques

This section describes the advanced techniques for configuring recording detail:

- Recording Activity for a Class 64
- Recording Activity for a single method 65

4.1.2.3.1 Recording Activity for a Class

In addition to setting breakpoints and markers in the code editor, you record all the operations of a class, or a subset by using the *Class Markup* Feature.

This feature is available from the Project Browser context menu while on a Class. Select the operations to record, choose the marker type and enter a name for the set. When you click on the **OK** button the markers are stored as a marker set using the name you specify.

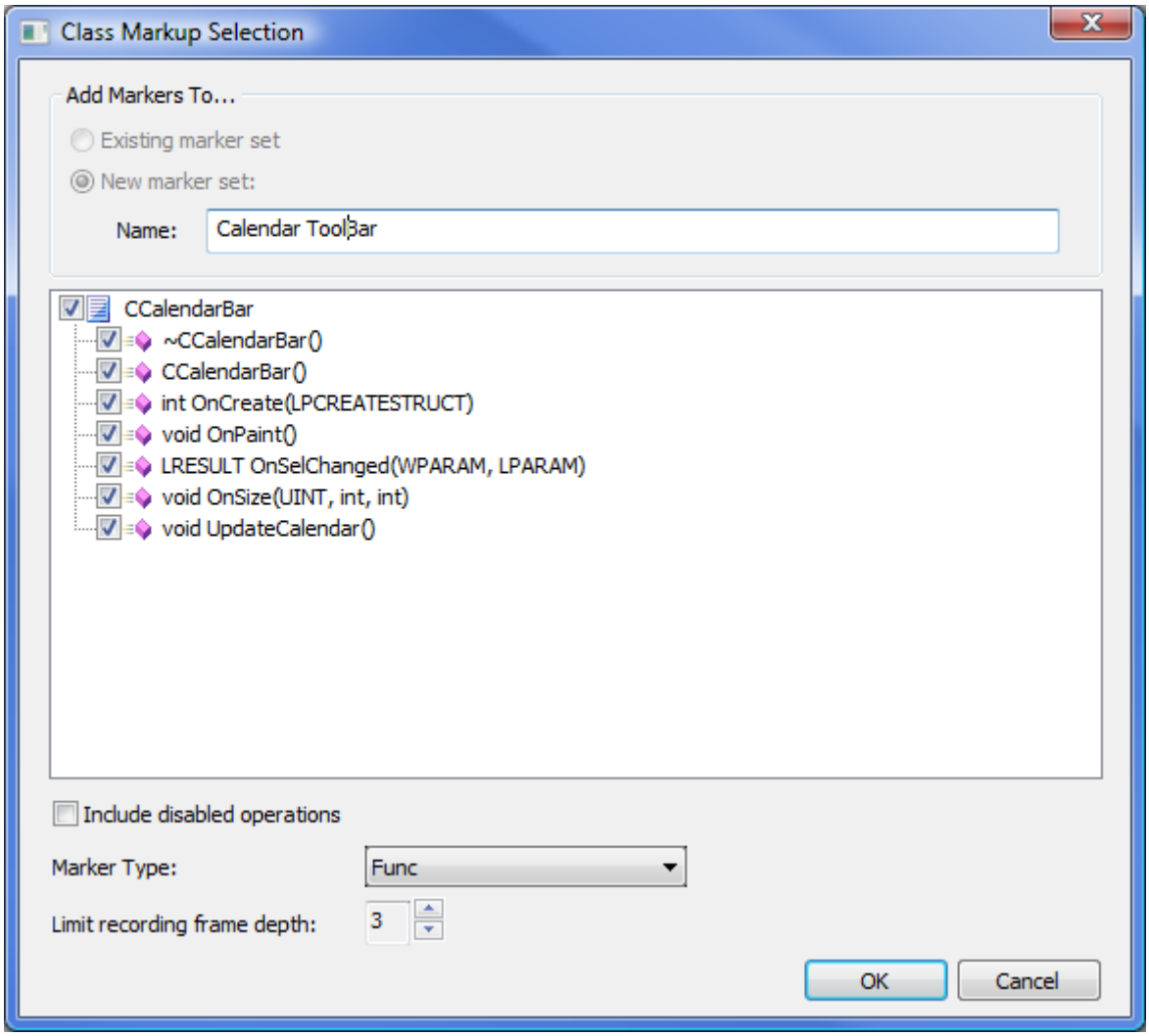
This set can then be loaded either before or during a session.

The marker type specifies the action to take when the process encounters that marker.

- Record function
- Record Stack Trace
- Break execution

You can also specify a recording depth. This limits the recording, which if uncontrolled can ultimately produce Sequence Diagrams that are too complicated to read. When you specify a depth, the Debugger does not record beyond this depth.

The depth is relative to the stack depth where the Debugger first encountered the recording marker. So, if the stack depth is 7 when recording begins, and the Limit Depth is set to 3, the Debugger does not record beyond a Stack depth of 10.



4.1.2.3.2 Recording Activity for a Single Method

A *Method Auto Record* 66 marker enables you to record activity for a particular function during a debug session. The debugger records any function calls executed after the marker point, and always stops recording when this function exits. The function marker combines a Start Recording marker and an End Recording marker in one.

```
185 ////////////////////////////////////////////////////////////////////////////
186 // CRecurrenceDlg message handlers
187
188 BOOL CRecurrenceDlg::OnInitDialog()
189 {
190     CBCGPDialog::OnInitDialog();
191
192     UINT nMask =
193         CBCGPDateTimeCtrl::DTM_SPIN         |
194         CBCGPDateTimeCtrl::DTM_DATE         |
195         CBCGPDateTimeCtrl::DTM_TIME         |
196         CBCGPDateTimeCtrl::DTM_CHECKBOX     |
197         CBCGPDateTimeCtrl::DTM_DROPCALENDAR |
198         CBCGPDateTimeCtrl::DTM_CHECKED;
199
200     UINT nFlags = CBCGPDateTimeCtrl::DTM_CHECKED | CBCGPDateTimeCtrl::DI
201     //--------------------
202     // Setup date fields:
```

4.1.3 Place Recording Markers

This section explains how to deploy recording markers:

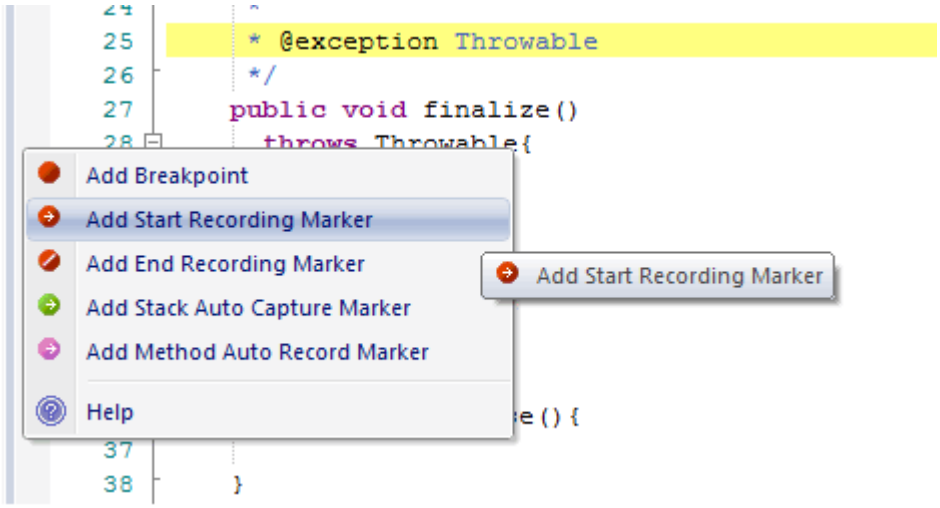
- Marker types 66
- Setting Recording Markers 70
- The Breakpoint and Markers window 71
- Activate and Disable Markers 71
- Working with Marker Sets 72
- Differences between breakpoints and markers. 72

4.1.3.1 Marker Types

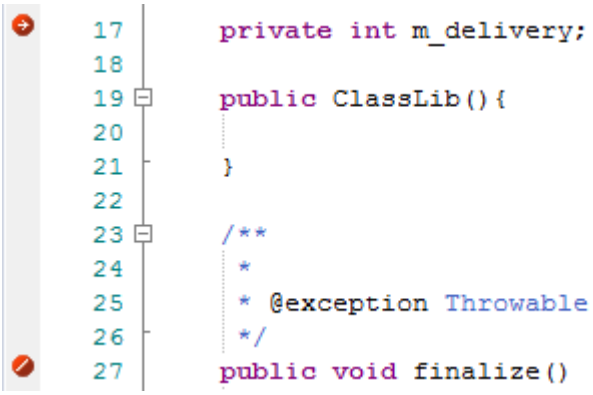
Trace marking is a feature that enables you to silently record code executed between two points, and incorporate it in a Sequence diagram. The feature also enables you to capture the execution of multiple threads. It can be particularly useful in capturing event driven sequences (such as mouse and timer events) without any user intervention.

The recording markers are breakpoints; however, instead of stopping, the debugger behaves according to the type of marker. If the marker is denoted as a recording *start point*, the debugger immediately begins to trace all executed calls from that point for the breaking thread. Recording is stopped again when either the thread that is being captured terminates or the thread encounters a *recording end point*.

Recording markers are set in the source code editor. If you right-click on the breakpoint margin at the point to begin recording, a context menu displays:

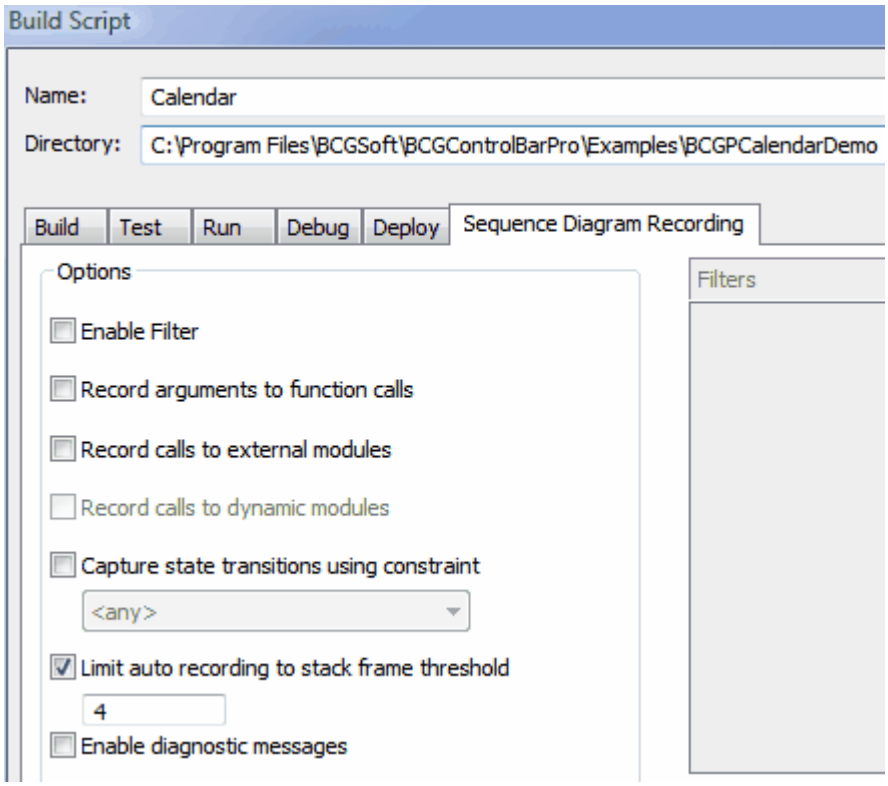


Select the **Add Start Recording Marker** option, then right-click on the breakpoint margin at the point to stop recording and select the **Add End Recording Marker** context menu option. The markers are shown below:



When the debugger is run it continues to run the thread, recording a stack history, until either the **End Recording** marker is encountered or the thread terminates, unlike normal breakpoints where the debugger halts and displays the line of code.

