Enterprise Architect

User Guide Series

Build and Debug

Sparx Systems Enterprise Architect provides a complete toolset to build, debug, visualize, record, test and profile software applications, integrated with modeling, design and code generation; Analyzer Scripts define how applications will be analyzed.

Author: Sparx Systems
Date: 2021-09-02
Version: 15.2

CREATED WITH ENTERPRISE ARCHITECT

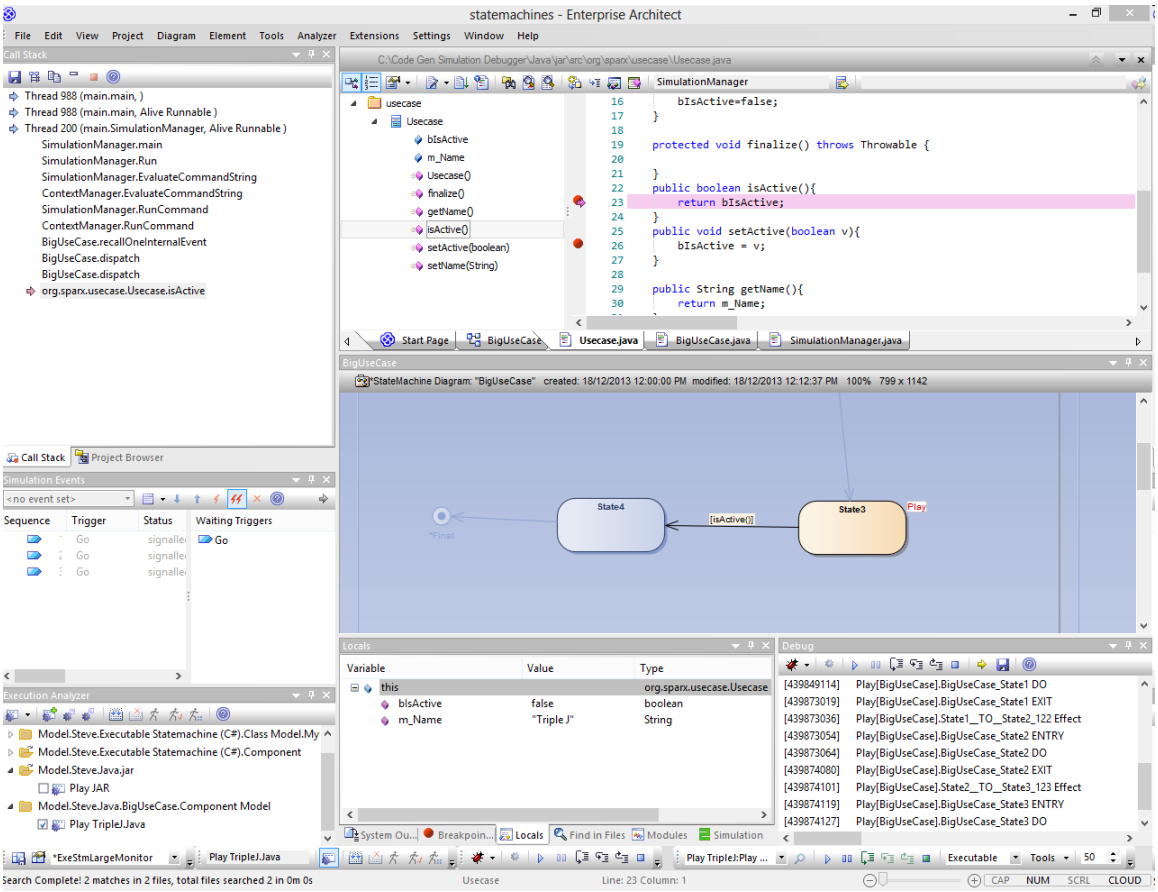
Table of Contents

Build and Debug 6
Analyzer Scripts 11
Managing Analyzer Scripts 14
Analyzer Script Editor 22
Build Scripts 29
Cleanup Script 34
Test Scripts 38
Testpoints Output 42
Debug Script 45
Operating System Specific Requirements 47
UAC-Enabled Operating Systems 50
WINE Debugging 52
Java 55
General Setup for Java 56
Advanced Techniques 60
Attach to Virtual Machine 61
Internet Browser Java Applets 63
Working with Java Web Servers 65
JBOSS Server 69
Apache Tomcat Server 71
Apache Tomcat Windows Service 73
.NET 74
General Setup for .NET 75

Debugging an Unmanaged Application 78
Debug COM Interop 79
Debug ASP .NET 81
The Mono Debugger 83
Debugging Configuration Linux 86
Debugging Configuration Windows 90
The PHP Debugger 93
PHP Debugger - System Requirements 99
PHP Debugger Checklist 100
The GNU Debugger (GDB) 104
The Android Debugger 107
Java JDWP Debugger 113
Tracepoint Output 119
Workbench Setup 122
Microsoft C++ and Native (C, VB) 124
General Setup 125
Debug Symbols 128
Merge Script 129
Code Miner Script 130
Services Script 134
Run Script 135
Deploy Script 138
Recording Scripts 141
Build Application 146
Locate Compiler Errors in Code 148
Debugging 150
Run the Debugger 153

Breakpoint and Marker Management 160
Setting Code Breakpoints 165
Trace Statements 167
Break When a Variable Changes Value 171
Trace When Variable Changes Value 176
Detecting Memory Address Operations 179
Breakpoint Properties 182
Failure to Bind Breakpoint 185
Debug a Running Application 186
View the Local Variables 189
View Content Of Long Strings 195
View Debug Variables in Code Editors 197
Variable Snapshots 198
Actionpoints 201
View Variables in Other Scopes 209
View Elements of Array 212
View the Call Stack 214
Create Sequence Diagram of Call Stack 218
Inspect Process Memory 220
Show Loaded Modules 222
Process First Chance Exceptions 224
Just-in-time Debugger 227
Services 228

Build and Debug



Enterprise Architect builds on top of its already exceptional code generation, diagramming and design capabilities with a complete suite of tools to build, debug, visualize, record, test, profile and otherwise construct and verify software applications. The toolset is intimately connected to the modeling and design capabilities and provides a unique and powerful means of constructing software from a model and keeping model and code synchronized.

Enterprise Architect helps you define 'Analyzer Scripts' linked to Model Packages that describe how an application will be compiled, which debugger to use and other related information such as simulation commands. The Analyzer

Script is the core configuration item that links your code to the build, debug, test, profiling and deployment capabilities within Enterprise Architect.

As a measure of how competent the toolset is, it should be noted that Enterprise Architect is in fact built, debugged, profiled, tested and otherwise constructed fully within the Enterprise Architect development environment. Many of the advanced debugging tools such as 'Action Points' have been developed to solve problems inherent in the construction of large and complex software applications (such as Enterprise Architect) and are routinely used on a daily basis by the Sparx Systems development team.

It is recommended that new users take the time to fully understand the use of the Analyzer Scripts and how they tie the model to the code, to the compilers and to other tools necessary for building software.

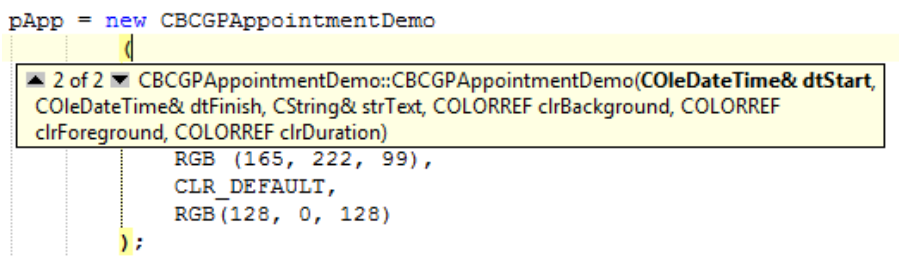
Integrating Model and Code

Model Driven Engineering is a modern approach to software development and promises greater productivity and higher quality code, resulting in systems getting to market faster and with fewer faults. What makes this approach compelling is the ability for the architecture and the design of a system to be described and maintained in a model, and then generated to programming code and schemas that can be synchronized with and visualized within the model.

Enterprise Architect's Model Driven Development

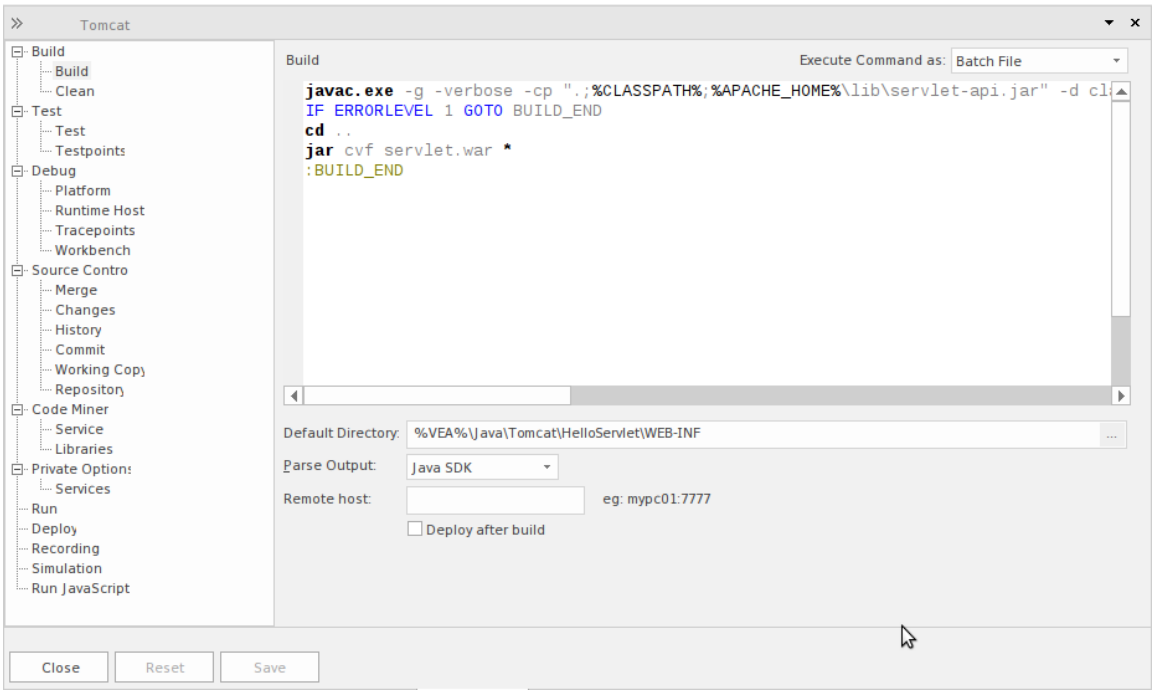
Environment (**MDDE**) supports this approach and provides a set of flexible tools to increase productivity and reduce errors. These include the ability to define the architecture and design in models, generate code from these models, synchronize the code with the models and maintain the code in sophisticated code editors. Source code or binaries can also be imported, and users can record and document pre-existing or recently developed code. The Analyzer Script tool helps you to describe how to build, debug, test and deploy an application.

Facility	Description
Model Driven Development	Model Driven Development provides a more robust, accessible and faster development cycle than traditional coding-driven cycles. A well constructed model, intimately linked with source code build, run, debug, test and deploy capabilities provides a rich, easily navigated and easily understood target architecture. Traceability, linkage to Use Cases, Components and other model artifacts, plus the ability to readily record and document pre-existing or recently developed code, make Enterprise Architect's development environment uniquely powerful. Enterprise Architect incorporates industry

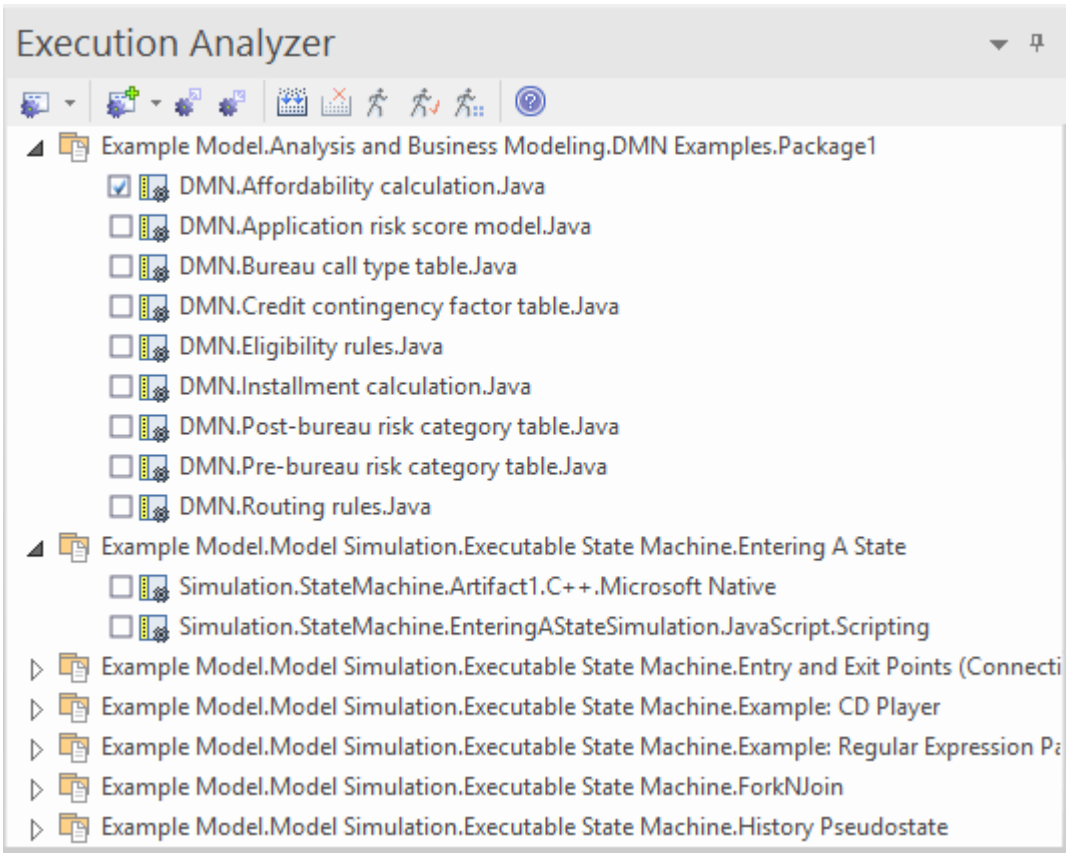
	standard intelligent editing, debuggers and modeling languages.
The Model Driven Development Environment (MDDE)	The **MDDE** provides tools to design, visualize, build and debug an application: • UML technologies and tools to model software • Code generation tools to generate/reverse engineer source code • Tools to import source code and binaries • Code editors that support different programming languages • Intelli-sense to aid coding • Analyzer scripts that enable a user to describe how to build, debug, test and deploy the application • Integration with compilers such as Java, .Net, Microsoft C++ • Debugging capabilities for Java, .NET, Microsoft C++ and others • Advanced visualization, recording, inspection, testing and profiling capabilities 

Analyzer Scripts

Analyzer Scripts are used by the Execution Analyzer. You do not need to worry about creating these. They are not the same type of script as JavaScript or PHP, but are managed using a familiar user interface - a tree view - and you can quickly locate the feature to change. Analyzer Scripts can be shared by users of a community model and are easily imported and exported as XML files.