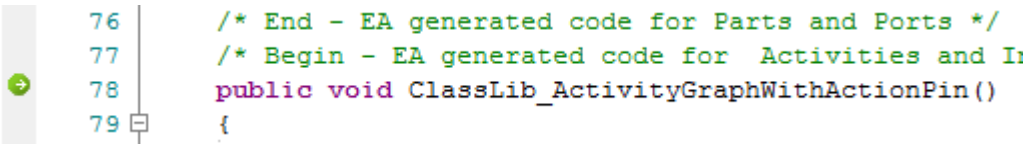
It is useful to limit the stack depth 63 when recording particularly high-level points in an application, as the stack frame count can result in too much information being collected. You can limit stack depth using the Sequence Diagram Recording tab on the Build Script dialog.



Running this Calendar example with the one function record marker in *CRecurrenceDlg::OnInitDialog()* produced the following output in the Recording History window:

Sequence	Insta...	Method	Direction	Method
00000001			Call	CRecurrenceDlg.OnInitDialog
00000002		CRecurrenceDlg.OnInitDialog	Call	CBCGPDialog.OnInitDialog
00000003		CBCGPDialog.OnInitDialog	Call	CBCGPDialog.IsVisualManagerStyle
00000004			Return	CBCGPDialog.OnInitDialog
00000005		CBCGPDialog.OnInitDialog	Call	CBCGPDialog.IsVisualManagerNCArea
00000006			Return	CBCGPDialog.OnInitDialog
00000007		CBCGPDialog.OnInitDialog	Call	CBCGPDlgImpl.EnableVisualManagerStyle
00000008		CBCGPDlgImpl.EnableVisualManagerStyle	Call	CBCGPButton.GetThisClass
00000009			Return	CBCGPDlgImpl.EnableVisualManagerStyle
00000010		CBCGPDlgImpl.EnableVisualManagerStyle	Call	ATL.operator==
00000011			Return	CBCGPDlgImpl.EnableVisualManagerStyle
00000012		CBCGPDlgImpl.EnableVisualManagerStyle	Call	ATL.operator==
00000013			Return	CBCGPDlgImpl.EnableVisualManagerStyle
00000014		CBCGPDlgImpl.EnableVisualManagerStyle	Call	CBCGPGroup.CBCGPGroup

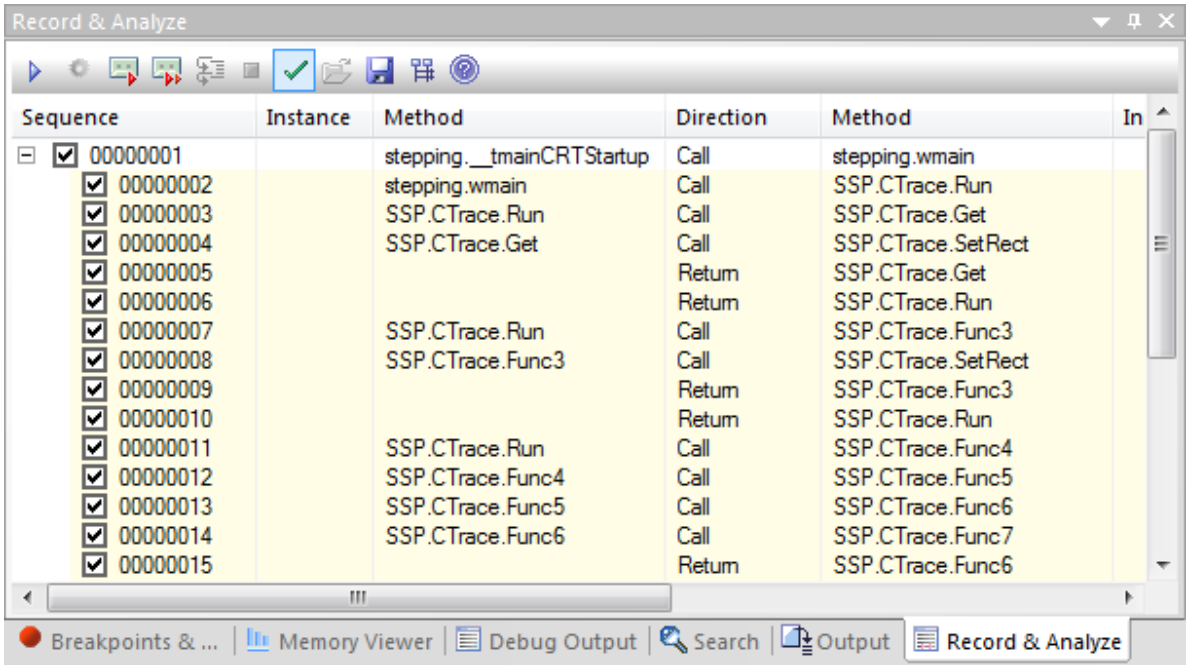
Stack Auto-Capture Marker

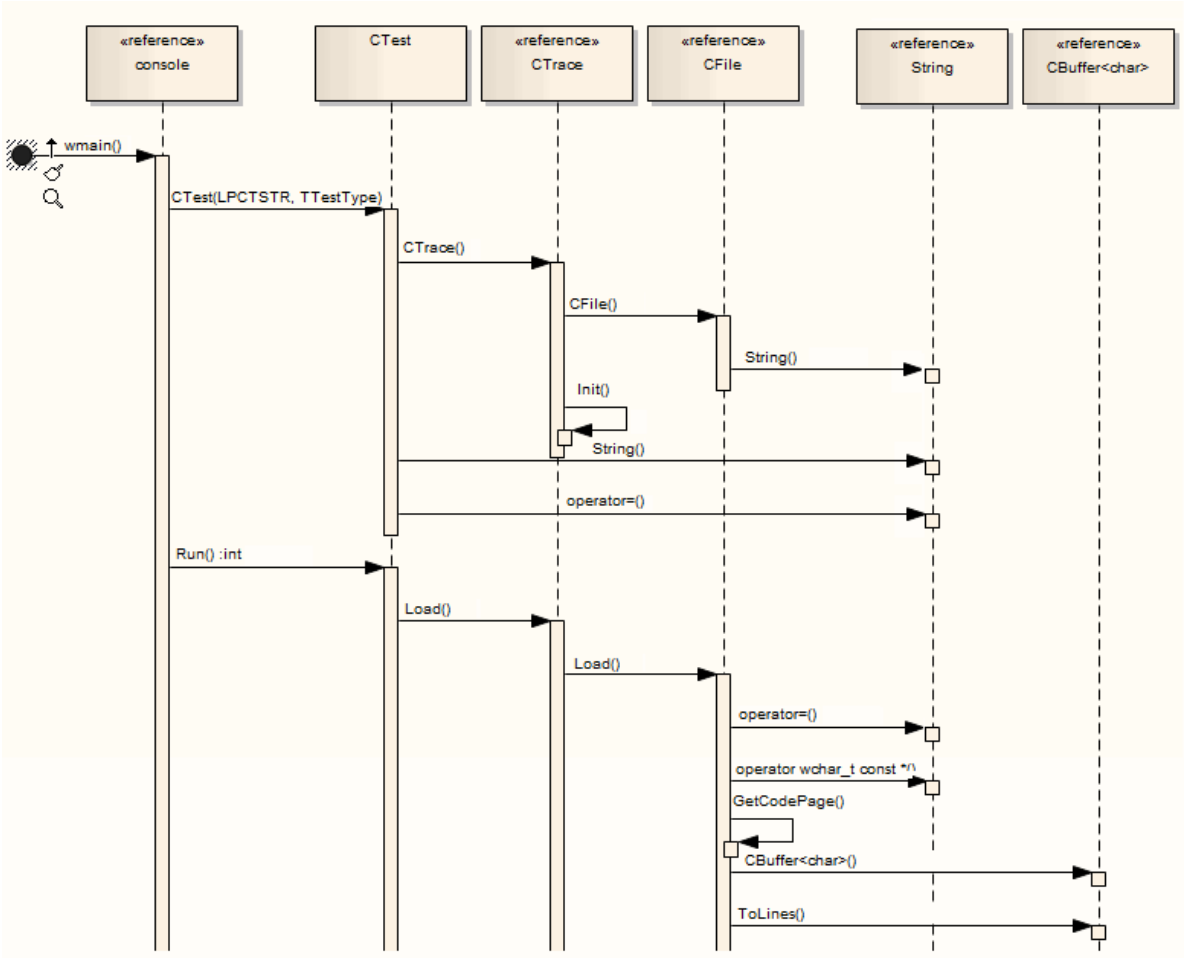


(Native Code only.) Stack markers enable you to capture any unique stack traces that occur at a point in an application. To insert a marker at the required point in code, right-click on the line and select the **Add Stack Auto Capture Marker** context menu option.

Each time the debugger encounters the marker it performs a stack trace. If the stack trace is not in the recording history, it is copied. The application then continues running. Stack markers provide a quick and

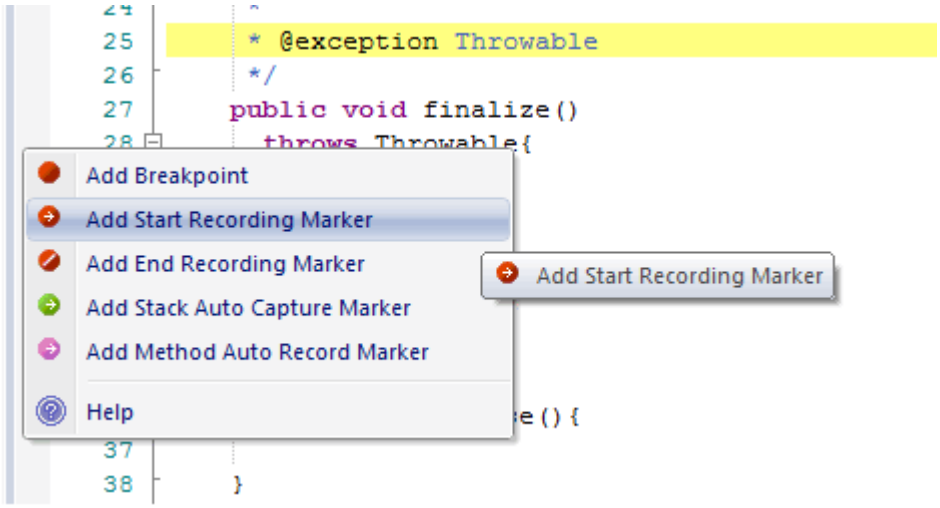
useful picture of where a point in an application is being called from.





4.1.3.2 Setting Recording Markers

Recording markers are set in the source code editor. If you right-click on the breakpoint margin at the point to begin recording, a context menu displays:



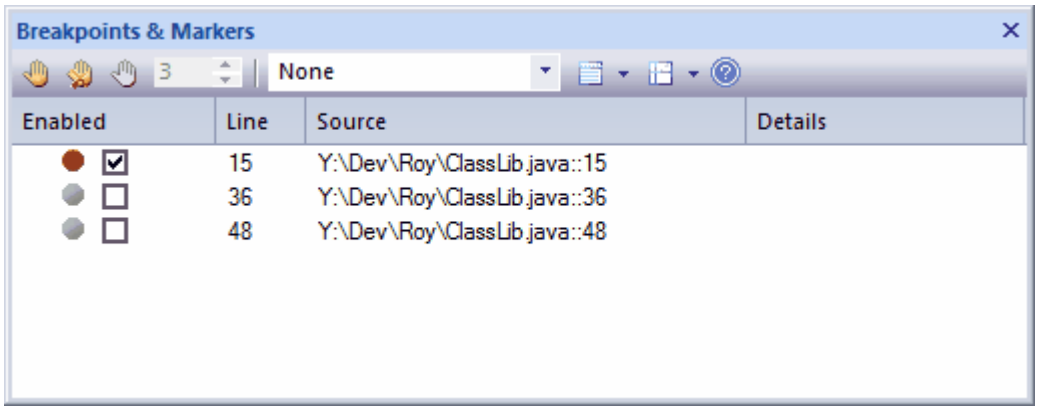
Select the **Add Start Recording Marker** option, then right-click on the breakpoint margin at the point to stop recording and select the **Add End Recording Marker** context menu option. The markers are shown below:

```
17      private int m_delivery;
18
19      public ClassLib(){
20
21      }
22
23      /**
24       *
25       * @exception Throwable
26       */
27      public void finalize()
```

When the debugger is run it continues to run the thread, recording a stack history, until either the **End Recording** marker is encountered or the thread terminates, unlike normal breakpoints where the debugger halts and displays the line of code.

4.1.3.3 The Breakpoints and Markers Window

The Breakpoints and Markers window allows you to manage control of the process. Here you can enable, disable, delete markers and also manage them as sets. You can organize how they are displayed, either in list view or grouped by file or class.



4.1.3.4 Activate and Disable Markers

To delete a specific breakpoint, either:

- If the breakpoint is enabled, click on the red breakpoint circle in the left margin of the Source Code Editor
- Right-click on the breakpoint marker in the editor and select the appropriate context menu option, or
- Select the breakpoint in the Breakpoints & Markers tab and press **[Delete]**.

Whether you are viewing the *Breakpoints* folder or the Breakpoints & Markers window, you can right-click on an existing breakpoint and select a context menu option either to delete it or to convert it to a start recording marker or end recording marker 66.

You can also delete all breakpoints by clicking on the **Delete all breakpoints** button on the Breakpoints & Markers window toolbar ( ).

To disable a breakpoint, deselect its checkbox on the Breakpoints & Markers window or, to disable all breakpoints, click on the **Disable all breakpoints** button in the toolbar ( ). The breakpoint is then shown as an empty grey circle. Select the checkbox or use the **Enable all breakpoints** button to enable it again ( ).

4.1.3.5 Working with Marker Sets

Marker sets enable you to group markers into collections.

A set can be used to record a specific Use Case, which might involve the operations of various Classes. Once a set is created it is saved with the Model. Any other user using the Model has access to that set.

Sets are normally loaded prior to the point at which an action is to be captured. For example, to record a sequence involving a particular dialog, you might set markers for the areas to record, saving the markers as a set. When you begin debugging, prior to invoking the dialog you would then load the set. Once you bring up the dialog in the application, the operations you have marked are recorded. Review the recording history and create a Sequence diagram.

4.1.3.6 Differences to Breakpoints

Breakpoints differ from Markers in that they always break execution whereas Markers operate silently without intervention.

4.1.4 Control the Recording Session

This section describes how you control the recording session:

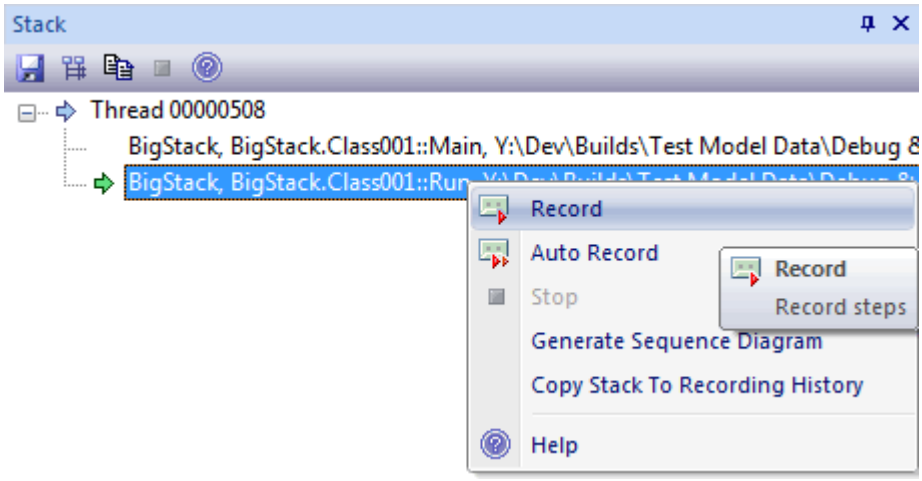
- Auto Recording 72
- Manual Recording 73
- Pause Recording 73
- Resume Recording 73
- Stop Capture. 73

4.1.4.1 Auto-Recording

Auto-Recording is available when the process being debugged is at a breakpoint.

You can use the record button on the Record & Analyze window toolbar.

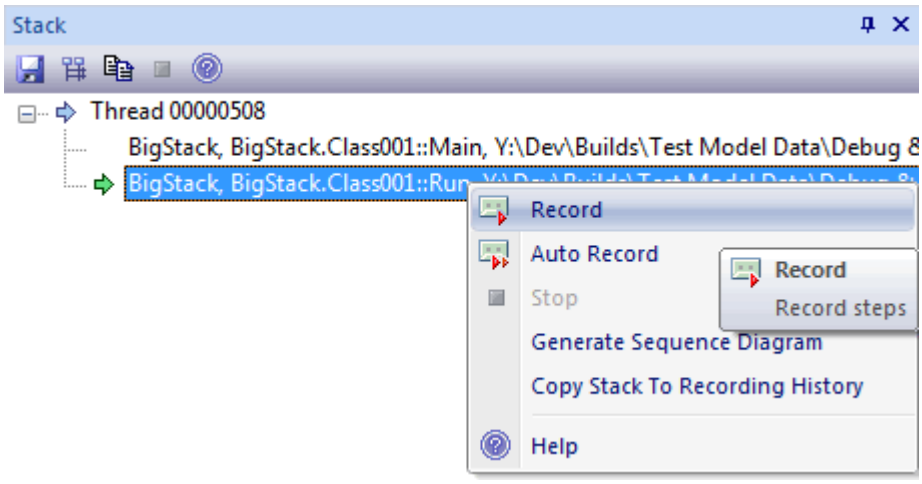
Alternatively, select the thread in the stack window:



4.1.4.2 Manual Recording

Manual Recording is available when the process being debugged is at a breakpoint.

Display the Stack window and use the context menu to switch to record mode.



Thereafter you must issue debug commands {StepIn, StepOver, StepOut, Stop} manually.

Each time you issue a step command and the thread stack changes, the sequence of execution is logged.

When you have finished tracing, click on the Stop button ().

4.1.4.3 Pause Recording

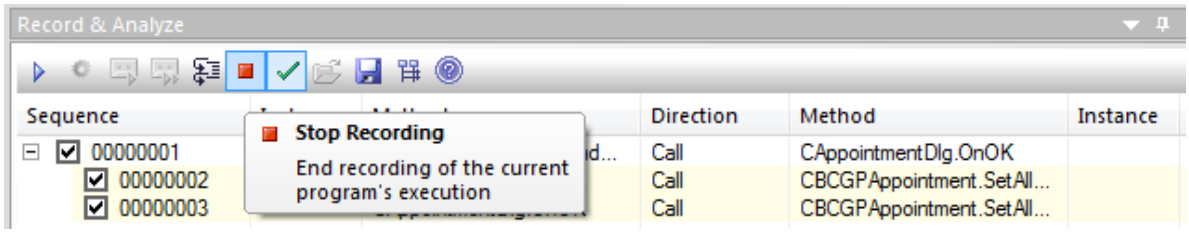
You can pause recording by using the **Pause/Resume Execution** button on the Debug window toolbar or in the Debug Management window (**[Alt]+[8]**).

4.1.4.4 Resume Recording

You can resume recording using the **Pause/Resume Execution** button on the Debug window toolbar or in the Debug Management window (**[Alt]+[8]**).

4.1.4.5 Stop Capture

To stop recording at any time click on the **Stop Recording** button on the Record & Analyze window toolbar.

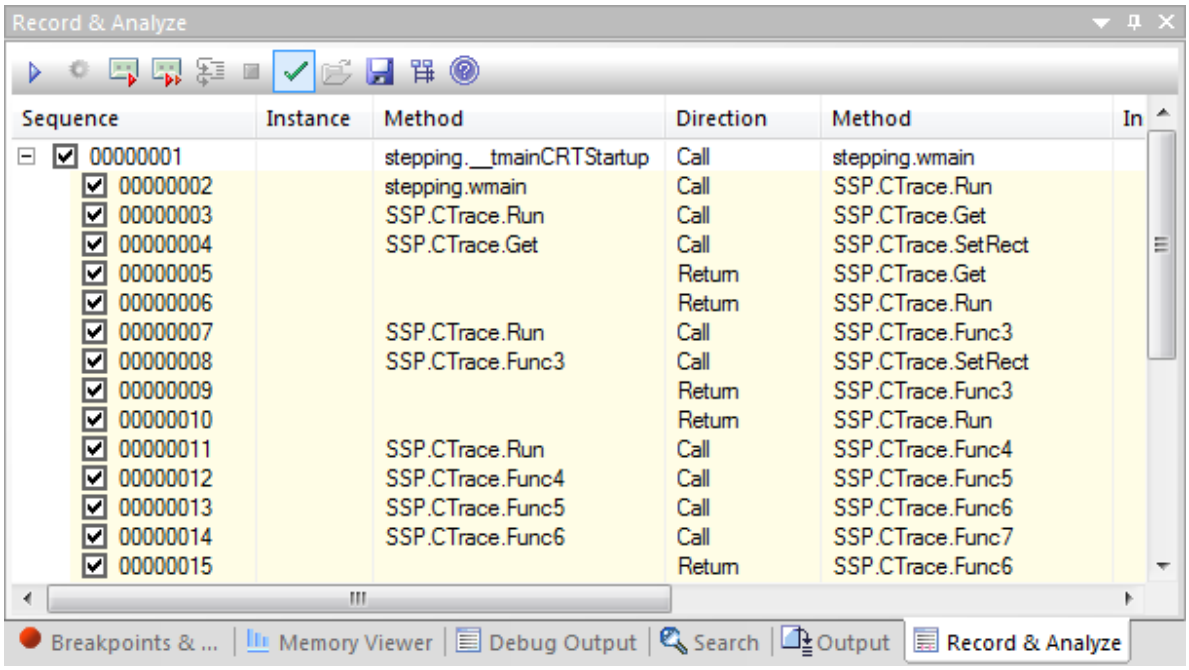


4.1.5 Generating Sequence Diagrams

Once you have captured activity and are about to generate the diagram, firstly select a package in the Project Browser where you intend the Sequence diagram to be stored. Then use the toolbar on the Record & Analyze window to generate the diagram.

4.1.5.1 The Recording History

All information recorded as a result of the application encountering recording markers set by the user is held in the Record & Analyze window.



The columns in this window are as follows:

- **Sequence** - The unique sequence number

  Note:

  The checkbox against each number is used to control whether or not this call should be used to create a Sequence diagram from this history. In addition to enabling or disabling the call using the checkbox, you can use context menu options to enable or disable an entire call, all calls to a given method, or all calls to a given Class.

- **Threads** - The operating system thread ID
- **Delta** - The elapsed thread CPU time since the start of the sequence
- **Method** - There are two **Method** columns: the first shows the caller for a call or for a current frame if a return; the second shows the function called or function returning
- **Direction -** Stack Frame Movement, can be *Call, Return, State, Breakpoint* or *Escape* (*Escape* is used internally when producing a Sequence diagram, to mark the end of an iteration)
- **Depth** - The stack depth at the time of a call; used in the generation of Sequence diagrams
- **State** - The state between sequences
- **Source** - There are two **Source** columns: the first shows the source filename and line number of the caller for a call, or for a current frame if a return; the second shows the source filename and line number of the function called or function returning.
- **Instance** - There are two **Instance** columns; these columns only have values when the Sequence diagram produced contains State transitions. The values consist of two items separated by a comma - the first item is a unique number for the instance of the Class that was captured, and the second is the actual instance of the Class.

For example: supposing a Class *CName* has an internal value of 4567 and the program created two instances of that Class; the values might be:

- 4567,1
- 4567,2

The first entry shows the first instance of the Class and the second entry shows the second instance.

4.1.5.2 Generate a Diagram

To generate a Sequence diagram for all history click on the toolbar **Create Sequence Diagram** icon ( ).

To generate a Sequence diagram for a single sequence, select it and then click the toolbar **Create Sequence Diagram** icon ( ).

4.1.5.3 Diagram Features

The Sequence diagram produced includes the following:

References

When the VEA cannot match a function call to an operation within the model, it still creates the sequence, but it creates a reference for any Class that it cannot locate. It does this for all languages.

Fragments

Fragments displayed in the Sequence diagram represent loops or iterations of a section(s) of code. The VEA does its best to match function scope with method calls to as accurately as possible represent the execution visually.