A single project can have multiple configurations and these can be found grouped together in the **Analyzer window**.



Each Analyzer Script is defined for a Package, so projects can co-exist quite happily. In many organizations, the procedures to manage systems are distributed, and vary from individual to individual and group to group. Analyzer Scripts in an Enterprise Architect model can provide some peace of mind to these organizations, by trusting a single, shared and accountable procedure for building and deploying any variety of configurations. All aspects of a script are optional. You can, for instance, debug without one; however, with a few lines they can enable these powerful features:

- Building
- Testing
- Debugging
- Recording

- Execution
- Deployment
- Simulation

Remote Script Execution

Various Analyzer Script sections such as Build and Run, provide a 'Remote Host' field. This field is used to describe the computer on which the script should run. In order to use this feature, the Sparx Satellite service must be running on the machine. The format of this field is *hostname:port,* where *hostname* is the IP address or network name of a Windows or Linux machine and *port* is the Port number that the Satellite service is listening on. The primary goal of this feature is to allow a user of Enterprise Architect running on Linux to execute commands native to Linux.

Managing Analyzer Scripts

The **Execution Analyzer window** enables you to manage all Analyzer scripts in the model. You can use the window toolbar buttons or script context menu options to control script tasks. Scripts are listed by Package; the list only shows Packages that have Analyzer scripts defined against them. Each user can set their own active script, independent of other users of the same model; one user activating a script does not impact the currently active scripts for other users or affect the scripts available to them. The active script governs the behavior of the Execution Analyzer; when choosing the build command from a menu, for example, or clicking the **Debug button** on a toolbar.

There can be a lot of Packages with scripts listed in the window. To help you locate and isolate a particular Package, use the '**Filter** Packages' context menu options, described in the *Context Menu Options* table in this topic.

Access

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard	**Shift+F12**

Shortcuts	

Toolbar Options

Toolbar Button	Action
	Quick access to the Analyzer core windows such as **Call Stack** or Local Variables, plus the power features: • **Profiling** • Recording • **Testpoints** • Simulation
	Create and edit a new Analyzer Script for the selected Package, under either Linux or Windows™.
	Export Scripts. Export one or more Analyzer Scripts to an XML file, which can be used to import the scripts into another model. The 'Execution Analyzer: Export' dialog displays from which you select the script

	or scripts to export, followed by a prompt for the target file name and location.
	Import Scripts. Import one or more Analyzer Scripts into the current model from a previously exported XML file. The 'Find Package' dialog displays, on which you select the Package into which to import the scripts, followed by a prompt for the source filename and location.
	Execute the 'Build' command of the active script.
	Cancel the 'Build' command currently in progress.
	Execute the 'Run' command of the active script.
	Execute the 'Test' command of the active script.
	Execute the 'Deploy' command of the active script.

Context Menu Options

Right-click on the required script or Package to display the context menus.

Option	Action
Add New Script	Add a new script to the selected Package. The **Execution Analyzer window** displays, showing the 'Build' page.
Paste Script	Paste a copied script from the Enterprise Architect clipboard into the selected Package. You can paste the copied script several times; each copy has the suffix 'Copy'. To rename the copied script, press **F2** and overtype the script name.
Export Scripts	Export scripts from the selected Package. The 'Execution Analyzer: Export' dialog displays, from which you select the script or scripts to export, followed by a prompt for the target filename and location.
Import Scripts	Import scripts from a .XML file into the selected Package.

	A prompt displays for the source filename and location.
Filter Packages	Displays a submenu of options to: • Enter the Package path against which to filter the list of Packages - when you select the '**Filter** Packages' option, a prompt displays to accept the currently-selected Package path, or you can remove the *end* elements of the path to specify a larger set of Packages; when you click on the **OK button** the first Package in the list is expanded to list the scripts it contains • Switch between showing the full list of Packages and hiding the full list of Packages to expose only the currently-selected Package • Remove the currently-active filter to show the full list of Packages
Select In Project Browser	Highlight the selected Package in the **Browser window**. Display the Browser window, which is now expanded to show the highlighted Package.
Build	Execute the 'Build' command of the selected script.

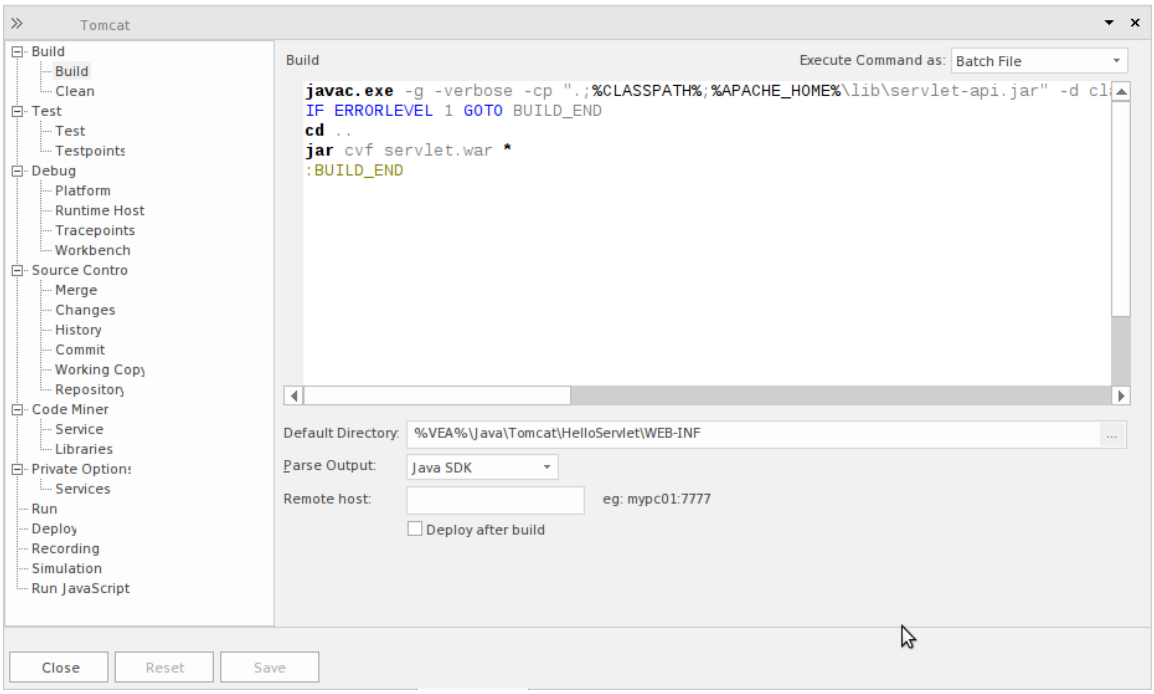
Clean	Execute the 'Clean' command of the selected script.
Rebuild	Execute the 'Clean' and 'Build' commands of the selected script.
Debug	Execute the 'Debug' command of the selected script.
Run	Execute the 'Run' command of the selected script.
Test	Execute the 'Test' command of the selected script.
Deploy	Execute the 'Deploy' command of the selected script.
Merge	Execute the 'Merge' source control command of the selected script.
Changes	Execute the 'Changes' source control command of the selected script.
History	Execute the 'History' source control command of the selected script.
Commit	Execute the 'Commit' source control

	command of the selected script.
Working Copy	Execute the 'Working Copy' source control command of the selected script.
Repository	Execute the 'Repository' source control command of the selected script.
Run JavaScript	Execute the 'Run JavaScript' command of the selected script.
Start Simulation	Start the simulation referenced by the 'Analyzer Script Simulation' page.
Run Executable Statemachine	Start a simulation of the selected Executable Statemachine Artifact.
Edit	Open the selected script in the 'Analyzer Scripts Editor'.
Copy	Copy the selected script to the Enterprise Architect clipboard.
Paste	Paste the most-recently copied script to the same Package as the selected script. You can paste the copied script several times; each copy has the suffix 'Copy'. To rename the copied script, press **F2** and

	overtype the script name.
Delete	Delete the selected script; there is no prompt for confirmation. To delete a Package from the **Execution Analyzer window**, delete the scripts from the Package. When the last script is deleted, the Package is no longer listed.
Package Default	Set the selected script as the default script for the Package. The icon to the left of the script changes color; any previous Package default reverts to normal.

Analyzer Script Editor

The Analyzer Script Editor has a straightforward user interface, with a tree view of the scripts on the left enabling easy navigation of the script groups, and a content view on the right in which you define and configure the scripts.



Access

From the 'Execution Analyzer' window, either:

- Double-click on a script to edit it or
- Right-click on a script and select the 'Edit' option

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer

Keyboard Shortcuts	**Shift+F12**

Execution Analyzer Scripts

Task - Page	Action
Build - Build	Enter the script or command to build the application. This can be an Apache Ant or Visual Studio command, but can also be tailored depending on your development environment. Note: Remember to select a parser to get directly to the source code in the event of any errors. The parser field is on the same page and offers support for many languages.
Build - Clean	Enter the script or command to clean the previous build. This is the command line you would normally issue to build your system. This can be an Apache Ant or Visual Studio command depending on your development environment.

Test - Test	Enter the script or command to test the application. This is typically where an nUnit or jUnit invocation might be configured, but it just as easily could be any procedure or program.
Test - Testpoints	Specify where the output from a **Testpoint** run is sent.
Debug - Platform	Specify the debugging platform, the application to be debugged, and the mode of debugging (attach to process or run).
Debug - Tracepoints	Specify where the output from Tracepoints encountered during a debug session are sent.
Debug - Workbench	For .NET projects, the assembly to load. Not required for Java.
Debug - Runtime Host	Allows Enterprise Architect to launch the program to be debugged using a command line. This is typically used for Mono or Java programs that use a socket transport for debugging. This command is executed prior to the **Debugger** being run. The Port number specified in this command should be the same value passed to the 'Port' option in the Debug

	page. When the Debugger starts, it will attempt to connect to the runtime on this Port. If successful, it then binds any breakpoints and resumes the program,which it assumes to be suspended. Java and Mono both have command line options on the debugging transport to initially suspend the process till the Debugger connects.
Source Control - Merge	This is the script that executes when the 'Merge' option is chosen from the context menu of an Analyzer Script. It provides a place to run a program or shell script to examine the differences between source files.
Source Control - Changes	This is the script that executes when the 'Changes' option is chosen from the context menu of an Analyzer Script. It provides a place to run a source control program such as 'svn' that might list current changes to a source control repository.
Source Control - History	This is the script that executes when the 'History' option is chosen from the context menu of an Analyzer Script. It provides a place to run a source control

	program such as 'svn' that might list a history of changes to a source control repository.
Source Control - Commit	This is the script that executes when the 'Commit' option is chosen from the context menu of an Analyzer Script. It provides a place to run a source control program such as 'svn' that might commit changes to a source control Working Copy.
Source Control - Working Copy	This is the script that executes when the 'Working Copy' option is chosen from the context menu of an Analyzer Script. It provides a place to run a source control program such as 'svn' to perform actions on the current working copy of a source repository.
Source Control - Repository	This is the script that executes when the 'Repository' option is chosen from the context menu of an Analyzer Script. It provides a place to run a source control program such as 'svn' to perform actions on a source repository.
Code Miner - Service	In this section you can choose how the Code Miner service operates. You can

	choose either a remote server or to use libraries locally.
Code Miner - Libraries	This section provides a place for the management of Code Miner libraries. Here, you can create libraries based on a project codebase or repository. Code Miner libraries created here can be searched using mFQL queries. Queries composed in mFQL can be used to search one or more libraries in a single operation.
Private Options - Services	This is where the IP address and Port of Enterprise Architect Satellite services for Linux and Windows are configured. These services provide enterprise-wide support for system management functions and remote debugging scenarios.
Run JavaScript	This is the script that executes when the 'Run JavaScript' option is chosen from the context menu of an Analyzer Script.
Run	Enter a command to run an application.
Deploy	Enter a script or command to deploy the project. Build your jar file. Deploy to your device, an emulator or Tomcat

	server. Publish a web site. Its up to you.
Recording	Does your Sequence diagram resemble the national grid? Reduce the clutter with filters. Filters define exclusion zones in your code base that can cut down dramatically on any 'noise' that is being recorded. Even accurate noise is not always helpful.
Simulation	Complete the configuration for **Simulation Control**.

Build Scripts

The 'Build' page enables you to enter commands to build your project. You can use Enterprise Architect Local Paths and environment variables in composing your command line(s). You can choose to create your own build script, entering various shell commands. You can also choose to simply run an external program or batch file such as an Ant script.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Build > Build' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Build > Build' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Analyzer Context Menu	Build

Keyboard Shortcuts	**Ctrl+Shift+F12**

Build

Write your script in this text box, using the windows shell commands; the format and content of this section depends on the actual compiler you use to build your project. If the path or arguments contain spaces, surround them with quotes; for example: "c:\program files (x86)\java\bin\javac.exe".

Here are some examples:

Visual Studio:

"C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\devenv.com" /Rebuild Debug RentalSystem.sln

Visual Studio using a Local Path:

"%VsCompPath%\devenv.exe" /build Debug Subway.sln

Java:

C:\Program Files (x86)\Java\jdk1.6.0_22\bin\javac.exe" -g -cp "%classpath%;." **%r***.java

Java using a Local Path:

"%JAVA%\bin\javac.exe" -g -cp "%classpath%;." %r*.java

Wildcard Java Builds (%r) - Source files in sub-folders can be built using the %r token. The token has the effect of causing a recursive execution of the same command on any files in all sub-folders, as shown in the example.