States

If a State Machine has been used during the recording process, any transitions in State are presented after the method call that caused the transition to occur. States are calculated on the return of every method to its caller.

4.1.5.4 Saving Recording

To save a sequence to an XML file, click on the sequence and on the toolbar **Save** button ( ).

To access an existing sequence file, either:

- Click on the toolbar **Open** icon ( ), or
- Right-click on a blank area of the screen and click on the **Load Sequence From File** context menu option.

The Windows Open dialog displays, from which you select the file to open.

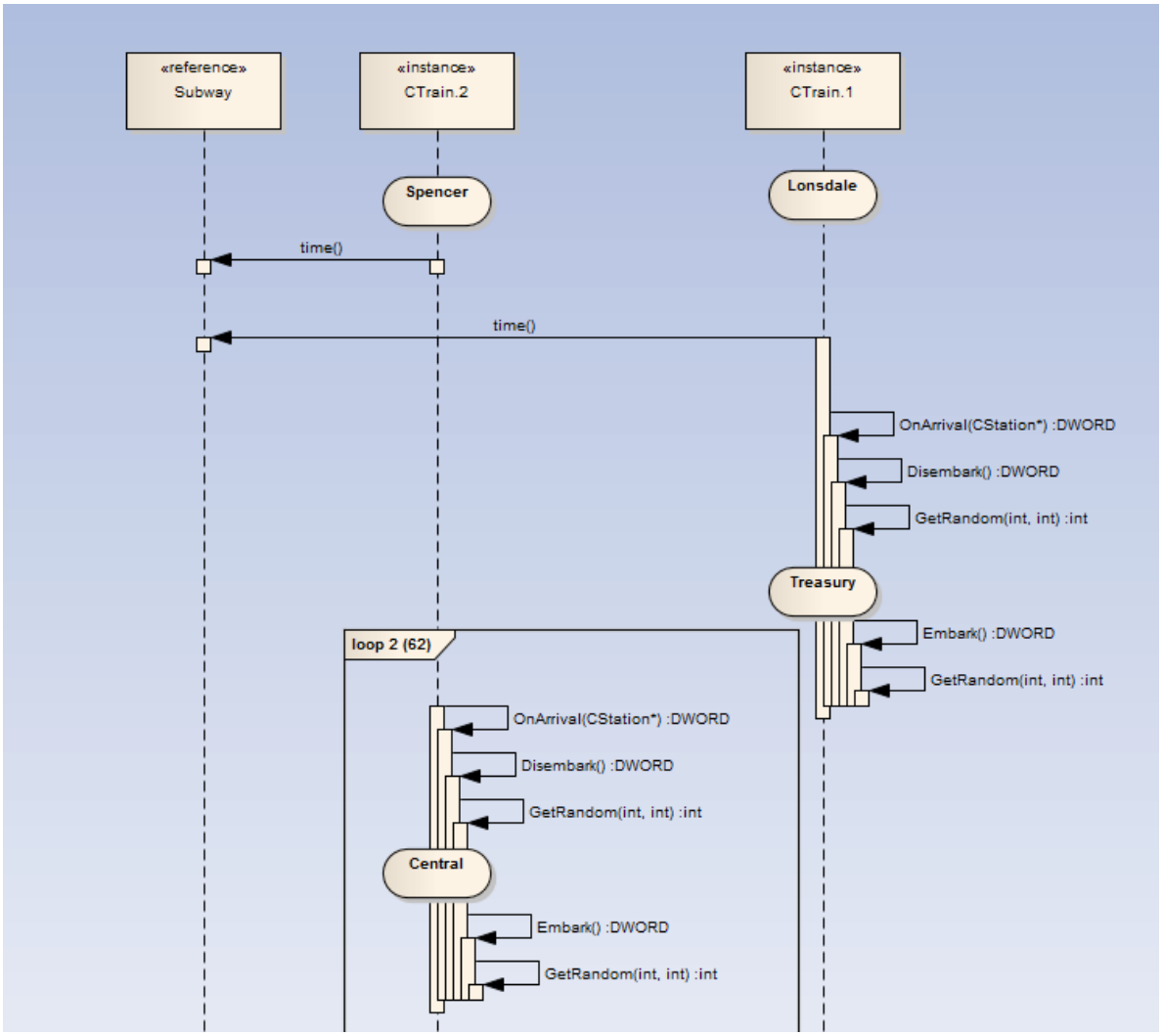
4.1.6 Add State Transitions

This topic describes how to add State Transitions. It covers:

- Setup for Capturing State Changes 76
- The State Machine 77
- Recording and Mapping State Changes. 78

4.1.6.1 Setup for Capturing State Changes

You can generate Sequence diagrams that show transitions in state as a program executes. The illustration below shows a project that has, in its State Machine, a number of States that correspond to stations in the Melbourne underground railway system.

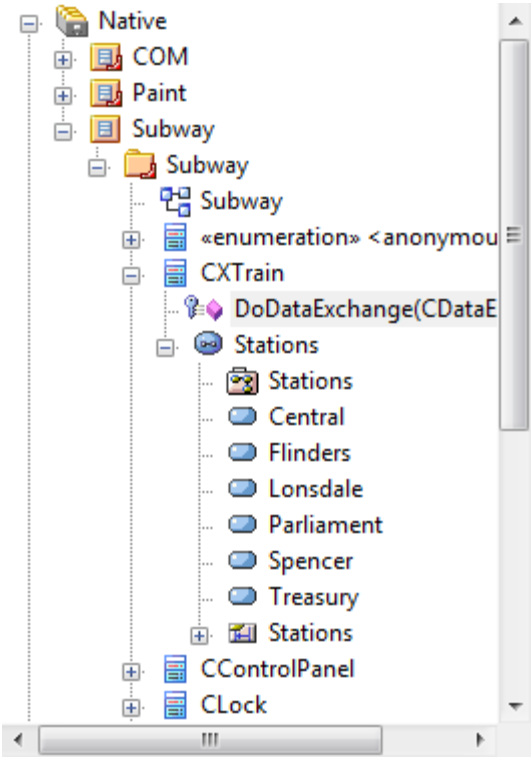


Showing State transitions on your debug-generated Sequence diagrams is optional; you set an option in the package script associated with the Class for which you intend to record States.

Note:

If you do not have a package script for the Class or package you must create one. Sequence diagrams can only be generated for a package that has been configured for debug.

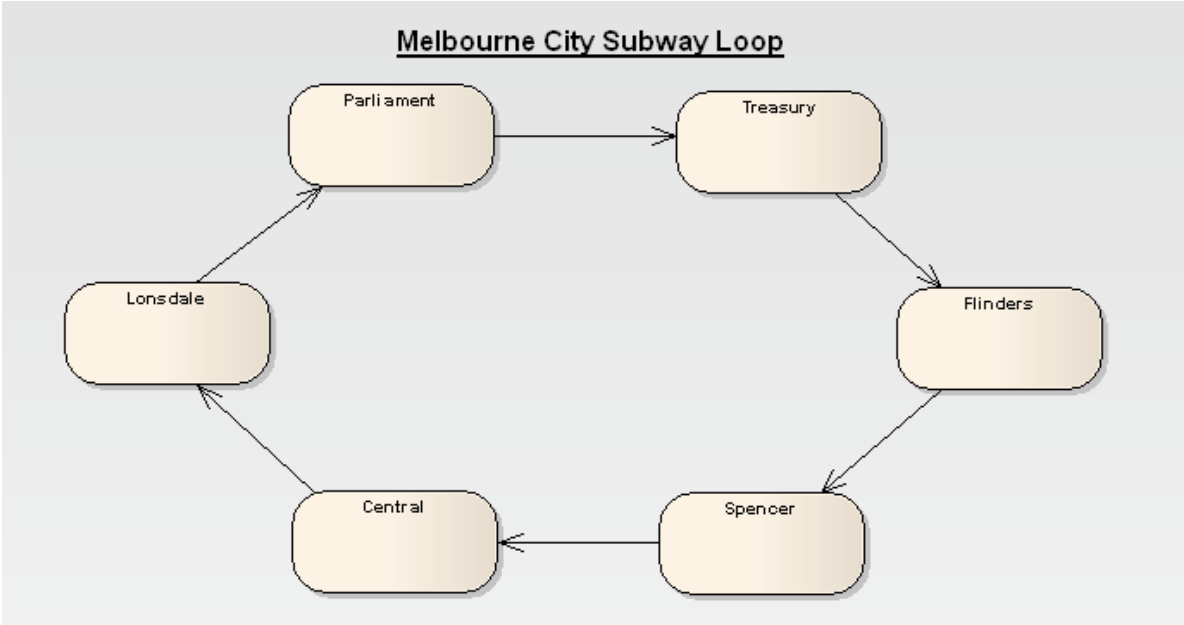
Next, you create a *State Machine* under the Class. On the State Machine you create the *State* elements that correspond to any states to be captured for your Class. The debugger evaluates your States by checking *constraints* on the States you create. The States on this diagram are then used by the debugger and State transitions are incorporated into the diagram.



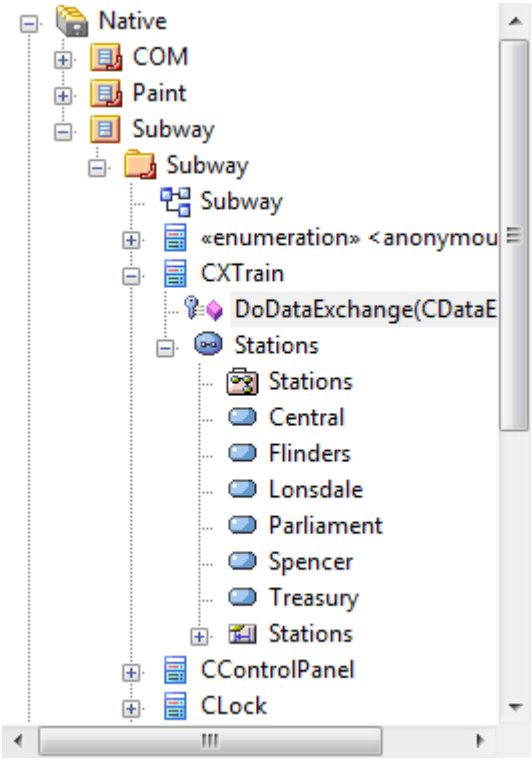
This figure shows the Class *CXTrain* with a State Machine called *Stations*. It has a child diagram also called *Stations,* on which the States {*Central,Flinders,Lonsdale...*} are placed.

4.1.6.2 The State Machine

A State Transition diagram can be used to illustrate how States change during the execution of an application. The Visual Execution Analyzer can build a State Machine to model all the valid system states and explicitly describe the transitions between each state. The diagram below is a State Machine that shows the different States within the Melbourne Underground Loop subway system. A train traveling on the subway network can be stopped at any of the stations represented on the State Machine below.

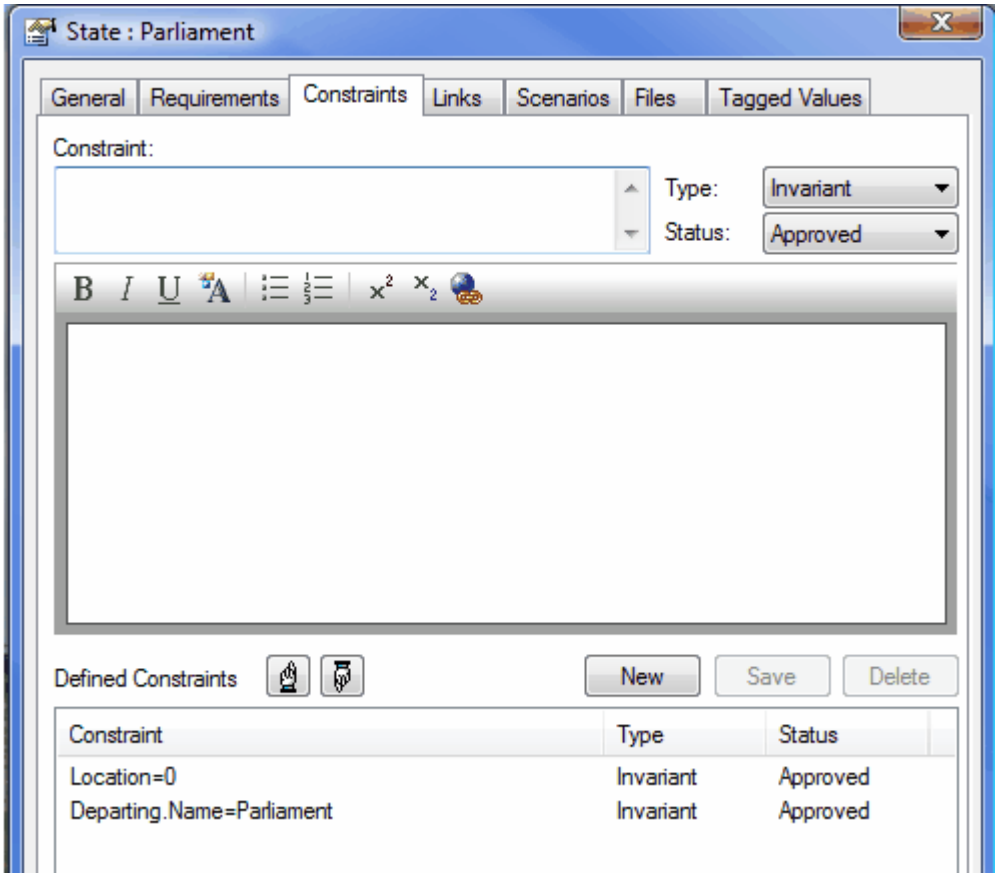


This State Machine diagram is a child of the *CXTrain* Class.



4.1.6.3 Recording and Mapping State Changes

The State Properties dialog below is for the State *Parliament*. The Constraints tab is open to show how the State is linked to the Class *CXTrain*. A State can be defined by a single constraint or by many; in the example below the State *Parliament* has two constraints.



The *CXTrain* Class has a member called *Location* of type *int,* and a member called *Departing.Name* of type *CString*.

The values of constraints can only be compared for *elemental, enum* and *string* types. What this constraint means is:

- when an instance of the *CXTrain* Class exists and
- its member variable *Location* has the value **0** and
- the member variable *Departing.Name* has the value **Parliament** then
- this State is evaluated to **true**.

Operators in Constraints

There are two types of operators you can use on constraints to define a State:

- Logical operators AND and OR can be used to combine constraints
- Equivalence operators {= and !=} can be used to define the conditions of a constraint.

All the constraints for a State are subject to an AND operation unless otherwise specified. You can use the OR operation on them instead, so you could rewrite the constraints in the above example as:

```
Location=0 OR

Location=1 AND

Departing.Name!=Central
```

Below are some examples of using the equivalence operators:

```
Departing.Name!=Central AND

Location!=1
```

Note:

Quotes around strings are optional. The comparison for strings is always case-sensitive in determining the truth of a constraint.

4.2 Unit Testing

Enterprise Architect supports integration with unit testing tools in order to make it easier to develop good quality software.

Firstly, Enterprise Architect helps you to create test Classes with the JUnit and NUnit transformations (see the *MDA Transformations User Guide*). Then you can set up 80 a test script 53 against any package and run 81 it. Finally, all tests results are automatically recorded 82 inside Enterprise Architect.

4.2.1 Set Up Unit Testing

In order to use unit testing in Enterprise Architect, you must first set it up. This happens in two parts.

Firstly the appropriate tests must be defined 53. Enterprise Architect is able to help with this. By using the JUnit or NUnit transformations and code generation (see *Code Engineering Using UML Models*) you can create test method stubs for all of the public methods in each of your Classes.

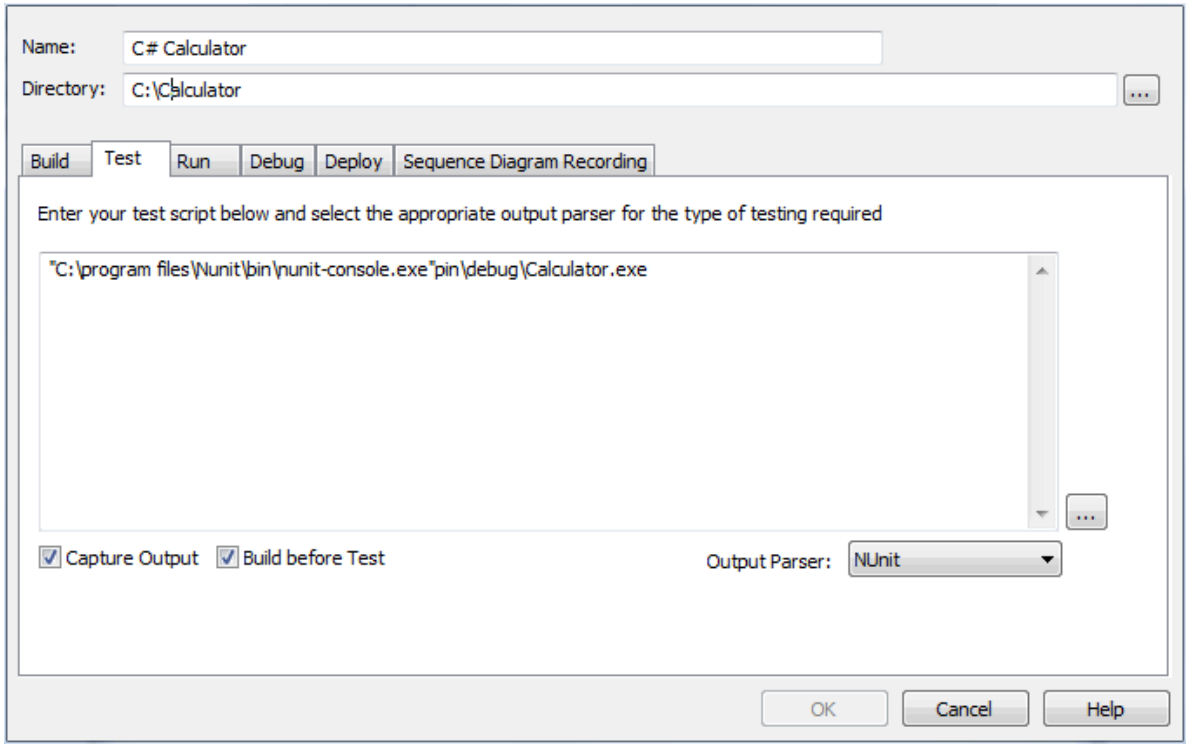
The following is an *NUnit* example in *C#* that is followed through the rest of this topic, although it could also be any other .Net language or Java and JUnit.

```
[TestFixture]
public class CalculatorTest
{

    [Test]
    public void testAdd(){
            Assert.AreEqual(1+1,2);
    }

    [Test]
    public void testDivide(){
            Assert.AreEqual(2/2,1);
    }

    [Test]
    public void testMultiply(){
            Assert.AreEqual(1*1,1);
    }

    [Test]
    public void testSubtract(){
            Assert.AreEqual(1-1,1);
    }
}
```

This code can be reverse engineered into Enterprise Architect so that Enterprise Architect can record all test results against this Class.

Once the unit tests are set up, you can then set up the Build and Test scripts to run the tests. These scripts must be set up against a package.

The sample above can be called by setting up the Package Build Scripts 12 dialog as follows.

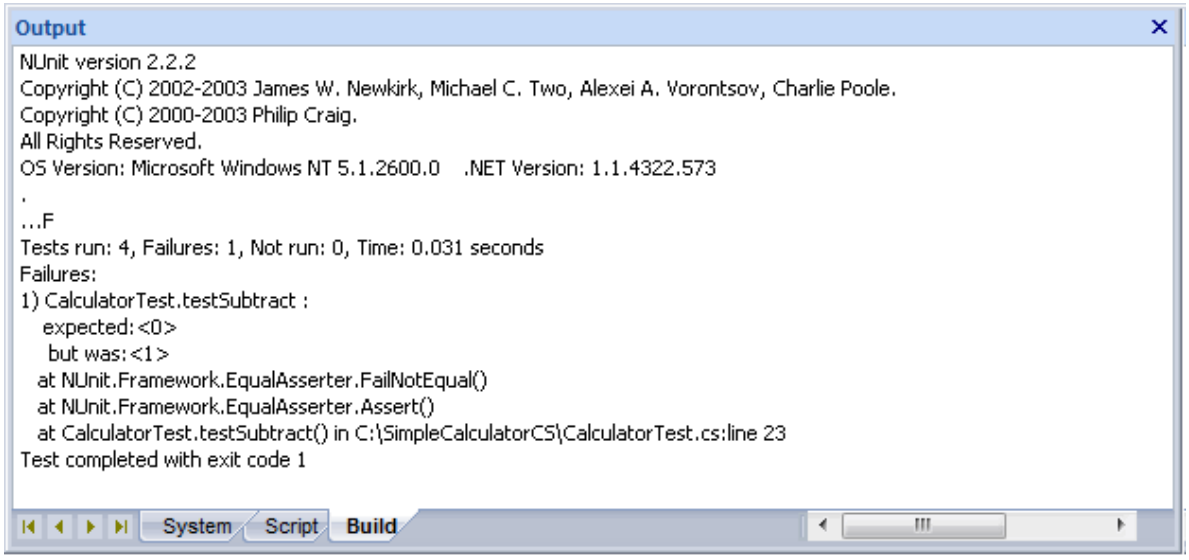


If Enterprise Architect is to handle unit testing, it is important that you select the **Capture Output** checkbox and select the appropriate **Output Parser** for the testing. Without doing this you won't see the program output and therefore you cannot open the source at the appropriate location.

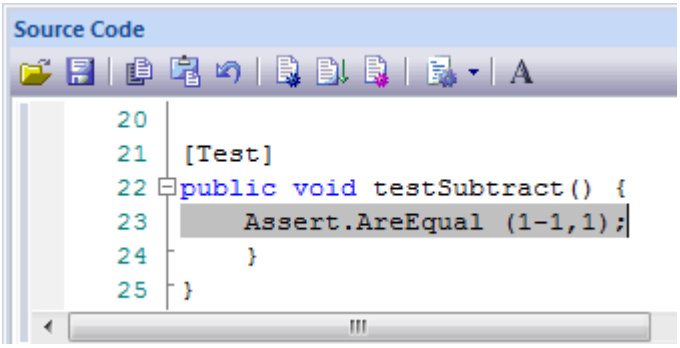
4.2.2 Run Unit Tests

You can run the test script you set up previously, by selecting the **Project | Execution Analyzer | Test** menu option.

The following output is generated.



Notice how NUnit reports that four tests have run, including one failure. It also reports what method failed and the file and line number the failure occurred at. If you double-click on that error, Enterprise Architect opens the editor to that line of code.



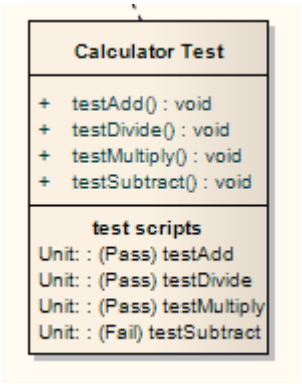
This enables you to quickly find and fix the error.

Enterprise Architect also records the run status of each test as described in Record Test Results 82.

4.2.3 Record Test Results

Enterprise Architect is able to automatically record all results from tests by a testing script 80 in Enterprise Architect. In order to use this feature, you just reverse engineer the test Class (see *Code Engineering Using UML Models*) into the package containing your test script.

Once your model contains your test Class, on the next run of the test script 81 Enterprise Architect adds test cases to the Class for each test method found. On this and all subsequent test runs all test cases are updated with the current run time and if they passed or failed as shown in the following illustration.



The error description for each failed test is added to any existing results for that test case, along with the current date and time. Over time this provides a log of all test runs where each test case has failed. This can then be included in generated documentation and could resemble the following.