Execute Command as

Click on the drop down arrow and select the mode of executing the script:

- Batch File - Use this option to execute a shell script in a system command window; environment variables can be accessed by commands in this script
- Process - Use this option to execute the command as a single program; the command should specify the path to the program, plus any command line arguments

Default Directory

Browse for or type in the default directory path in which the build script process will run.

Parse Output

Click on the **drop-down arrow** and select a method for

automatically parsing the compiler output.

If you select this option, output from the script is logged in the **System Output** window; Enterprise Architect parses the output according to the syntax you specify.

Remote Host

Type in the ID of the remote host system and its port; for example, mypc01:7777.

If you set this property to #SYSTEMHOST#, the script is sent to the Windows Satellite Service when running on Windows, and the Linux Satellite Service when running under Wine. The service IDs and Ports are defined in the 'Private Options - Services' section of the Analyzer Scripts Editor.

Deploy after Build

Check this box to cause the Deploy script to be executed immediately after this Build script completes.

Notes

To execute the Build script, click on the Package in the

Browser window and either:

- Right-click on any Toolbar and select 'Analyzer Toolbars | Build', or
- Press **Ctrl+Shift+F12** or
- Select the 'Execute > Source > Build > Build' ribbon option

Cleanup Script

Incremental builds are the practice of only building those assets that have changed in some way. There are times, however, when there is cause to build everything again from scratch. This command is used for those occasions, to remove the binaries and intermediary files associated with a particular build or configuration. The project can then be rebuilt. When you execute the 'Rebuild' menu option on a script, the command(s) you specify in this field are executed, followed immediately by the 'Build' command from the same Analyzer script. Some compilers have options do this for you. Visual studio for example has the "/clean" command line switch.

This is an example script: devenv.com /Clean Debug MyProject.sln

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Build > Clean' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Build > Clean' page

Ribbon	Develop > Preferences > Analyzer > Edit

	Analyzer Scripts Execute > Source > Build > Clean
Analyzer Context Menu	Clean
Keyboard Shortcuts	**Shift+F12**

Aspects

Aspect	Detail
Clean	Enter the command to be executed when you select 'Clean' from the script context menu.
Execute Command as	Click on the **drop-down arrow** and select the appropriate option: • Batch File - Use this option to create a shell script that is executed in a system command window; environment variables can be accessed by commands in this script

	• Process - Use this option to run a single program - the command should specify the path to the program, plus any command line arguments; - if the path or arguments contain spaces, surround them with quotes - for example: "c:\program files (x86)\java\bin\javac.exe"
Default Directory	Defaults to the value entered for the Build script. If a value has not been set for the Build script, browse for or type in the default directory path in which the Clean script process will run.
Parse Output	Click on the **drop-down arrow** and select a method for automatically parsing the compiler output. If you select this option, output from the script is logged in the **System Output** window; Enterprise Architect parses the output according to the syntax you specify.
Remote Host	Type in the ID of the remote host system and its port; for example, mypc01:7777. If you set this property to #SYSTEMHOST#, the script is sent to the Windows Satellite Service when

	running on Windows, and the Linux Satellite Service when running under Wine. The service IDs and Ports are defined in the 'Private Options - Services' section of the Analyzer Scripts Editor.

Test Scripts

These sections explain how to configure the 'Test' page of an Analyzer Script for performing unit testing on your code. Most users will apply this to NUnit and JUnit test scenarios. Enterprise Architect accepts the output from these systems and can automatically add to and manage each unit test case history. To view the case history, you would select the test case Class element and press **Alt+2** > Testing.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Test > Test' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Test > Test' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Analyzer Context Menu	Test

Keyboard Shortcuts	**Shift+F12**

Actions

Test Type the Test command or script in this field. For example:

- NUnit - "C:\Program Files\NUnit\bin\nunit-console.exe" "bin\debug\Calculator.exe"
- JUnit - java junit.textui.Testrunner %N

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.

If you include the string %N in your test script it is replaced by the fully namespace-qualified name of the currently selected Class when the script is executed.

Execute Command As Click on the **drop-down arrow** and select the appropriate option:

- Batch File - Use this option to create a

shell script that is executed in a system command window; environment variables can be accessed by commands in this script

- Process - Use this option to run a single program - the command should specify the path to the program, plus any command line arguments; if the path or arguments contain spaces, surround the path with quotes - for example: "c:\program files (x86)\java\bin\javac.exe"

Default Directory Defaults to the value entered for the Build script. If a value has not been set for the Build script, browse for or type in the default directory path in which the Clean script process will run.

Parse Output When a parser is selected, output of nUnit and jUnit tests can be parsed, saved and managed from the model; (**Alt+2** > Testing). Be aware that output is only captured when a parser is selected.

Remote Host Type in the ID of the remote host system and its port; for example, mypc01:7777.

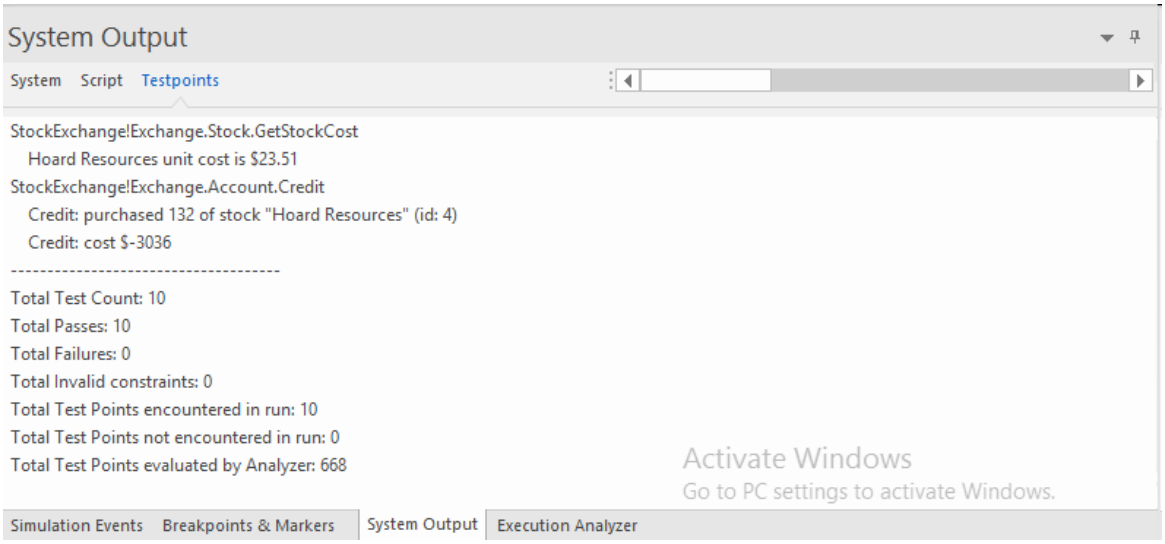
If you set this property to #SYSTEMHOST#, the script is sent to

	the Windows Satellite Service when running on Windows, and the Linux Satellite Service when running under Wine. The service IDs and Ports are defined in the 'Private Options - Services' section of the Analyzer Scripts Editor.
Build First	Select to ensure that the Package is compiled each time you run the test.

Testpoints Output

The '**Testpoints**' page of the Analyzer Script helps you to configure the output of a **Testpoint** run.

By default the output is logged to the **System Output** window, as in this example.



Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Test > **Testpoints**' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Test > Testpoints' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts

	Execute > Tools > Analyzer
Keyboard Shortcuts	**Shift+F12**

Options

Option	Description
Output	You can select from two options: • 'Screen' (the default) - The output is directed to the '**Testpoints**' tab of the **System Output** window • 'File' - The output is directed to file
Folder	Click on the folder to use for **Testpoint** log files.
Filename	Enter the name to use for the **Testpoint** log files.
Overwrite	When this option is selected, the file specified is overwritten each time a **Testpoint** run is performed.

Auto Number	When this option is selected, the **Testpoint** output is composed of the filename you specify and the number of the Test run; each time you perform a Test run the number is incremented.
Prefix trace output with function	When this option is selected, any trace statements executed during the **Testpoint** run are prefixed with the current function call.

Debug Script

The process of configuring the Debug section of an Analyzer Script is usually a one-time affair that rarely has to be revisited. So once you have your script working, you probably won't have to think about it again. The details you provide are not complicated, yet defining a script provides access to a great many benefits such as:

- Debugging
- Sequence diagram recording
- Executable StateMachine execution and simulation
- Test domain authoring and recording
- Behavioral profiling of processes on a variety of runtimes

All you need to do is select the appropriate platform and enter some basic details. The debugger platforms you can use include:

- Java
- Java Debug Wire Protocol (**JDWP**)
- Microsoft .NET **Debugger**
- Microsoft Native Code Debugger (C++, C, VB)
- Mono
- The PHP Debugger
- The GNU Debugger (GDB)

Access

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**F6**

Notes

- An Analyzer script is not necessary for debugging Enterprise Architect model scripts such as JavaScript or VBScript

Operating System Specific Requirements

The Enterprise Architect debugger is able to operate on a number of different platforms. This table describes the individual requirements for debugging on each platform.

Platforms

Platform	Detail
Microsoft .NET	• Microsoft™ .NET Frameworks 4.0, 3.5 and 2.0 • Language support: C, C#, C++, J#, VB.NET
Java	• Java SE Development Kit from Oracle™ (version 5.0 minimum) (either 32-bit or 64-bit JDK) The Java Platform **Debugger** Architecture (JPDA) was introduced in Java SE version 5.0. The JPDA provides two protocols for debugging; the Java Virtual Machine Tools Interface (**JVMTI**), and the Java Debug Wire Protocol (**JDWP**).

	Enterprise Architect's debugger supports both protocols.
GNU Debugger (GDB)	Enterprise Architect supports debugging using the GNU **Debugger**, which enables you to debug your applications under Linux either locally or remotely. Requires GDB version 7.0 or higher. The source code file path must not contain spaces.
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.
PHP	Enterprise Architect enables you to perform local and remote debugging of PHP scripts in web servers. Requires web server to be configured to support PHP. Requires PHP to be configured to support XDebug PHP (3rd party PHP extension).

Notes

- The debugging facility is available in all editions of Enterprise Architect

UAC-Enabled Operating Systems

The Microsoft operating system Windows 7 provides User Account Control (UAC) to manage security for applications.

The Enterprise Architect **Visual Execution Analyzer** is UAC-compliant, and users of UAC-enabled systems can perform operations with the Visual Execution Analyzer 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.

Log in as Administrator

Step	Action
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.

Step	Action
1	Right-click on the Enterprise Architect icon and select the 'Properties' 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.

WINE Debugging

Configure Enterprise Architect to debug under WINE

Step	Action
1	At the command line, run $ winecfg.
2	Select the 'Applications' tab. Add the Enterprise Architect executable 'EA.exe' from the Enterprise Architect installations folder. Then add these programs from the **VEA** sub directories: • SSampler32.exe • SSampler64.exe • SSProfiler32.exe • SSProfiler64.exe
3	Select each program in turn, then switch to the 'Libraries' tab. Ensure these values are listed with a (native, built-in) precedence: • dbghelp • msxml4 • msxml6

4	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.
5	Copy any Side-By-Side assemblies that are used by the application.

Permissions

An installation of Enterprise Architect contains some native Linux programs that provide building and debugging services to Enterprise Architect under Wine. These programs need to checked using the Linux file system or shell to ensure they have the 'Execute' permission set appropriately. The programs are located in the "**VEA**/x86/linux" subdirectory of the Enterprise Architect installation.

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 drop-down menu on the **Debug window** toolbar 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.

Notes

- If WINE crashes, the back traces might not be correct
- 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

Java

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

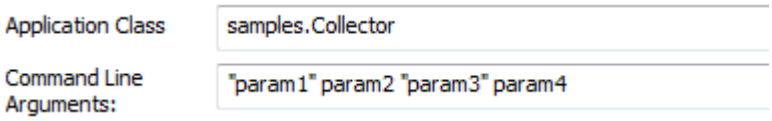
General Setup for Java

The general setup for debugging Java Applications supports two options:

- Debug an Application
- Attach to an application that is running

Option 1 - Debug an Application

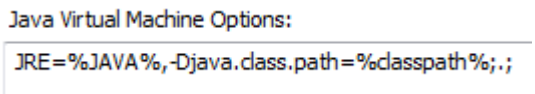
Field	Action
Debugger	Select Java.
x64	Select this checkbox if you are debugging a 64-bit application. Deselect the checkbox if you are debugging a 32-bit application.
Mode	Select Run.
Default Directory	This path is added to the class path property when the Java Virtual Machine is created.
Application Class	Identify the fully qualified Class name to debug; the Class must have a method

	declared with this signature: public static void main(String()); 
Command Line Arguments	Specify any parameters to be passed to the main method of the Application Class. Parameters containing spaces should be surrounded with double quotes.
Java Virtual Machine Options	Specify command line options for Virtual Machine creation. You also must provide a parameter for the Java Runtime Environment (JRE) as the path to be searched for the jvm.dll; this is the DLL supplied as part of the runtime environment or JDK from Sun Microsystems™. The JRE parameter can be either: • An Enterprise Architect-defined Local Path • An absolute file path (with no double quotes) to the installation folder of the Java JDK to be used for debugging The JRE parameter must point to the installation folder for the Java JDK. A

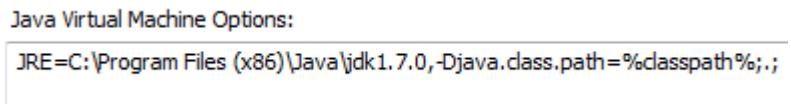
JDK installation is necessary for debugging to succeed. The JRE should not point to the installation of the public Java Runtime Environment, if that is installed. Environment variables can be used when specifying the VM startup options, such as classpath.

For example, using:

- An Enterprise Architect Local Path JAVA and an environment variable classpath:



- Or an absolute path to the JDK installation directory and an environment variable classpath:



In these two examples, the debugger will create a virtual machine using the JDK located at the value of the JRE parameter.

If no classpath 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.