Failed at 05-Jul-2006 1:02:08 PM
expected: <0>
but was: <1>

Failed at 28-Jun-2006 8:45:36 AM
expected: <0>
but was: <2>

4.3 Profiling Native Applications

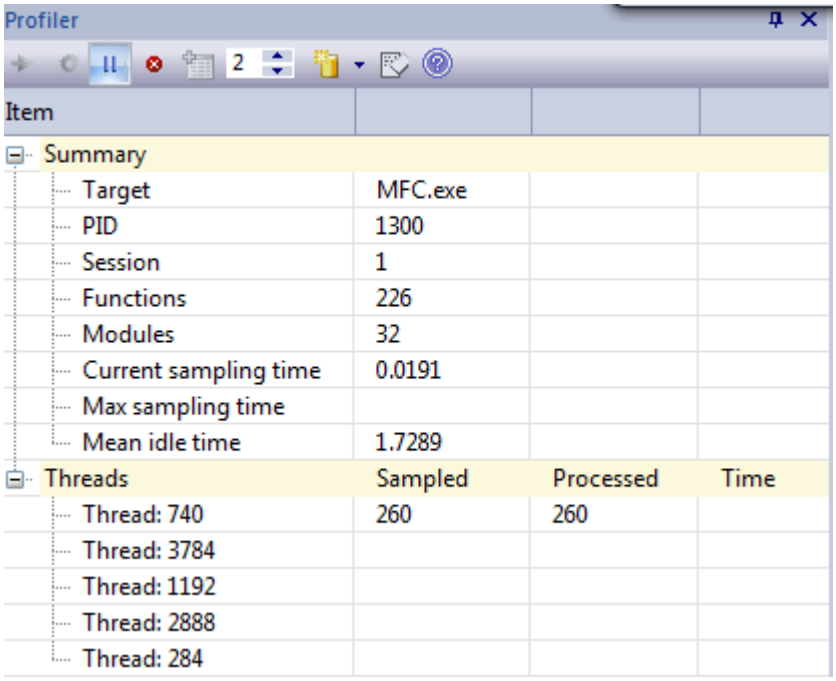
The Visual Execution Profiler enables you to quickly report on:

- The most frequently called functions in a running application
- Tasks in an application that are taking more time than expected
- Which functions are taking the most time in an application.

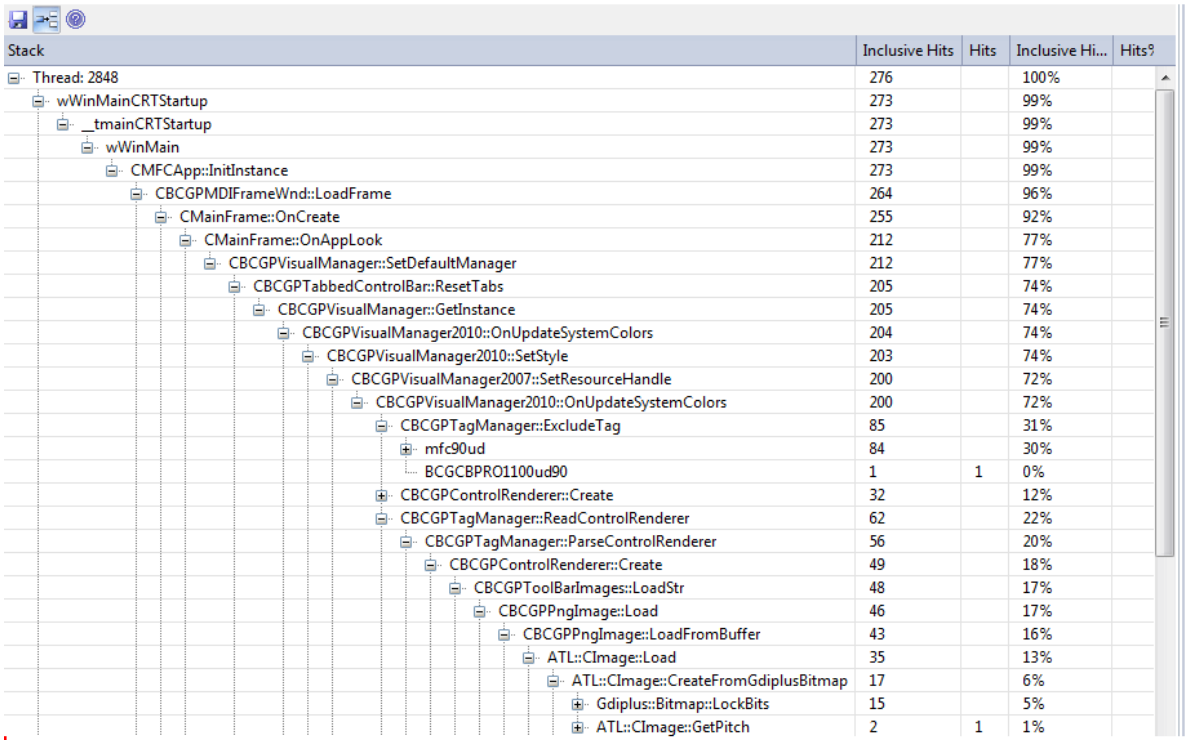
The Profiler, or sampler, is available in the Enterprise Architect Professional, Corporate, Business and Software Engineering, System Engineering and Ultimate editions.

Note:

The Profiler only works with MS Native Windows applications, but can be used under WINE (Linux and Mac) to debug standard Windows applications deployed in a WINE environment.



The Profiler can generate a report that shows how these functions are called in relation to the application, as illustrated below:



See Also

- Profiler System Requirements 84
- Profiler Operation 85

4.3.1 System Requirements

Prerequisites

The Profiler window 85 becomes available when a model is opened. Options 85 on the Profiler window toolbar enable you to attach to an existing process or launch a new application if a Package Script been specified.

Supported Platforms

Enterprise Architect supports profiling on native Windows applications (C, C++ and Visual Basic) compiled with the Microsoft™ native compiler where an associated PDB file is available. Select **Microsoft Native** from the list of debugging platforms in your package script.

The Profiler can sample both Debug and Release configurations of an application, providing the PDB for each executable exists and is up to date.

4.3.2 Getting Started

The Profiler window can be accessed by selecting the **View | Execution Analyzer | Profiler** menu option, or by selecting it from the *Analysis Windows* folder on the Debugger window (**[Alt]+[8]**). The toolbar options are explained in the table below.

The Profiler operates by taking samples of a process at intervals of up to 250 milliseconds. At these intervals the Profiler interrupts the process and collects stack information for all threads running at that time. This information is sent back to Enterprise Architect where it is collected sorted and and stored.

You can Pause and Resume profiling at any time during the session. You can also clear any sample data collected and begin again.

If you stop the Profiler and the process is still running, you can quickly attach to it again.

Icon	Use to
	(When an application is configured for the package) 82 create the Profiler process, which launches the configured application.
	Profile an application that is already running.
	When the application is running, pause and resume sample capture. Pausing sampling enables the **Report** and **Erase** buttons.
	Stop the Profiler process. If any samples have been collected, the **Report** button is enabled.
	Generate a report 82 on the current number of samples collected.
	Set the interval, in milliseconds, at which samples are taken of the target process. The range of possible values is **1** - **250**.
	Set Profiler options, using a drop-down menu. The options are: • **Load Report from Disk** - load and display a previously-generated report from an xml disk file • **Package Build Scripts ([Shift]+[F12])** - display the Build Script 12 dialog to enable creation or

Icon	Use to
	editing of package scripts and debug configuration • **Start Sampling Immediately** - begin sample collection immediately upon either process start (main thread entry point executed) or attachment of process by Profiler • **Capture Debug output** - capture any appropriate debug output and redirect it to the Enterprise Architect Output window • **Stop Process on Exit** - select to terminate the target process when the Profiler is stopped.
	Erase the collected data.
	Display the Help topic for this window.

4.3.3 Start & Stop the Profiler

For most debugging operations it is necessary to have first configured a Package Script that typically defines the application to build, test and debug as well as sequence recording options.

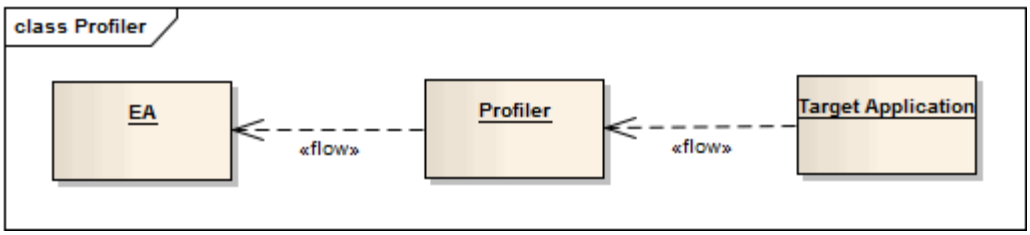
It is possible to use the Profiler without doing any of this by using the **Attach to Process** button.

If the Application to Profile is the one defined in the current Package, use the **Launch** button.

	(When an application is configured for the package) 82 create the Profiler process, which launches the configured application.
	Profile an application that is already running.
	Stop the Profiler process.

4.3.4 Profiler Operation

Enterprise Architect creates a Profiler process whenever you click on the **Launch** or **Attach to Process** button on the Profiler window toolbar. This process operates by collecting samples from the stacks of every thread in the target process.



The sampler process exits if you click on the **Stop** button, if the target application terminates, or if you close the current model.

You can turn sample collection on and off at any time during a session. When sampling is turned on or resumed, the Profiler process becomes active and samples are collected from the target. Resuming sampling collects completely new samples.

The Profiler process idles if sampling is turned off or paused during a session. The **Report** and **Erase** buttons then become enabled.

Click on the **Report** button to produce a call graph summary similar to that in the *Visual Execution Profiler* 82 topic. This report can be saved to file.

Click on the **Erase** button to discard any samples currently collected for the target.

4.3.5 Setting Options

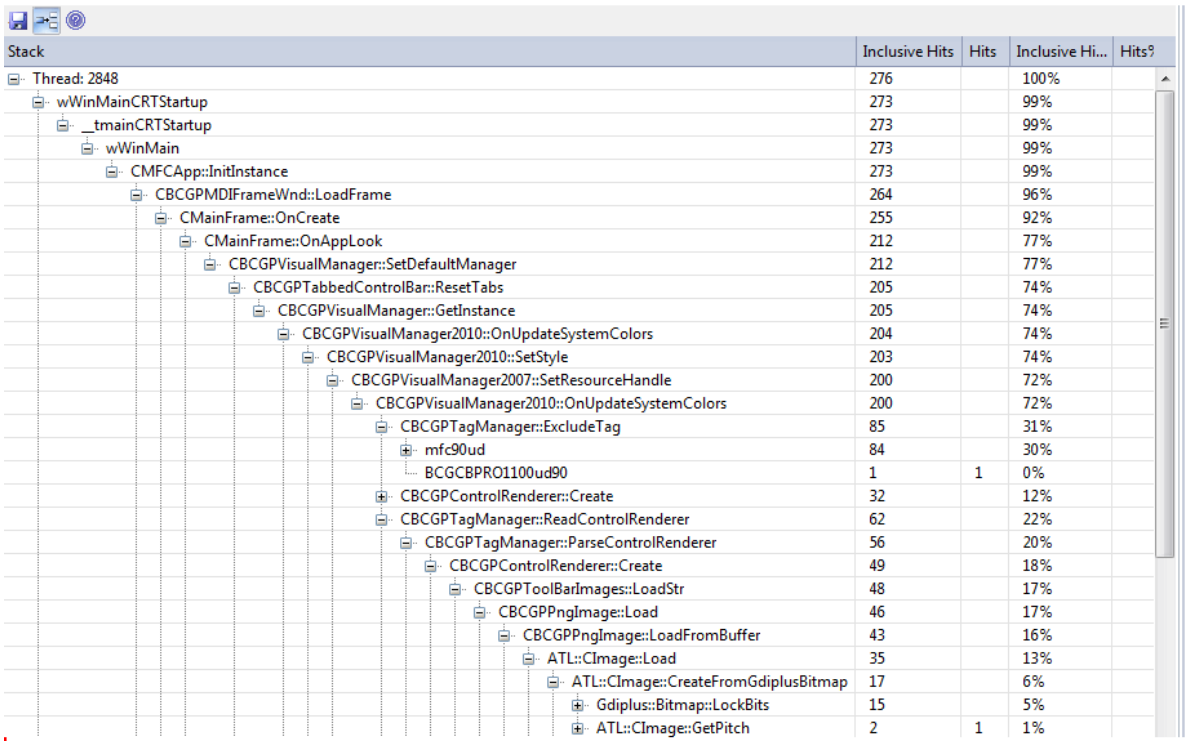
Interval

	Set the interval, in milliseconds, at which samples are taken of the target process. The range of possible values is **1** - **250**.

	Set Profiler options, using a drop-down menu. The options are: • **Start Sampling Immediately** - begin sample collection immediately upon either process start (main thread entry point executed) or attachment of process by Profiler • **Capture Debug output** - capture any appropriate debug output and redirect it to the Enterprise Architect Output window • **Stop Process on Exit** - select to terminate the target process when the Profiler is stopped.

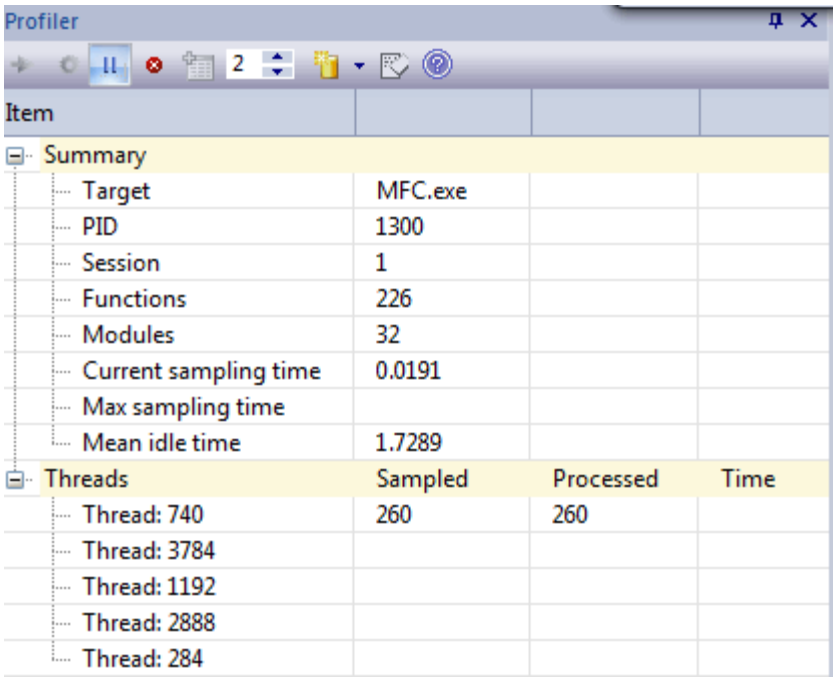
4.3.6 Save and Load Reports

The Profiler Reports can be *saved* in either binary format or xml format. Save the report using the toolbar above the report (Stack) view.



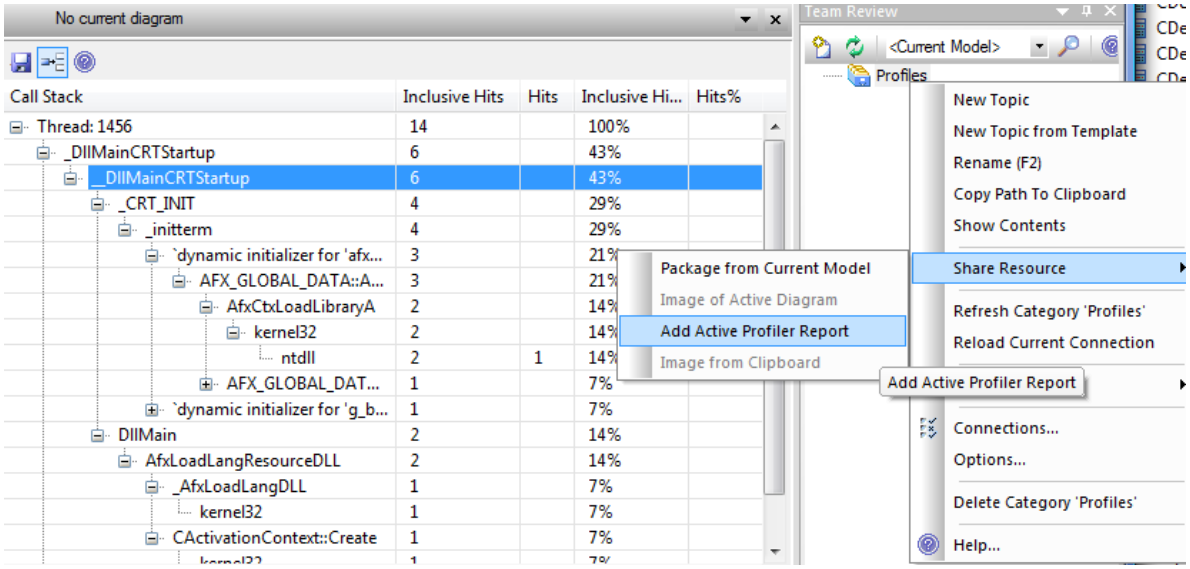
Stack	Inclusive Hits	Hits	Inclusive Hi...	Hits?
Thread: 2848	276		100%	
wWinMainCRTStartup	273		99%	
__tmainCRTStartup	273		99%	
wWinMain	273		99%	
CMFCApp::InitInstance	273		99%	
CBCGPMDIFrameWnd::LoadFrame	264		96%	
CMainFrame::OnCreate	255		92%	
CMainFrame::OnAppLook	212		77%	
CBCGPVisualManager::SetDefaultManager	212		77%	
CBCGPTabbedControlBar::ResetTabs	205		74%	
CBCGPVisualManager::GetInstance	205		74%	
CBCGPVisualManager2010::OnUpdateSystemColors	204		74%	
CBCGPVisualManager2010::SetStyle	203		74%	
CBCGPVisualManager2007::SetResourceHandle	200		72%	
CBCGPVisualManager2010::OnUpdateSystemColors	200		72%	
CBCGPTagManager::ExcludeTag	85		31%	
mfc90ud	84		30%	
BCGCBPRO1100ud90	1	1	0%	
CBCGPControlRenderer::Create	32		12%	
CBCGPTagManager::ReadControlRenderer	62		22%	
CBCGPTagManager::ParseControlRenderer	56		20%	
CBCGPControlRenderer::Create	49		18%	
CBCGPToolBarImages::LoadStr	48		17%	
CBCGPPngImage::Load	46		17%	
CBCGPPngImage::LoadFromBuffer	43		16%	
ATL::CImage::Load	35		13%	
ATL::CImage::CreateFromGdiplusBitmap	17		6%	
Gdiplus::Bitmap::LockBits	15		5%	
ATL::CImage::GetPitch	2	1	1%	

To *load* a report use the Profiler Toolbar **Options** button  and select the **Load Report From Disk** option.



4.3.7 Save Report in Team Review

You can save any current report as a resource for a Category, Topic or Post in the Team Review. The report can then be shared and reviewed at any time as it is saved with the model.



4.4 Object Workbench

This section describes the Object Workbench:

- How it works 88
- Workbench variables 88
- Create Workbench Variables 89
- Invoke Methods. 90

4.4.1 How it Works

The Workbench is a tool in Enterprise Architect Debugging, enabling you to create your own variables 89 and invoke methods 90 on them. Stack trace can be recorded and Sequence diagrams produced from the invocation of such methods. It provides a quick and simple way to debug your code.

Platforms Supported

The Workbench supports the following workbench platforms:

- Microsoft .NET (version 2.0 or later)
- Java (JDK 1.4 or later)

Note:

The Workbench does not currently support the creation of Class instances written in native C++, C or VB.

Mode

The Workbench operates in two modes:

Idle mode

When the Workbench is in idle mode, instances can be created and viewed and their members inspected.

Active mode

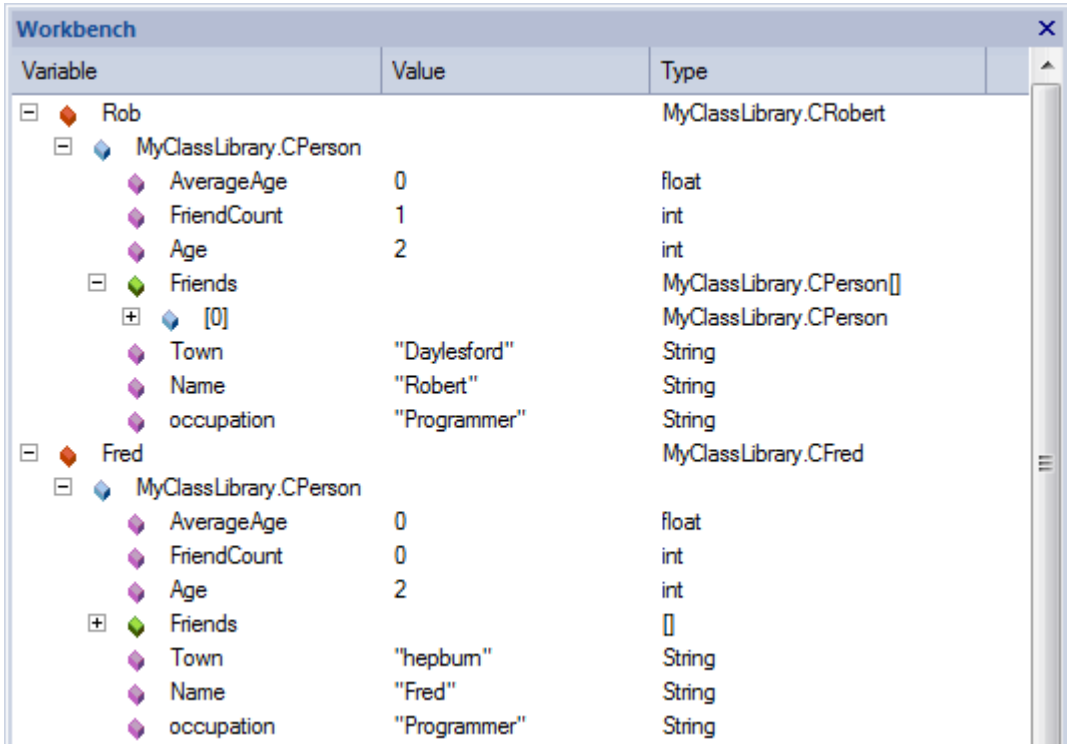
When methods are invoked on an instance, the Workbench enters *Active* mode, and the variables displayed change if the debugger encounters any breakpoints. If no breakpoints are set, then the variables do not change. The Workbench immediately returns to *Idle* mode.

Logging

The results of creating variables and the results of calls on their methods are displayed in the Debug Output window.

4.4.2 Workbench Variables

You can create (and delete) workbench variables from any Class in your model. When you do so, you are asked to name the variable. It then displays in the Workbench window. It shows the variable in a hierarchy, displaying its type and value and those of any members.



Workbench Requirements

- NET framework version 2 is required to workbench any .NET model.
- The package from which the variable is created must have a debugger configured (see the *Debug Tab* 15 topic).

Constraints (.NET)

- Members defined as *struct* in managed code are not supported.
- Classes defined as *internal* are not supported.

Delete Workbench Variables

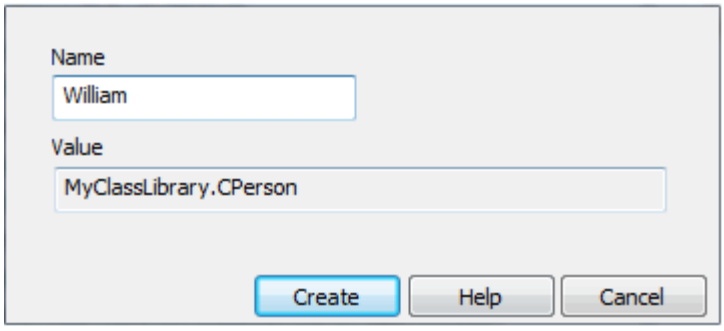
You can delete variables using the **Delete** shortcut menu on any instance on the Workbench. If all instances are deleted the debugger is shut down, and the Workbench window is closed.