If source files and .class files are located

	under different directory trees, the classpath property MUST include both root path(s) to the source and root path(s) to binary class files.

Option 2 - Attach to Virtual Machine

There is very little to specify when attaching to a VM; however, the VM must have the Sparx Systems debugging agent loaded.

Field	Action
Debugger	Select Java
Mode	Select Attach to Virtual Machine

Advanced Techniques

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

- Attach to Virtual Machine
- Internet Browser Java Applets

Attach to Virtual Machine

You can debug a Java application by attaching to a process that is hosting a Java Virtual Machine; you might want to do this for attaching to a webserver such as Tomcat or JBOSS.

The Java Virtual Machine Tools Interface from Sun Microsystems is the API used by Enterprise Architect; it allows a debugging agent to be specified when the JVM is created.

To debug a running JVM from Enterprise Architect, the Sparx Systems' debugging agent must have been specified as a startup option to the JVM when it was started; how this is accomplished for products such as Tomcat and JBOSS should be provided by that product's own documentation.

For java.exe, the command line option to load the Enterprise Architect debugging agent could be (depending on your environment):

- -agentpath:"c:\program files\sparx systems\ea**VEA**\x86\SSJavaProfiler32"
- -agentpath:"c:\program files (x86)\sparx systems\ea\VEA\x86\SSJavaProfiler32"
- -agentpath:"c:\program files (x86)\sparx systems\ea\VEA\x64\SSJavaProfiler64"

The appropriate option will depend on your operating system and whether you are working on a 32-bit application or a 64-bit application.

Alternatively, if you add the appropriate VEA directory to

your PATH environment variable you can choose to use:

- -agentlib:SSJavaProfiler32
- -agentlib:SSJavaProfiler64

It is not necessary to configure an Analyzer Script when you attach to a Virtual Machine; you can just use the **Attach button** on one of the Analyzer toolbars.

If you configure an Analyzer Script, there are only two things that must be selected:

- Select 'Java' as the debugging platform
- Choose the 'Attach to Virtual Machine' option

Internet Browser Java Applets

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

Attach to the browser process hosting the Java Virtual Machine (JVM) from Enterprise Architect

Step	Action
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**.
4	On the installed version to use, include one of these options in the 'Runtime Parameters' field, depending on your environment and whether you are working on a 32-bit application or a 64-bit application: -agentpath:"c:\program files\sparx

	systems\ea**VEA**\x86\SSJavaProfiler32" -agentpath:"c:\program files (x86)\sparx systems\ea\VEA\x86\SSJavaProfiler32" -agentpath:"c:\program files (x86)\sparx systems\ea\VEA\x64\SSJavaProfiler64"
5	In this field add the required Class paths. At least one of these paths should include the root path of the source files to use in debugging.
6	Set breakpoints.
7	Launch the browser.
8	Attach to the browser process from Enterprise Architect.

Working with Java Web Servers

If you are debugging Java web servers such as JBOSS and Apache Tomcat (both Server configuration and Windows Service configuration) in Enterprise Architect, apply these configuration requirements and procedures.

Note: The debug and record features of the **Visual Execution Analyzer** are not supported for the Java server platform 'Weblogic' from Oracle.

Attach to process hosting the Java Virtual Machine from Enterprise Architect

Step	Action
1	Build binaries for the web server code to be debugged, with debug information.
2	Launch the server with the 'Virtual Machine startup' option, described in *Server Configuration*.
3	Import source code into the Enterprise Architect Model, or synchronize existing code.
4	Set breakpoints.

5	Launch the client.
6	Attach to the process from Enterprise Architect.

Server Configuration

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

- Any VM to be debugged, created or hosted by the server must have the Sparx Systems Agent command line option specified or in the VM startup option (that is:
  -agentlib:SSJavaProfiler32 or
  -agentlib:SSJavaProfiler64)
- 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

Analyzer Script Configuration

Using the 'Debug' tab of the 'Build Script' dialog, create a script for the code you have imported and:

- Select the 'Attach to process' radio button and, in the field below it, 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.

Run the Debugger

The breakpoints could show a question mark. In this case the Class might not have been loaded yet by the VM. If the question mark remains even after you are sure the Class containing the breakpoint has been loaded, then either:

- The binaries being executed by the server are not based on the source code
- The debugger cannot reconcile the breakpoint to a source file (check Class paths), or
- The JVM has not loaded the Sparx Systems agent

Step	Action
1	Run the server and check that the server process has loaded the Sparx Systems Agent: DLL SSJavaProfiler32.DLL or SSJavaProfiler64 Use 'Process Explorer' or similar tools to prove that the server process has loaded the agent.
2	In Enterprise Architect, open the source code and set some breakpoints.
3	Click on the **Run Debug button** in Enterprise Architect. The 'Attach To Process' dialog displays.
4	Select the server process hosting the application.
5	Click on the **OK button**. A confirmation message displays in the **Debug window**, stating that the process has been attached.

JBOSS Server

In this JBoss example, for a 32-bit application, the source code for a simple servlet is located in the directory location:

C:\Benchmark\Java\JBOSS\Inventory

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

C:\JBOSS\03b-dao\build\distribution

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.

This is an excerpt from the command file that executes the JBOSS server; the Class to be debugged is at:

com/inventory/dto/carDTO

Therefore, the root of this path is included in the JBOSS_CLASSPATH.

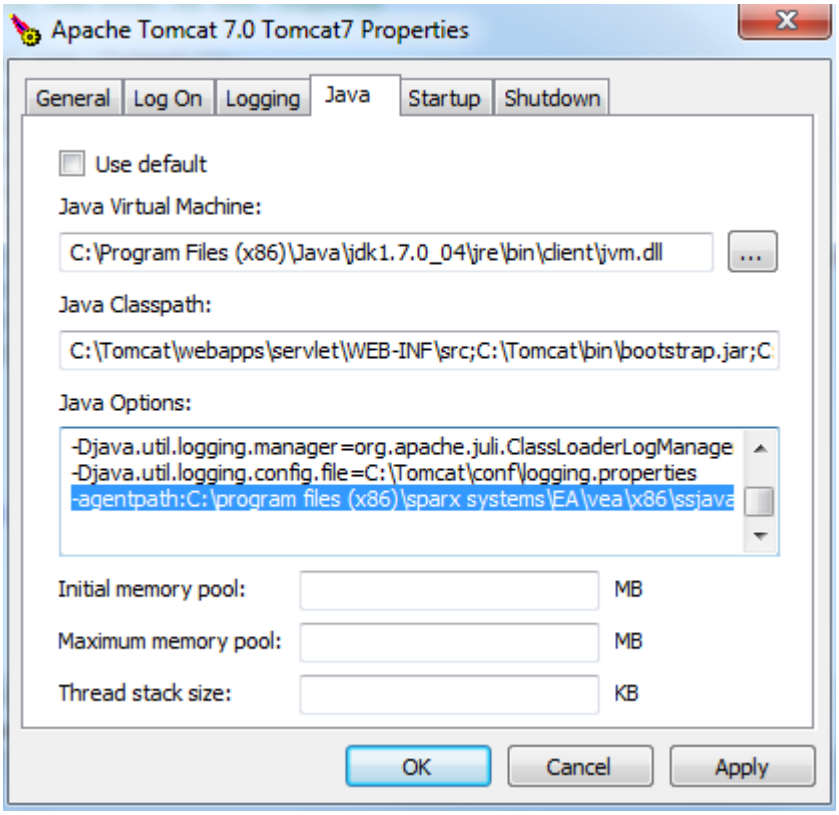
Example Code

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_CLASSPA
TH%;%JAVAC_JAR%;%RUNJAR%;
)
set JAVA_OPTS=%JAVA_OPTS% -agentpath:"c:\program
files\sparx systems\vea\x86\ssjavaprofiler32"
```

Apache Tomcat Server

The Apache Tomcat Server can be configured for debugging using the Java debugger in Enterprise Architect. This example shows the configuration dialog for Apache Tomcat 7.0 on a PC running Windows 7.



These three points are important:

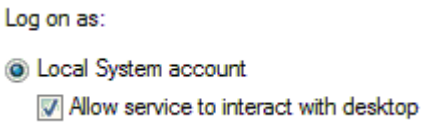
- The 'Java Virtual Machine' specifies the runtime from an installation of the Java JDK
- The source path to any servlet to be debugged is added to Java Classpath; in this case we add the path to the Tomcat servlet: c:\tomcat\webapps\servlet\WEB-INF\src
- The 'Java Options' include the path to the Sparx Systems debugging agent:

```
-agentpath:c:\program files (x86)\sparx
systems\vea\x86\ssjavaprofiler32
```

Apache Tomcat Windows Service

Configuration

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.

.NET

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

- General Setup for .NET
- Debugging an Unmanaged Application
- Debug COM Interop
- Debug ASP .NET

General Setup for .NET

This is the general setup for debugging Microsoft .NET applications. You have two options when debugging:

- Debug an application
- Attach to an application that is running

Option 1 - Debug an application

Field	Action
Debugger	Select Microsoft .NET as the debugging platform.
x64	Select this checkbox if you are debugging a 64-bit application. Deselect the checkbox if you are debugging a 32-bit application.
Mode	Select the Run radio button.
Default Directory	This is set as the default directory for the process being debugged.
Application Path	Select and enter either the full or the relative path to the application

	executable. • If the path contains spaces, specify the full path; do not use a relative path • If the path contains spaces, the path must be enclosed by quotes
Command Line Arguments	Parameters to pass to the application at start up.
Show Console	Create a console window for the debugger; not applicable to attaching to a process.
Symbol Search Paths	Specify any additional paths to locate debug symbols for the debugger; separate the paths with a semi-colon.

Option 2 - Attach to an application that is running

Field	Action
Debugger	Select Microsoft .NET as the debugging platform.

x64	Select this checkbox if you are debugging a 64-bit application. Deselect the checkbox if you are debugging a 32-bit application.
Mode	Select the Attach to Process radio button.

Debugging an Unmanaged Application

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 specify should match the version loaded by the managed code invoked by the debuggee.

This is a sample config file:

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

'Version' is the version of the CLR targeted by your plug-in or COM code.

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 Plug-ins and ActiveX components.

Debug .NET Managed Code Executed Using COM

Step	Action
1	Create a Package in Enterprise Architect and import the code to debug.
2	Ensure the COM component is built with debug information.
3	Create a Script for the Package.
4	In the 'Debug | Platform' page, you can select 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 you are using:

- An In-Process COM server, attach to the client process
- 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.

Notes

- 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

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, you can import from the root and include both ASP code and web pages at the same time.

It is necessary to launch the client first, as the ASP .NET service process might not already be running; load the client using your browser - this ensures that the web server is running.

In the debug set up you would then select the 'Attach' radio button. When this choice is selected, the debugger will prompt you each time for the process to debug.

Click on the **Debug window** **Run button** to start the debugger; the 'Attach To Process' dialog displays.

The name of the process varies across Microsoft operating systems, as explained in the *ASP .NET SDK*; for example, on Windows XP, the name of the process resembles 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 should be red, indicating they have been bound.

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.

Notes

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 synchrony; try rebuilding and re-importing source code

The Mono Debugger

Mono is a software platform sponsored by the .NET Foundation to facilitate cross-platform development. It is popular with game developers for its rich gaming, API-based and portability features.

Enterprise Architect provides support to the Mono community by providing a modern environment for both modeling and developing software. Existing projects can be imported, built and debugged natively on both Linux and Windows.

Overview

Debugging under Mono involves the cooperation of three processes. The Mono runtime manages the application and communicates using a socket protocol with the Enterprise Architect Debugger, which in turn communicates with Enterprise Architect acting as the front end. When you launch Mono you need to direct it to support debugging, which you achieve using a command line directive in which you name the host and Port number that Mono should listen on. The host can be omitted, in which case Mono will accept connections from any IP address. The host can have the value 'localhost' to restrict connections to the same machine. The Port number is a number of your choosing.

The host and Port number are the important pieces of information, as they are used when configuring the Analyzer

Script.

Requirements for Windows

- Enterprise Architect (version 14 minimum)
- Mono for Windows (version 5.4 minimum)

Requirements for Linux

- Enterprise Architect (version 14 minimum)
- Mono for Linux (version 5.4 minimum)
- Wine for Linux

The Runtime Host Page

This page is optional and is only useful where Mono and Enterprise Architect will be running on the same machine. It provides the ability to run Mono first with the required debugging directives, before the Enterprise Architect debugger is started. After the debugger connects, it resumes the Mono runtime, which has been started as suspended. If the application runs on a different machine from the Enterprise Architect you are using, you should clear this section.

Debugging Configuration Linux

Debugger Configuration

This section describes the Debug section of an Analyzer Script in respect to debugging Mono under Linux. Fields that are not listed here are not required.

Debugger	Select 'Mono'.
Default Directory	This is the fully qualified native Linux path where the application is located in Unix format.
Connection	• port: the debugging Port • host: the name or IP address of the machine where Mono runs ('localhost' if the machine is the same) • localpath: the root path of the source code in Windows format; this is the path to the source files that you use to set breakpoints in the code editor in Enterprise Architect • remotepath: the root path of the source code in Unix format, this being the path to the source files used to build the

	program under Linux These paths are returned during debug events, and are then mapped to the local path, so that Enterprise Architect can display the source file during a breakpoint or step - both parameters can specify the same physical source file root, but must use the Windows or Unix format for each field • shutdown: (true or false); when true the VM is terminated when the **Debugger** is stopped • timeout: the timeout in milliseconds for socket calls • output: the Wine / Windows path of the log file to write to • logging: (true or false); when true, extra messages are logged in the **Debug window** and socket messages are logged to the specified output file

Starting Mono Automatically

You can configure Enterprise Architect to start Mono for you when you start the debugger. You do this by

configuring the 'Runtime Host' page of your Analyzer Script. The format of the commands is described here:

cd path-to-program

/usr/bin/mono --debug --debugger-agent=transport=dt_socket,address=*host:port*,server=y,suspend=y *program*

where:

- *path-to-program* is the directory path where the program is located

- *host* is one of these:

- localhost
- an IP address
- a networked machine name

- *port* is the Port for the socket

- *program* is the name of the application (such as MonoProgram.exe)

Starting Mono Manually using the Command Line

You can start Mono manually from a console. Locate the program in your file explorer, then open a console at that location. The format of the command line is described here:

/usr/bin/mono --debug --debugger-agent=transport=dt_socket,address=*host:port*,ser

ver=y,suspend=y *program*

where *host* is one of these:

- localhost
- an IP address
- a networked machine name

port is the Port for the socket and *program* is the name of the application (for example, MonoProgram.exe).

Debugging Configuration Windows

Debugger Configuration

This section describes the Debug section of an Analyzer Script in respect to debugging Mono under Windows. Fields that are not listed here are not required.

Field	Description
Debugger	Select 'Mono'.
x64	Select if the program to be debugged is a 64bit executable.
Run or Attach	Choose 'Run' to name the program to launch. Choose 'Attach' if you will always attach to a running process.
Default Directory	The default directory that the program will take when it runs.
Application Path	The full path of the Mono application.
Command Line Arguments	Any parameters to pass to the program. If the parameters contain spaces, surround them in double quotes (")

Starting Mono Automatically

You can configure Enterprise Architect to start Mono for you when you start the debugger. You do this by configuring the 'Runtime Host' page of your Analyzer Script. The format of the commands is described here:

cd *path-to-program*

mono --debug --debugger-agent=transport=dt_socket,address=*host:port*,server=y,suspend=y *program*

where:

- *path-to-program* is the directory path where the program is located

- *host* is one of these:

- localhost
- an IP address
- a networked machine name

- *port* is the Port for the socket

- *program* is the name of the application (such as MonoProgram.exe)

Starting Mono Manually using the command

line

You can start Mono manually from a console. Locate the program in your file explorer, then open a console at that location. The format of the command line is described here:

mono --debug --debugger-agent=transport=dt_socket,address=*host:port*,server=y,suspend=y *program*

where *host* is one of these:

- localhost
- an IP address
- a networked machine name

port is the Port for the socket and *program* is the name of the application (for example, MonoProgram.exe).

The PHP Debugger

The Enterprise Architect PHP **Debugger** enables you to debug PHP.exe scripts. This section discusses basic set up and the various debugging scenarios that are commonly encountered; the scenarios concern themselves with the mapping of file paths, which is critical to the success of a remote debugging session.

- Script Setup
- Local Windows Machine (Apache Server)
- Local Windows Machine (PHP.exe)
- Remote Linux Machine (Apache Server)
- Remote Linux Machine (PHP.exe)

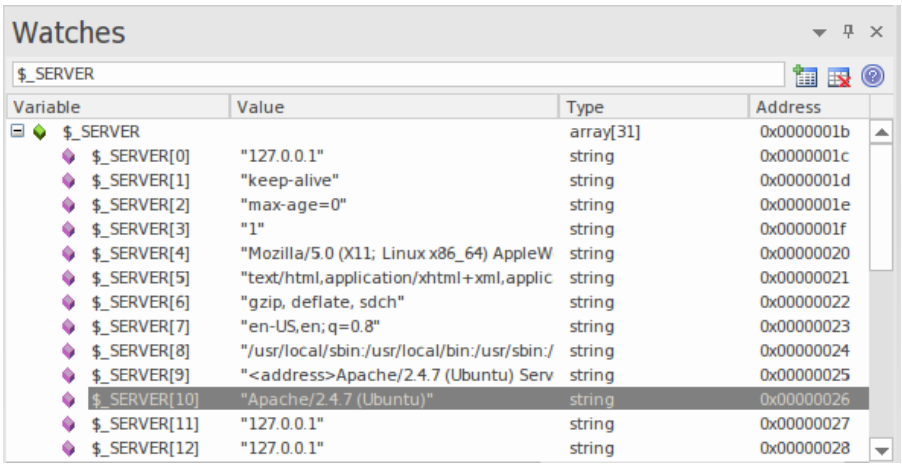
Setup and Scenarios

Scenario	Details
Script Setup	An Analyzer Script is a basic requirement for debugging in Enterprise Architect; you create a script using the toolbar of the Execution Analyzer. Select PHP.XDebug as the debugging platform; when you select this platform the property page displays these connection settings:

	• host - localhost - The adaptor that Enterprise Architect listens on for incoming connections from PHP • localpath - %LOCAL% - Specifies the local file path to be mapped to a remote file path; this is a remote debugging setting - for local debugging, clear the value, the value is a placeholder and you should edit it to fit your particular scenario • remotepath - %REMOTE% - Specifies the remote file path that a local file path is to be mapped to; this is a remote debugging setting - for local debugging, clear the value, the value is a placeholder and you should edit it to fit your particular scenario • logging - Enter true or false to enable logging of communication from XDebug server • output - names the file path on the remote machine to be used with the logging option; this file will always be overwritten
Local Machine Apache Server	In this situation, consider this configuration: • O/S: Windows7

	• Network computer name: MyPC • Network share MyShare mapped to c:\myshare • Source files in Enterprise Architect have been imported from c:\myshare\apache\myapp\scripts • Apache document root is set to //MyPC/MyShare/apache In this scenario an Analyzer Script for the connection parameters might be configured as: • host: localhost • port: 9000 • localpath: c:\myshare\apache\ • remotepath: MyPC/MyShare/apache/
Local Machine PHP.EXE	In this scenario an Analyzer Script for the connection parameters might be configured as shown, as file paths always map to same physical path: • host: localhost • port: 9000 • localpath: • remotepath:
Remote Linux	In this situation consider this configuration:

Machine Apache Server	Local Machine: • O/S: Windows7 • Source files in Enterprise Architect have been imported from c:\myshare\apache\myapp\scripts Remote Machine: • O/S: Linux • Apache document root is set to home/apache/htdocs • Source files in Apache are located at home/apache/htdocs/myapp/scripts In this scenario an Analyzer Script for the connection parameters might be configured as: • host: localhost • port: 9000 • localpath: c:\myshare\apache\ • remotepath: home/apache/htdocs/
Remote Linux Machine PHP.exe	In this situation consider this configuration: • Local Machine • O/S: Windows7 • Source files in Enterprise Architect have been imported from c:\myshare\apache\myapp\scripts

	• Remote Machine • O/S: Linux • Source files in Apache located at home/myapp/scripts In this scenario an Analyzer Script for the connection parameters might be configured as: • host: localhost • port: 9000 • localpath: c:\myshare\apache\ • remotepath: home/
PHP Global variables	When you are at a breakpoint, you can examine the values of PHP globals using the **Analyzer Watch window**. To list every global, type either 'globals' or 'superglobals' into the field. To show an individual item, enter its name. This image shows the value of the PHP environment variable $_SERVER being displayed. 

PHP Debugger - System Requirements

This topic identifies the system requirements and operating systems for the Enterprise Architect PHP debugger.

System Requirements:

- Enterprise Architect version 9
- PHP version 5.3 or above
- PHP zend extension XDebug 2.1 or above
- For web servers such as Apache, a server version that supports the PHP version

Supported Operating Systems:

- Client (Enterprise Architect)
- Microsoft Windows XP and above
- Linux running Crossover Office
- Server (PHP)
- Microsoft Windows XP and above
- Linux

PHP Debugger Checklist

This topic provides a troubleshooting guide for debugging PHP scripts in Enterprise Architect.

Check Points

Check Point	Details
System Requirements	• Apache HTTP Web Server version 2.2 • PHP version 5.3 or above • XDebug version 2.1.1
Enterprise Architect	• The model has an Analyzer Script configured to use the PHP XDebug platform • PHP source code has been imported into the model (for recording and testpoints) • When the PHP XDebug platform is selected from the 'Analyzer Script' dialog, default runtime settings are listed in the 'Connection' field: localpath:%LOCAL% remotepath:%REMOTE% Either define local paths for these default

	variables or edit the script to provide actual paths. For example: local source, remote source localpath:c:\code samples\vea\php\sample remotepath:webserver/sample • 'webserver' is a network or local share • 'sample' is a folder below share
PHP	In order to debug PHP scripts in Enterprise Architect, it is a requirement that the PHP is configured properly to load the XDebug extension. Settings similar to these should be used: • [xdebug] • xdebug.extended_info=1 • xdebug.idekey=ea • xdebug.remote_enable=1 • xdebug.remote_handler=dbgp • xdebug.remote_autostart=1 • xdebug.remote_host=X.X.X.X • xdebug.remote_port=9000 • xdebug.show_local_vars=1 The IP address X.X.X.X refers to and should match the host specified in the model Analyzer Script. The IP address is the address XDebug

	connects with and the same address the Enterprise Architect PHP agent listens on.
Apache	For debugging using Apache, these lines should be present in the Apache configuration file, httpd.conf: LoadModule php5_module "php_home/php5apache2_2.dll" AddHandler application/x-httpd-php .php PHPIniDir "php_home" The value "php_home" is the PHP installation path (the path where php.ini and apache dll exist).
Troubleshooting	To prevent both PHP and Apache timeouts during a debugging session, these settings might require modification. The settings were used while developing the PHP Debugging agent in Enterprise Architect.
PHP	File: php.ini ; Enterprise Architect prevents PHP timeouts when debugging PHP extensions max_execution_time = 0

	; Enterprise Architect prevents web server timeouts when debugging PHP extensions max_input_time = -1 ; Enterprise Architect logs errors display_errors = On ; Enterprise Architect displays startup errors display_startup_errors = On
Apache	File: httpd.conf ; Enterprise Architect prevents timeouts while debugging php extensions Timeout 60000

The GNU Debugger (GDB)

When debugging your applications you can use the GNU **Debugger** (GDB), which is portable and runs on Unix-like systems such as Linux, as well as on Windows. The GDB works for many programming languages including Ada, Java, C, C++ and Objective-C. Using the GDB, you can debug your applications either locally or remotely.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Debug > Platform' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Debug > Platform' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Context Menu	**Browser window** | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	**Shift+F12**

Set up the GNU Debugger

Task	Details
Set up Script	An Analyzer Script is a basic requirement for debugging in Enterprise Architect; you create a script using the Execution Analyzer toolbar. On the 'Platform' page of the **Execution Analyzer Script Editor**, in the '**Debugger**' field click on the **drop-down arrow** and select 'GDB'.
Define Connection Settings	The property panel displays a number of connection settings for which you provide values. • path - <path> - The complete file path of the GDB executable; you only specify this if the GDB cannot be found in the system path • source - <path>, <path> - The path in which the debugger will search for source files, if they do not reside in the executable directory • remote - F - Set for remote debugging;

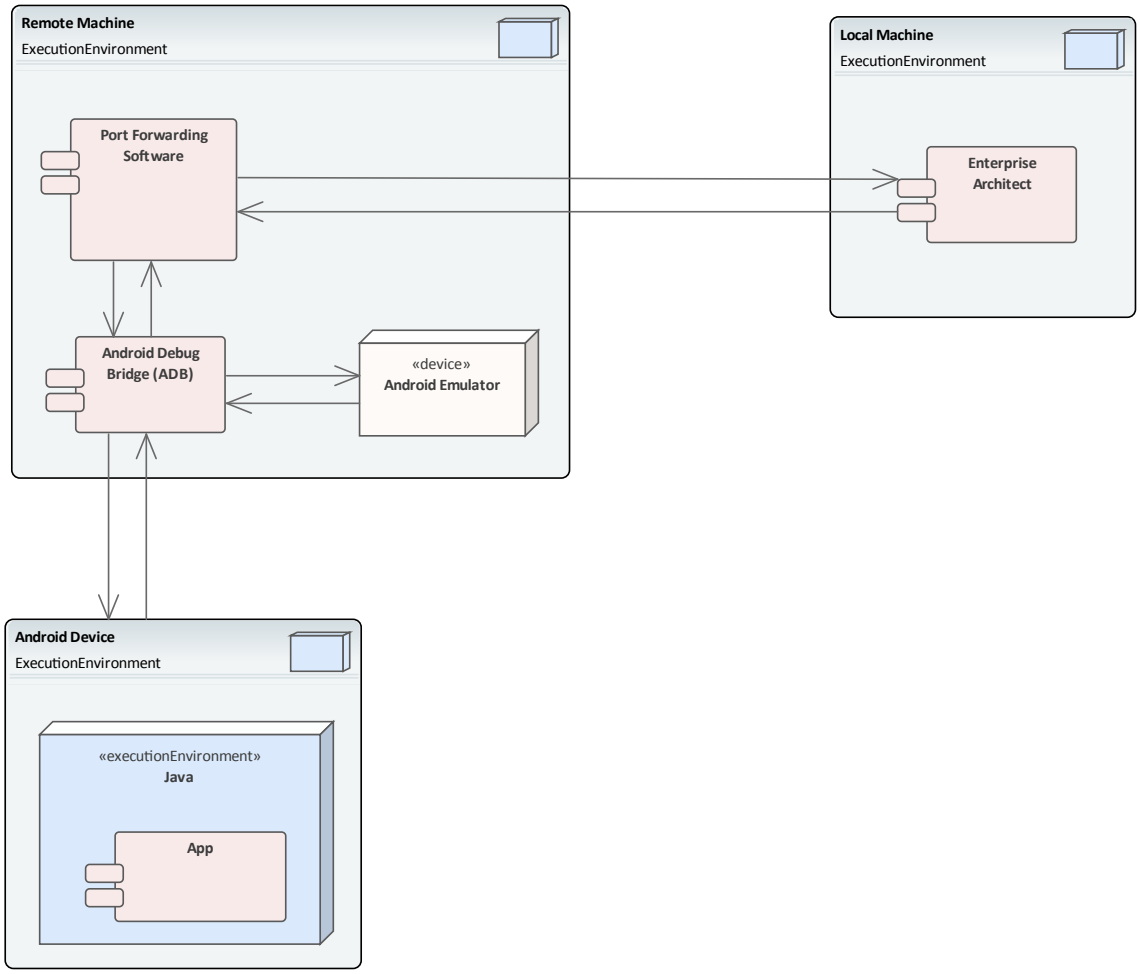
	otherwise leave blank • port - <nnnnn> - The Port to connect to on the remote server • host - localhost - The host name to connect to • fetch - T - Set to retrieve the binary from the remote system • dumpgdb - <path> - The filename to write the GDB output to • initpath - <path> - The complete file path to the gbinit file

Notes

- A requirement of the GDB is that your source code file path does not contain spaces; the debugger will not run correctly with spaces in the file path

The Android Debugger

If you are developing Java applications running on Android devices or emulators, you can also debug them. The Local and Remote machines can be on either a 32-bit platform or a 64-bit platform.



System Requirements

On the Remote machine, this software is required:

- Android SDK, which includes the android debug bridge, ADB (you need to be familiar with the SDK and its tools)

- Java JDK (32 and 64 bit support)
- Port Forwarding software (3rd party)

On the Local machine, this software is required:

- Enterprise Architect Version 10 or higher

Analyzer Script Settings

Field/Button	Action
Debugger	Click on the **drop-down arrow** and select Java (**JDWP**).
Run	Click on this radio button.
Default Directory	Not applicable - leave blank.
Application path	Not applicable - leave blank.
Command Line Arguments	Not applicable - leave blank.
Build first	Not applicable - leave blank.

Show console	Not applicable - leave blank.
Show diagnostic messages	Not applicable - leave blank.
Connection	Not applicable - leave blank.
Port	This is the application port, forward-assigned using **adb** or other means, through which Enterprise Architect and the Android Virtual Machine (VM) can communicate.
Host	Host computer (defaults to localhost) If Android is running on an emulator on a device attached to a networked computer, enter the network name here. By default, debugging will attempt to connect to the Port you specify on the local machine.
Source	This is the source equivalent of the classpath setting in Java. The root to each source tree should be listed. If more than one is specified, they should be separated by a semi-colon; that

	is: c:\myapp\src;c:\myserver\src You must specify at least one root source path. When a breakpoint occurs the debugger searches for the java source in each of the source trees listed here.
Logging	Enables logging additional information from debugger possible values: true,false,1,0,yes,no
Output	Specifies the full name of the local log file to be written. The folder must exist or no log will be created. The log file typically contains a dump of bytes sent between debugger and VM.
Platform	If you are debugging Java running under any android scenario, select Android. For all other scenarios, select Java.

Configure Ports for Debugging - Port Forwarding (Local)

The debugger can only debug one VM at a time; it uses a single Port for communication with the VM. The Port for the application to be debugged can be assigned using ADB, which is supplied with the Android SDK.

Before debugging, start the application once in the device. When the app starts, discover its process identifier (pid):

adb jdwp

The last number listed is the pid of the last application launched; note the pid and use it to allow the debugger to connect to the VM:

- adb forward tcp:port jdwp:pid
  - port = Port number listed in analyzer script
  - pid = process id of the application on the device

Configure Ports for Debugging - Port Forwarding (Remote)

To debug remotely, the same procedure should be followed as for the local machine, but the communication requires additional forwarding as the socket created using the **adb** forward command will only listen on the local adapter. The socket is bound to the localhost, and attempts to connect to this Port will be met with 'connection refused' messages.

In order to achieve remote debugging it is necessary to have a proxy running on the remote machine that listens to all incoming connections and forwards all traffic to the adb Port; there are numerous software products available to do this.

Remote debugging with Enterprise Architect will not work unless you have configured a proxy Port forwarder.

Java JDWP Debugger

Java provides two main debugging technologies: an in-process agent-based system called the Java Virtual Machine Tools Interface (**JVMTI**) and a socket-based paradigm called the Java Debug Wire Protocol (**JDWP**). A Java Virtual Machine can name either one of these but not both, and the feature must be configured when the JVM is started.

System Requirements

1. The Enterprise Architect **JDWP** debugger will only be able to communicate with a JVM started with the 'JDWP' option. Here is an example of the command line option:
   java -agentlib:jdwp=transport=dt_socket,address=localhost:9000,server=y,suspend=n -cp "c:\java\myapp;%classpath%" demo.myApp "param1" "param2"
2. The Virtual Machine should not be currently attached to a debugger.
3. It is not possible for a VM to be debugged by Enterprise Architect and Eclipse at the same time.

Analyzer Script Settings

Field/Button	Action
Debugger	Click on the **drop-down arrow** and select Java (**JDWP**).
Run	Click on this radio button to run the debugger when the script is executed.
Default Directory	Not applicable - leave blank.
Application path	Not applicable - leave blank.
Command Line Arguments	Not applicable - leave blank.
Build first	Not applicable - leave blank.
Show console	Not applicable - leave blank.
Show diagnostic messages	Not applicable - leave blank.
Connection	Not applicable - leave blank.

Port	Set the application Port forward-assigned to the VM process during start-up, in the Java command-line options.
Host	Set the host computer (defaults to localhost) If VM is running on a networked computer, enter the network name or url here. By default debugging will attempt to connect to the Port you specify on the local machine.
Source	This is the source equivalent of the *classpath* setting in Java. List the root to each source tree; specify at least one root source path. If you specify more than one, separate them with a semi-colon; for example: c:\myapp\src;c:\myserver\src When a breakpoint occurs the debugger searches for the Java source in each of the source trees listed here.
Logging	Enable or disable logging of additional information from the debugger. Possible values include: • true

	• false • 1 • 0 • yes • no
Output	Specify the full name of the local log file to be written. If the folder does not already exist, no log will be created. The log file typically contains a dump of bytes sent between the debugger and VM.
Platform	Select Java.

Configure Ports for Debugging

The debugger can only debug one VM at a time; it uses a single Port for communication with the VM. The Port for the application to be debugged is assigned when the VM is created.

Local Debugging

Where both Enterprise Architect and the Java VM are

running on the same machine, you can perform local debugging. It is necessary to launch the VM with the **JDWP** transport enabled - see the documentation on *Java Platform **Debugger** Architecture (JPDA)* at Oracle for the command line option specifications. For example:

```
java
-agentlib:jdwp=transport=dt_socket,address=localhost:9000,
server=y,suspend=n -cp
"c:\samples\java\myapp;%classpath%" samples.MyApp
"param1" "param2"
```

In this example the values for the Analyzer script would be 'host: localhost' and 'port:9000'.

Remote Debugging

Where Enterprise Architect is running on the local machine and the Java VM is running on a remote machine, you can perform remote debugging. It is necessary to launch the VM with the **JDWP** transport enabled - see the documentation on JPDA at Oracle for the command line option specifications. Here is an example, where the remote computer has the network name testmachine1:

```
java
-agentlib:jdwp=transport=dt_socket,address=9000,server=y,
suspend=n -cp "c:\samples\java\myapp;%classpath%"
samples.MyApp "param1" "param2"
```

Note the absence of a host name in the address. This means

the VM will listen for a connection from any machine. In this example the values for the Analyzer script would be 'host: testmachine1' and 'port: 9000'.

Tracepoint Output

The Tracepoints page of the Analyzer Script enables you to direct where the output from any Trace statements goes during a debug session.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Debug > Tracepoints' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Debug > Tracepoints' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer > select and run script
Context Menu	**Browser window** | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	**Shift+F12**

Tracepoint properties

Field	Detail
Output	You can select from two options: • 'Screen' (the default) - The output is directed to the **Debug window** • 'File' - The output is directed to file
Folder	Enter the folder to use for Trace statement log files.
Filename	Enter the name to use for the Trace statement log files.
Overwrite	If selected, the specified file is overwritten each time a debug session is started.
Auto Number	If selected, the Trace log file is composed of the filename you specify and a number. Each time you start a debug session, the number is incremented.
Prefix trace output with	If selected, any Trace statements executed during the debug session run are

function	prefixed with the current function call.

Workbench Setup

This topic describes the requirements for setting up the **Object Workbench** on Java and Microsoft .NET.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Debug > Workbench' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Debug > Workbench' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**Shift+F12**

Platforms

Platform	Detail

Platforms Supported	The Workbench supports these platforms: • Microsoft .NET (version 2.0 or later) • Java (JDK 1.4 or later)
Microsoft .NET Workbench	The .NET workbench requires an assembly, which is used to create the workbench items. You specify the path to the assembly on the 'Workbench' page of the Analyzer Script. There are two constraints in using the .NET workbench: • Members defined as struct in managed code are not supported • Classes defined as internal are not supported
Java Workbench	The Java workbench uses the Virtual Machine settings configured in the Analyzer Script 'Debug' page to create the JVM.

Microsoft C++ and Native (C, VB)

You can debug native code only if there is a corresponding PDB file for the executable. A PDB file is created 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

General Setup

This is the general setup for debugging Microsoft Native Applications (C++, C, Visual Basic). You have two options when debugging:

- Debug an application
- Attach to an application that is running

Option 1 - Debug an application

Field	Action
Debugger	Select Microsoft Native as the debugging platform.
x64	Select this checkbox if you are debugging a 64-bit application. Deselect the checkbox if you are debugging a 32-bit application.
Mode	Select the Run radio button.
Default Directory	This is set as the default directory for the process being debugged.
Application	Select and enter either the full or the

Path	relative path to the application executable. • If the path contains spaces, specify the full path; do not use a relative path • If the path contains spaces, the path must be enclosed by quotes
Command Line Arguments	Parameters to pass to the application at start up.
Show Console	Create a console window for the debugger; not applicable for attaching to a process.
Symbol Search Paths	Specify any additional paths to locate debug symbols for the debugger; separate the paths with a semi-colon.

Option 2 - Attach to an application that is running

Field	Action
Debugger	Select Microsoft Native as the debugging

	platform.
x64	Select this checkbox if you are debugging a 64-bit application. Deselect the checkbox if you are debugging a 32-bit application.
Mode	Select the Attach to Process radio button.
Symbol Search Paths	Specify any additional paths to locate debug symbols for the debugger. You could specify a symbol server here if you prefer; separate the paths with a semi-colon or comma.

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 aberrant behavior - 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.

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; you pass additional symbol paths in a semi-colon separated list in the 'Debug' tab.

Merge Script

A Merge command in an Analyzer Script gives users an additional command to perform some action. The merge action is dependent on your requirements.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Merge' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Merge' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Source > Merge
Keyboard Shortcuts	**Ctrl+Alt+M**

Code Miner Script

The Code Miner system uses a set of databases to provide fast and comprehensive access to information derived from existing source code. The Intelli-sense features of Enterprise Architect's code editors and its search tools can make use of the information mined from these databases.

Through the Code Miner script pages, you can specify which Code Miner databases to use with a particular project and you can create, update and add new databases to the Code Miner library. The 'Services' page allows you to specify a local Code Miner library, or that you wish to access the library available through a Code Miner Service.

Different Code Miner details can be specified for each Analyzer Script, so the Code Miner libraries that are used are determined by the *active* Analyzer Script.

Access

On the **Execution Analyzer window**, either:

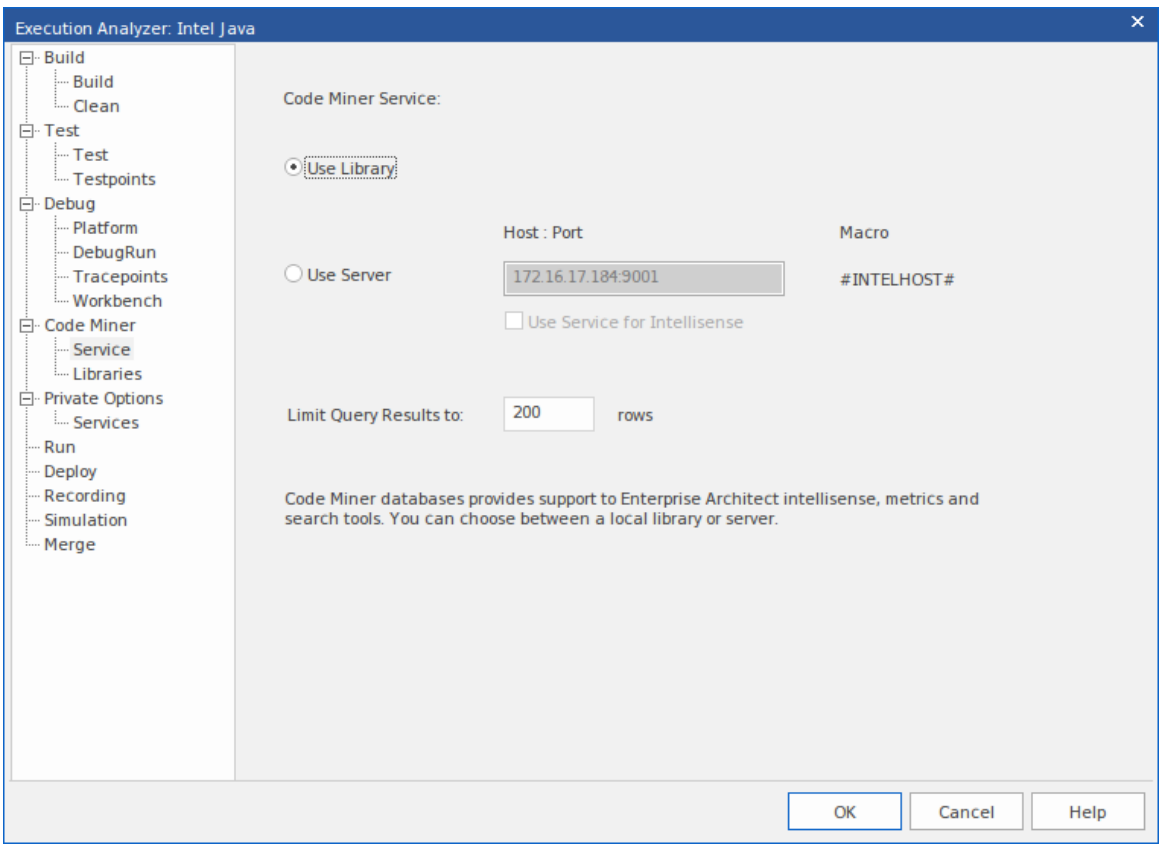
- Locate and double-click on the required script and select the 'Code Miner' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Code Miner' page

Ribbon	Execute > Tools > Analyzer, or

	Develop > Preferences > Analyzer > Edit Analyzer Scripts
Keyboard Shortcuts	**Shift+F12**

Code Miner Service

A Code Miner Library can be used locally, or it can be deployed to a server location where it can service multiple clients. You select the scenario to use on the 'Code Miner Service' page of the Analyzer Script.

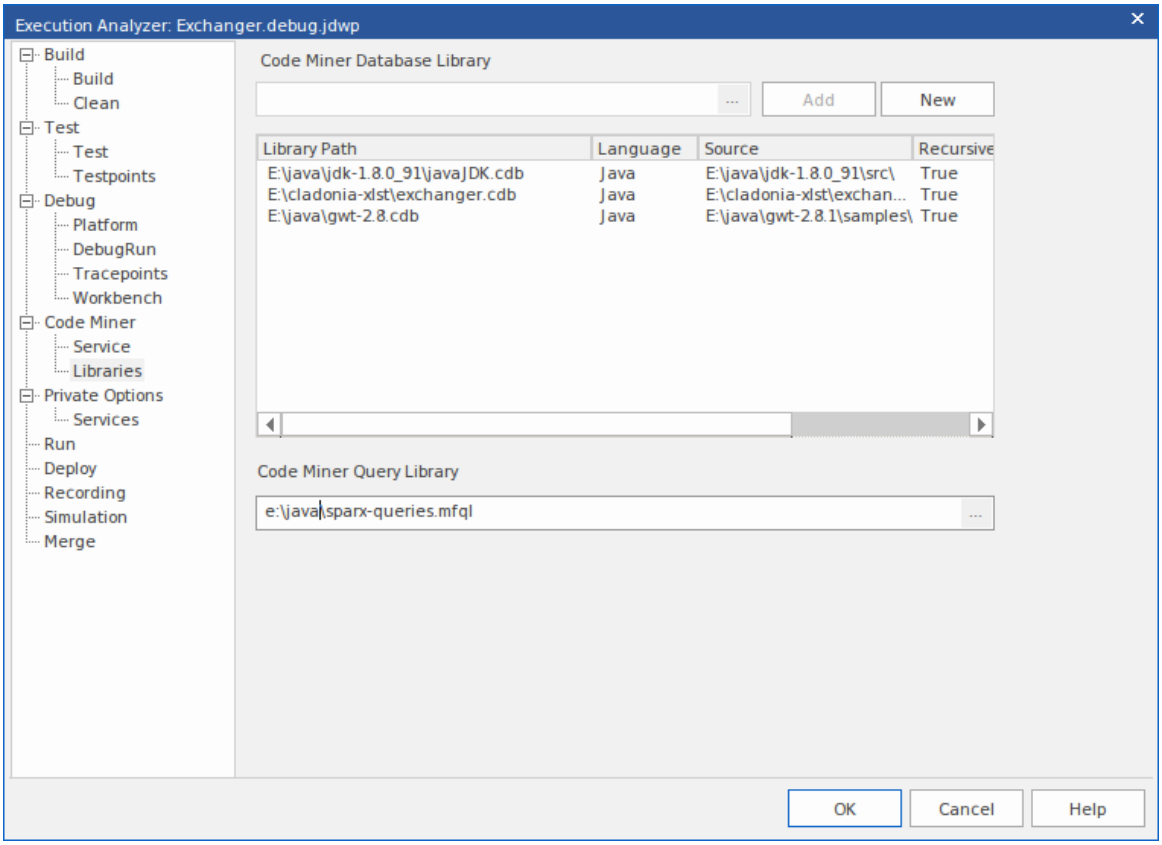


Code Miner Libraries

The Code Miner Database Libraries are a collection of databases containing information derived from source code. Together, these databases form the Code Miner Library used by the Intelli-sense features of Enterprise Architect. Usually, a library would be created for each framework or project. The Code Miner Libraries page allows new databases to be created, and existing databases to be added, updated or removed from a library.

Code Miner Query Libraries are a collection of functions, written in Code Miner's mFQL language, bundled together into one source file.

The Code Miner Database Library and Query Library for a given Analyzer Script are specified on the 'Code Miner | Libraries' page of the **Script Editor**.


Execution Analyzer: Exchanger.debug.jdwp
Build
Build
Clean
Test
Test
Testpoints
Debug
Platform
DebugRun
Tracepoints
Workbench
Code Miner
Service
Libraries
Private Options
Services
Run
Deploy
Recording
Simulation
Merge
Code Miner Database Library
Add
New
Library Path
Language
Source
Recursive
E:\java\jdk-1.8.0_91\javaJDK.cdb
Java
E:\java\jdk-1.8.0_91\src\
True
E:\cladonia-xlst\exchanger.cdb
Java
E:\cladonia-xlst\exchan...
True
E:\java\gwt-2.8.cdb
Java
E:\java\gwt-2.8.1\samples\
True
Code Miner Query Library
e:\java\sparx-queries.mfql
OK
Cancel
Help

Services Script

The 'Services' page of an Analyzer Script describes the default ports used when scripts are created by various **Visual Execution Analyzer** functions (Import project, Generate Executable StateMachine).

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Private Options | Services' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Private Options | Services' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**Shift+F12**

Run Script

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

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Run' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Run' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**Ctrl+Alt+N**

Script elements

Element	Description

Command	This is the command that is executed when you select the 'Execute > Run > Start' ribbon option; at its simplest, the script would contain the location and name of the file to be run.
Examples	These two examples show scripts configured to run a .Net and a Java application in Enterprise Architect. .Net: C:\benchmark\cpp\example_net_1\release\example.exe Java: customer 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 enclosed by quotes.

Notes

- Enterprise Architect provides the ability to start your application normally OR with debugging from the same script; the 'Analyzer' menu has separate options for

starting a normal run and a debug run

Deploy Script

These sections explain how to create a command script for deploying the current Package. The script can be executed by selecting the 'Execute > Source > Build > Deploy' ribbon option or by pressing **Ctrl+Shift+Alt+F12**.

Access

On the **Execution Analyzer window**, either:

- Locate and double-click on the required script and select the 'Deploy' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Deploy' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**Ctrl+Shift+Alt+F12**

Actions

Action	Detail
Execute Command as:	**Process** If the deployment is handled externally, enter the path to the program or batch file to run, followed by any parameters; the program is launched in a separate process. Example: C:\apache-ant-1.7.1\bin\ant.cmd myproject deploy **Batch File** When using this option, you can enter multiple commands that are then executed as a single script in a command console; you have access to any environment variables available in a standard command console. Example: @echo on IF NOT EXIST "%1%" GOTO DEPLOY_NOWAR IF "%APACHE_HOME%" == "" GOTO DEPLOY_NOAPACHE xcopy /L "%1%" "%APACHE_HOME%\webapps"

	GOTO DEPLOY_END rem rem NO WAR FILE rem :DEPLOY_NOWAR echo "%1% WAR file not found" GOTO DEPLOY_END rem rem NO APACHE ENVIRONMENT VARIABLE rem :DEPLOY_NOAPACHE echo "APACHE_HOME environment variable not found" :DEPLOY_END pause
Parse Output	Selecting a Parser from the list causes output of the deploy script to be captured; the output is parsed according to the syntax selected from the list. To display the **System Output** window, select the 'Start > Desktop > Design > System Output' ribbon option.

Recording Scripts

The beauty of recording is not really that we always get to see the bigger picture, but a chance to see a smaller picture that has some truth to tell. We have all seen Sequence diagrams that are less than helpful. (*The same message appearing 100 times in succession on a diagram does tell us something, but not much.*) Fortunately Enterprise Architect takes care of this first point through the use of fragments. Repeating behaviors are identified as Patterns and represented once as a fragment on the Sequence diagram. The fragment is labeled according to the number of iterations. The recording history, of course, always shows the entire history. We also need tools to help us focus the recording on particular areas of interest and reduce the noise from others. We can use filters to do this. With filters, you can exclude any Classes, functions, or even modules from any recording. You can create multiple sets of filters and use them with marker sets to target different Use Cases.

Access

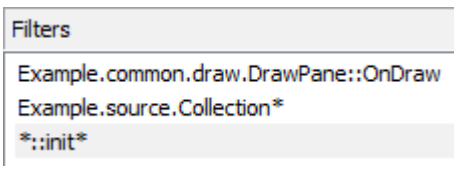
On the **Execution Analyzer window**, either:

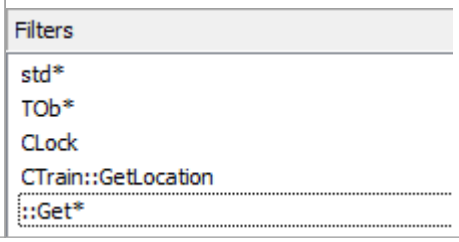
- Locate and double-click on the required script and select the 'Recording' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Recording' page

Ribbon	Develop > Preferences > Analyzer > Edit Analyzer Scripts Execute > Tools > Analyzer
Keyboard Shortcuts	**Shift+F12**

Filter Strings

Element	Discussion
Filtering	If the 'Enable **Filter**' checkbox is selected on the 'Recording' page of the **Execution Analyzer Script Editor**, the debugger excludes calls to matching methods from the recording. The comparison is case-sensitive. To add a value, click on the 'New' ('Insert') icon in the right corner of the 'Exclusion 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 that have a name matching the token; the string can contain the wildcard character * (asterisk). The method_name_token excludes calls to methods having a name that matches the token; again, the string can contain the wildcard character *. Both tokens are optional; if no Class token is present, the filter is applied only to global or public functions (that is, methods not belonging to any Class).
Example	In this Java example, the debugger would exclude: • Calls to the OnDraw method for the 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 (such as <clint> and <init>)  In this Native Code example, 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 the method GetLocation for Class CTrain • Calls to any Global or Public Function with a name beginning with Get 

Filters

Use Filter Entry	To Filter
::Get*	All public functions having a name beginning with 'Get' from the recording session (for example, GetClientRect in Windows API).
::Get	All methods beginning with 'Get' in any Class.

CClass::Get*	All methods beginning with Get for the CClass Class.
CClass::*	All methods for CClass Class.
ATL* std*	All methods for Classes belonging to Standard Template and Active Template Libraries.
CClass::GetName	The specific method(s) GetName for the CClass Class.

Build Application

This topic explains how to execute a Build script on your application, within Enterprise Architect.

Access

Ribbon	Execute > Source > Build > Build
Keyboard Shortcuts	**Ctrl+Shift+F12**