4.4.3 Create Workbench Variables

Right-click on the required Class node in the Project Browser and select the **Create Workbench Instance** context menu option, or press **[Ctrl]+[Shift]+[J]**. The menu option is also available from within a Class diagram.

Naming the Workbench

When you elect to create an instance of a type Enterprise Architect prompts you with the Workbench dialog to name the variable. Each instance name must be unique for the workbench.

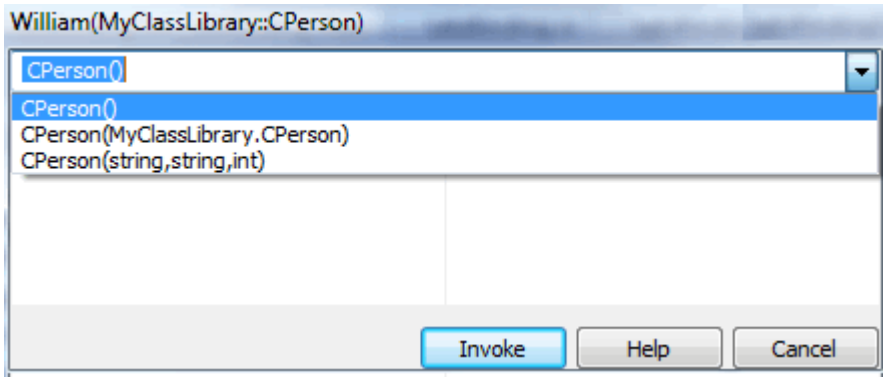


Choosing a Constructor

Having given the variable a name, you must now choose which constructor to use.

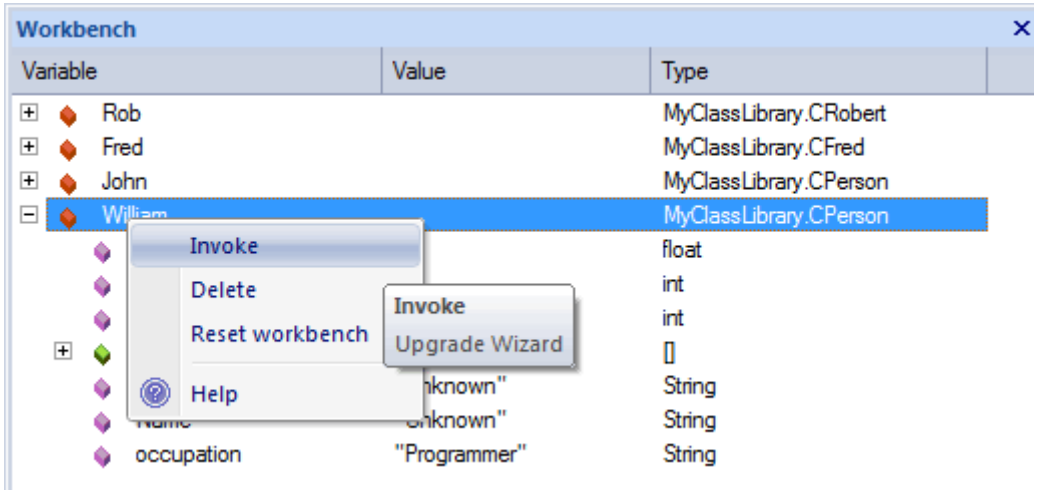
If you do not define a constructor, or define a single constructor taking no arguments, the default constructor or the defined constructor is automatically invoked.

Otherwise the following dialog displays. Select the constructor from the drop-down list and fill in any parameters required.



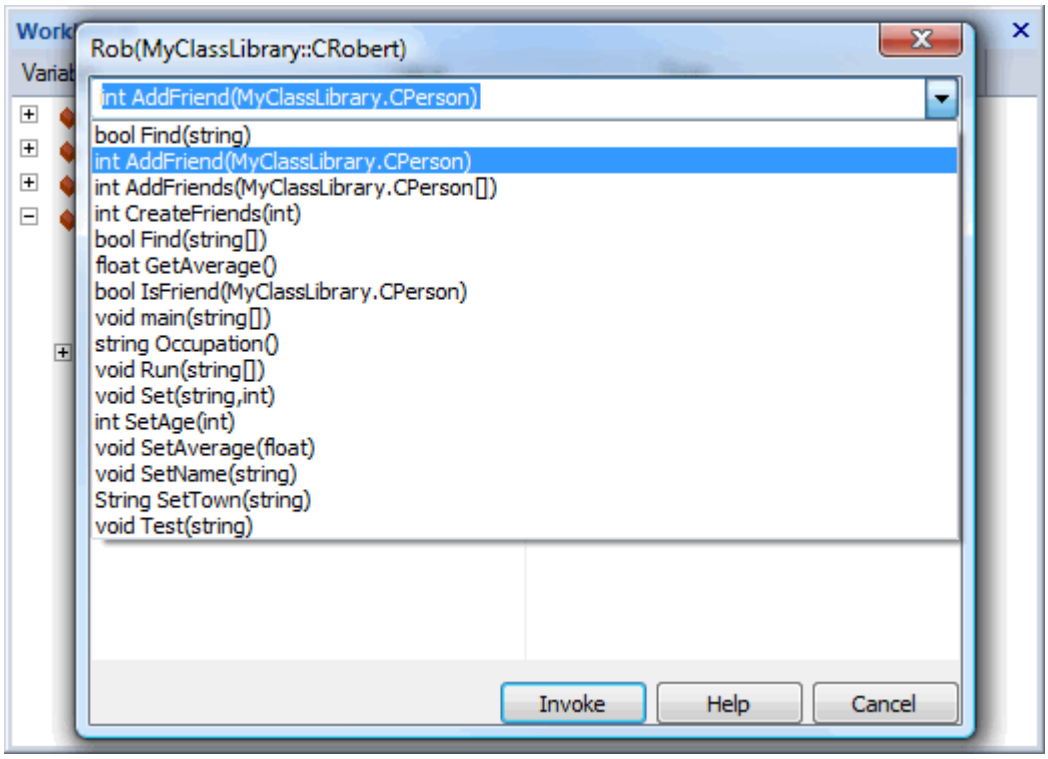
4.4.4 Invoke Methods

On the Workbench window, right-click on the instance on which to execute a method, and select the **Invoke** context menu option.



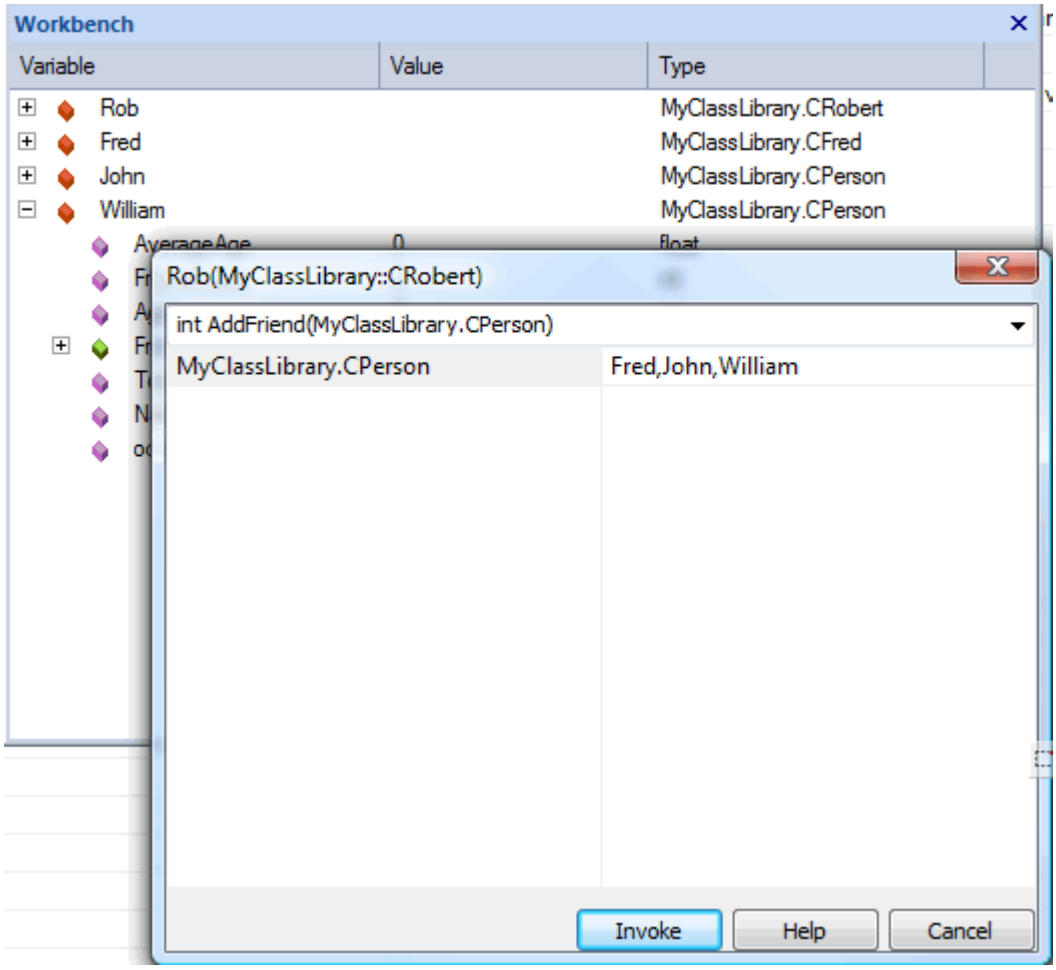
Choose Method

A list of methods for the type are presented in a dialog. Select a method from the list and click on the **Invoke** button. Note that all methods listed are public; private methods are not available.



Supply Arguments

In this example, you have created an instance or variable named *Rob* of type *MyClassLibrary.CRobert,* and have invoked a method named *AddFriends* that takes an array of *CPerson* objects as its only argument. What you now supply to it are the three other Workbench instances *Fred, John* and *William*.



Arguments

In the dialog above, type any parameters required by the constructor.

- **Literals as arguments**
  - Text: abc or "abc" or "a b c"
  - Numbers: 1 or 1.5
- **Objects as arguments**

  If an argument is not a literal then you can supply it in the list only if you have already created an instance of that type in the workbench. You do this by typing the name of the instance as the argument. The debugger checks any name entered in an argument against its list of workbench instances, and substitutes that instance in the actual call to the method.
- **Strings as arguments**

  Surrounding strings with quotes is unnecessary as anything you type for a string argument becomes the value of the string; for example, the only time you should surround strings with quotes is in supplying elements of a string array, or where the string is equal to the name of an existing workbench instance.

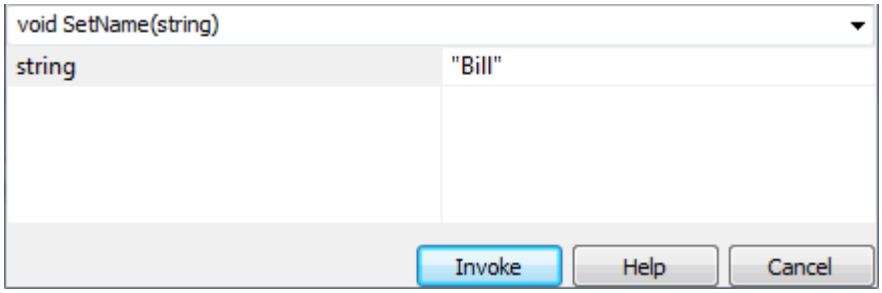
  "A b c"
  "a b $ % 6 4"
  A b c d
  As 5 7) 2 === 4
- **Arrays as arguments**

  Enter the elements that compose the array, separated by commas.

Type	Arguments
String[]	one,two,three,"a book","a bigger book"
CPerson[]	Tom,Dick,Harry

Note:

If you enter text that matches the name of an existing instance, surround it in quotes to avoid the debugger passing the instance rather than a string.



Invoke

Having chosen the constructor and supplied any arguments, click on the **Invoke** button to create the variable. Output confirming this action is displayed in the Output tab 47.

Index

- . -

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- I -

- J -

- L -

- M -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

Visual Execution Analyzer in Enterprise Architect

www.sparxsystems.com