Other	'Build' toolbar > **Execution Analyzer window** |

Action

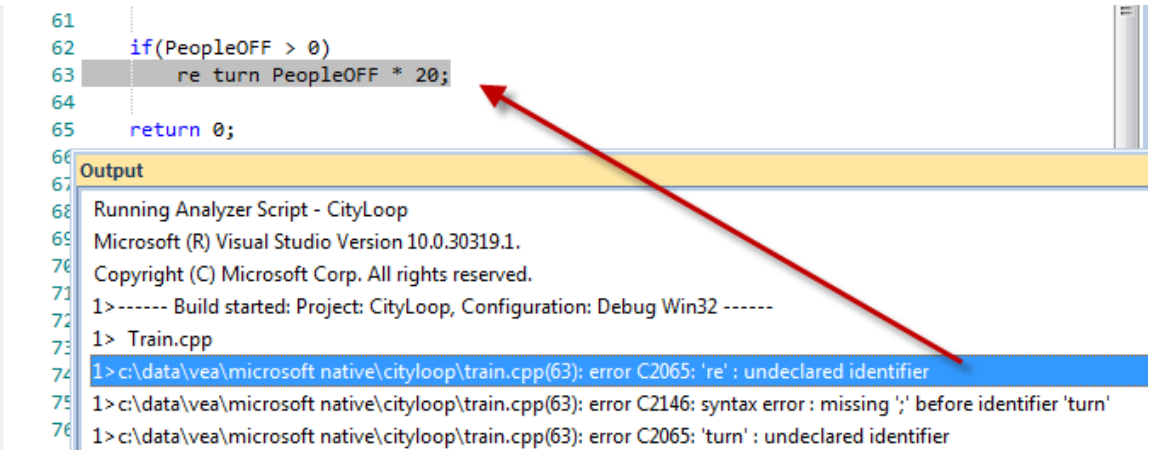
When you select the 'Build' option, it executes the 'Build' command in the script selected in the **Execution Analyzer window**. The progress and outcome of the build operation are displayed in the 'Build' tab of the **System Output** window.

You can quickly visit the line of code for any compilation

error appearing by double-clicking the error.

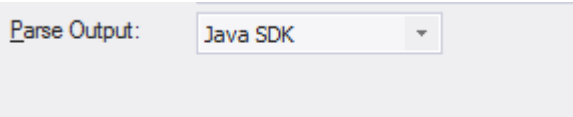
Locate Compiler Errors in Code

When you build an application using an Analyzer Script, compiler output is logged in the **System Output** window. You can double-click on any error message that appears here and be taken to the source code. When you do, the cursor is positioned on the line containing the error.



Tip

If output is missing, check that a language parser is mentioned in the Analyzer Script (**Shift+F12**).

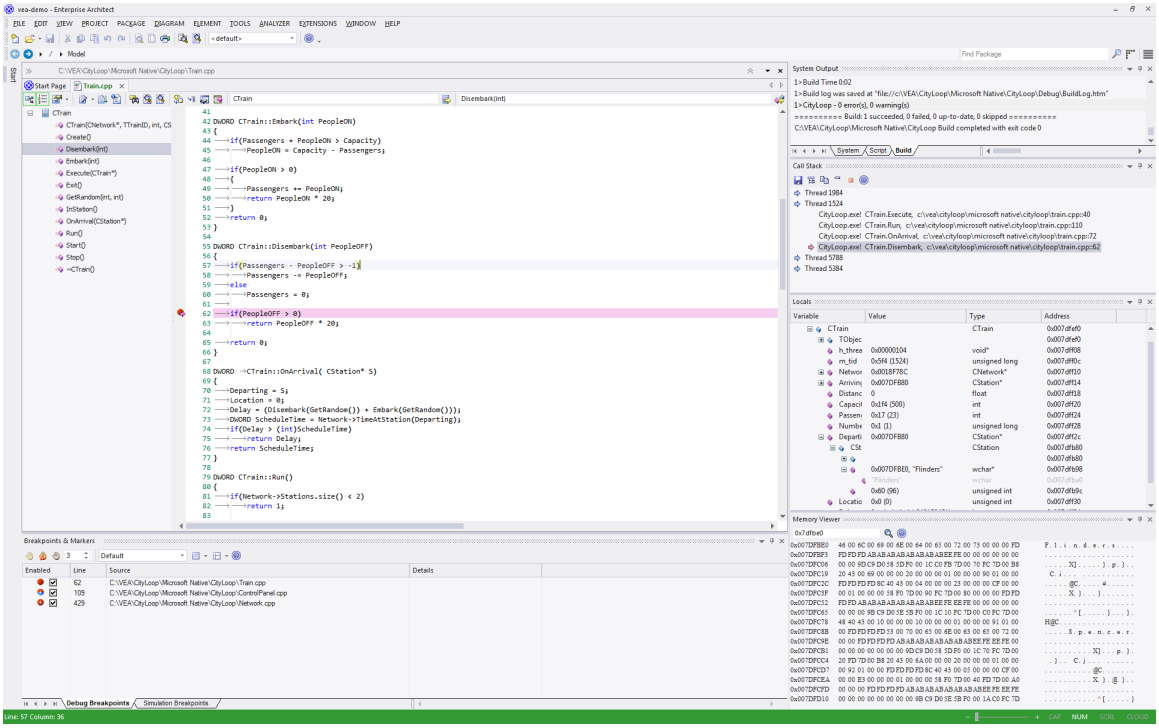


Access

	Start > Desktop > Design > **System**

Ribbon	Output
Keyboard Shortcuts	**Ctrl+Shift+8**

Debugging



Enterprise Architect is more than a drawing tool - it also provides every feature that you might expect in an IDE. Comprehensive debugging environments and tools for many major platforms are made available. Integrating the debugging capability within the modeling tool allows code to be developed, built and managed by its authors. Working and collaborating in an integrated model has made actions count and every action accountable in ways that are just not possible using other tool chains.

Features

Speed

Debuggers in Enterprise Architect are quick! Stepping

through programs will not take all day.

The Recording program can be executed without manual stepping.

Support

- C++, C and Visual Basic
- Microsoft .NET, ASP.NET WCF
- Java, using socket transport (**JDWP**) or in memory model (JVMIT)
- Android on an emulator or device
- JavaScript, VBScript and JScript
- PHP scripts on Apache web servers
- Remote Linux GDB processes using Enterprise Architect on Windows
- Simulation - debug simulations in UML and BPMN
- Executable StateMachines - debug an executing StateMachine

Isolation

The debuggers operate out-of-process from Enterprise Architect, isolating it from side effects.

Efficiency

Starting and stopping the debugger is quick and painless. It does not hold you back. Designed to be a responsive UI, the main UI thread is isolated from duties that are not its responsibility.

Productivity

Switch from modeling to requirements, from raising a

change request to tracking code changes in a model shared across an organization, to profiling recent code changes. All in the one tool.

Notes

- The debug and record features of the **Visual Execution Analyzer** are not available for the Java server platform 'Weblogic' from Oracle

Run the Debugger

Enterprise Architect provides a number of ways to start and control a debug session. There is the main **Debug window**, as well as a Debug toolbar and the 'Run' panel in the 'Execute' ribbon. It is always best to display the Debug window whenever you are running a debug session, as this is where all debug output is captured.

Access

Ribbon	Execute > Run > Start Execute > Tools > **Debugger** > Start Debugging
Keyboard Shortcuts	**Alt+8** (displays the **Debug window**) **F6** (begins execution of the application being debugged)
Toolbar	Explore > Portals > Show Toolbar > Debug

Using the Debug window

Action	Detail
Start the Debugger	When an Analyzer script has been configured to support debugging, you can start the debugger in these ways: • From the ribbon, select 'Execute > Run > Start > Run' • From the ribbon, select 'Execute > Tools > **Debugger** > Start Debugging' • On the 'Debug' toolbar, click on the button, or • Press **F6** You can also launch the debugger for any script through its context menu in the 'Analyzer Script Window', or press **Shift+F12** If you have no Analyzer Script, it is still possible to debug a running application by attaching to that process directly: • From the ribbon, select 'Execute > Tools > Debugger > Attach to Process', or • On the 'Debug' toolbar, click on the (Attach) button and choose the debugging platform manually

Pause/Resume Debugging	You can pause a debugging session, or resume the session after pausing, in these ways: • From the ribbon, select 'Execute > Run > Pause' • On the 'Debug' toolbar, click on the button
Stop the Debugger	To stop debugging, either: • From the ribbon, select 'Execute > Run > Stop' • On the 'Debug' toolbar, click on the (Stop) button • Press **Ctrl+Alt+F6** The debugger normally ends when the current debug process terminates; however, some applications and services (such as Java Virtual Machine) might require the debugger to be manually stopped.
Step Over Lines of Code	To step over the next line of code: • From the ribbon, select 'Execute > Run > Step Over', or • On the 'Debug' toolbar, click on the (Step Over) button, or

	• Press **Alt+F6**
Step Into Function Calls	To step into a function call: • From the ribbon, select 'Execute > Run > Step In', or • On the 'Debug' toolbar, click on the (Step In) button, or • Press **Shift+F6** If no source is available for the target function then the **Debugger** returns immediately to the caller.
Step Out Of Functions	To step out of a function: • From the ribbon, select 'Execute > Run > Step Out' • On the 'Debug' toolbar, click on the (Step Out) button, or • Press **Ctrl+F6** If the debugger steps out into a function with no source code, it will continue to step out until a point is found that has source code.
Show Execution Point	While the **Debugger** is paused, to return to the source file and line of code that the Debugger is about to execute: • From the ribbon, select 'Execute > Run

	> Start > Show Execution Point' • On the 'Debug' toolbar, click on the (Show Execution Point) button. The appropriate line is highlighted, with a pink arrow in the left margin of the screen.
Output	During a debug session, messages display in the **Debug window** detailing: • Startup of session • Termination of session • Exceptions • Errors • Trace messages, such as those output using Java System.out or .NET System.Diagnostics.Debug If you double-click on a debug message, either: • A pop-up displays with more complete message text, or • If there has been a memory leak, the file is displayed at the point at which the error occurred
Save Output (and Clear Output)	You can save the entire contents of the Debug output to an external .txt file, or you can save selected lines from the output to the Enterprise Architect

	clipboard. To save all of the output to file, click on the (Save output to file) button. To save selected lines to the clipboard, right-click on the selection and select the 'Copy Selected to Clipboard' option. When you have saved the output or otherwise do not want to display it any more, right-click on the current output and select the 'Clear Results' option.
Switch to Profiler	If you are running a debug session on code, you can stop the debug session and immediately switch to a **Profiling** session. To switch from the **Debugger** to the **Profiler**: • From the ribbon, select 'Execute > Tools > Debugger > Switch to Profiler' • On the **Debug window**, click on the ' | Switch to Profiler' option, or • On the Debug toolbar, click on the ' | Switch to Profiler' option The Profiler attaches to the currently-running process. This facility is not available for the Java debuggers.

Breakpoint and Marker Management

Breakpoints work in Enterprise Architect in the same way as in any other **Debugger**. Markers are similar to breakpoints, but in Enterprise Architect they have special powers. Simply put, markers perform actions - such as recording execution and analysis - that breakpoints do not. The action of a breakpoint is always to stop the program.

You set any marker or breakpoint in the Source Code editor, where they are visible in the left margin. Clicking in this margin will add a breakpoint at that line. Breakpoints and markers are interchangeable - you can change a breakpoint into a marker and vice versa, using its 'Properties' dialog. You can quickly view and edit a breakpoint or marker's properties using **Ctrl+click** either on its icon in the editor margin or in the **Breakpoints and Markers window**.

Breakpoints are maintained in sets. There is a default set for each model and each breakpoint typically resides there, but you can save the current breakpoint configuration as a named set, create a new set and switch between them. Breakpoint sets are shared; that is, they are available to the model community. The exception is the Default set which is a private and personal set allocated to each user of any model.

Access

Ribbon	Execute > Windows > Breakpoints Simulate > Dynamic Simulation > Breakpoints

Breakpoint and Marker Options

Option	Detail
Delete a breakpoint or marker	To delete a specific breakpoint: • If the breakpoint is enabled, click on the red breakpoint circle in the left margin of the **Source Code Editor**, or • Right-click on the breakpoint or marker in the Source Code Editor, the *Breakpoints* folder or the Breakpoints & Markers window and select the 'Delete' option, or • Select the breakpoint in the 'Debug Breakpoints' tab and press the **Delete key**
Delete all breakpoints	Click on the **Delete all breakpoints button** ( ).

Breakpoint properties	In the **Breakpoints window** or code editor, use the marker's context menu to bring up the properties. Here you can change the marker type, add or modify constraints and enter trace statements. (Useful shortcut: hold the **Ctrl key** while clicking the marker, to quickly show its properties.)
Disable a breakpoint	Deselect the checkbox against the breakpoint or marker.
Enable a breakpoint or marker	Select the checkbox against the breakpoint or marker.
Disable all breakpoints	Click on the button
Enable all breakpoints	Click on the Enable all breakpoints button ( ).
Break when memory address is modified	Click on the Data breakpoint button ( ).
Identify or change the	Check the field in the **Breakpoints & Events window** toolbar.

marker set	If necessary, click on the drop down arrow and select a different marker set. The Default set is normally used for debugging and is personal to your user ID; other marker sets are shared between all users within the model.
Change how breakpoints and markers are grouped on the Breakpoints & Events window	The breakpoints and markers can be grouped by Class or by code file. To group the items, click on the down arrow on the icon in the toolbar, and click on the appropriate option. If you do not want to group the items, click on the selected option to deselect it; the breakpoints and markers are then listed by line number.

Breakpoint States

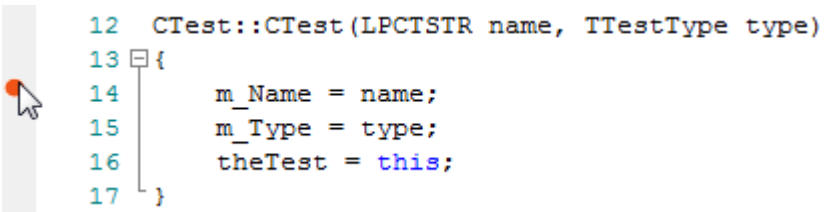
State	Remarks
	Debug Running: Bound *Debug Not Running:* Enabled
	Debug Running: Disabled

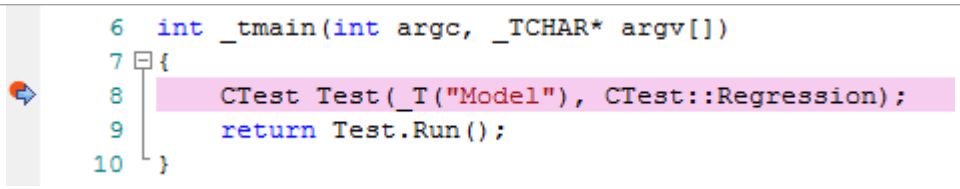
	Debug Not Running: Disabled
	Debug Running: Not bound - this usually means that a module is yet to be loaded. Also, dlls are unloaded from time to time. *Debug Not Running:* N/a
	Debug Running: Failed - this means the debugger was unable to a match this line of code to an instruction in any of the loaded modules. Perhaps the source is from another project or the project configuration is out of date. Note, that if the module date is earlier than the breakpoint's source code date you will see a notification in the debugger window. The text is red in color so they will stand out. This is clear sign that the project requires building. *Debug Not Running:* N/a

Setting Code Breakpoints

Normal Breakpoints are typically set on a line of source code. When the **Debugger** hits the indicated line during normal execution, the Debugger halts execution and displays the local variables, call stack, threads and other run-time information.

Set a breakpoint on a line of code

Step	Action
1	Open the source code to debug in the integrated source code editor.
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.



Alternatively, you can set the Breakpoint marker (or other marker) by right-clicking on the left margin on the required line, to display the breakpoint/marker context menu; select the appropriate marker type.

Trace Statements

A Trace Statement is a message that is output during execution of a debug session. Trace statements can be defined in Enterprise Architect without requiring any changes to your application source code.

Tracepoint Markers are set in the code editor. Like breakpoints, they are placed on a line of code. When that line of code executes, the debugger evaluates the statement, the result of which is logged to the **Debug window** (or to file if overridden by the Analyzer script).

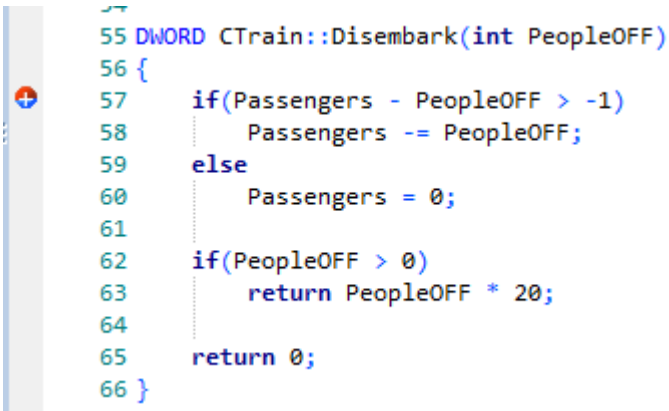
Access

Any existing Trace statements can be viewed and managed in the Breakpoints & Markers window. The Breakpoints & Markers window can be displayed using either of the methods outlined here.

Ribbon	Execute > Windows > Breakpoints

Add a Tracepoint Marker

Ste	Action

p	
1	Open the source code to debug in the source code editor.
2	Find the appropriate code line, right-click in the left margin and select the 'Add Tracepoint Marker' option. If a marker is already there, press **Ctrl+click** to show the **Breakpoint Properties window**.
3	Ensure the 'Trace statement' checkbox is selected.
4	In the text field under the 'Trace statement' checkbox, type the required Trace statement.
5	Click on the **OK button**. A **Tracepoint** Marker is shown in the left margin of the code editor. 

```
55 DWORD CTrain::Disembark(int PeopleOFF)
56 {
57     if(Passengers - PeopleOFF > -1)
58         Passengers -= PeopleOFF;
59     else
60         Passengers = 0;
61
62     if(PeopleOFF > 0)
63         return PeopleOFF * 20;
64
65     return 0;
66 }
```

Specifying a Trace Statement

A trace statement can be any freeform text. The value of any variables currently in scope can also be included in a trace statement by prefixing the variable name with a special token.

The available tokens are:

- $ - when the variable is to be interpreted as a string
- @ - when the variable is a primitive type (int, double, char)

Using our example in the image, we could output the number of people getting off a train by using this statement:

There were @Passengers before @PeopleOFF got off the train at $Arriving.Name Station

In addition to tracing the values of variables from your code, you can use the $stack and $frame keywords in your Trace statement to print the current stack trace; use:

- $stack - to print all frames, or
- $frame[start](count) - print a specific number of frames from the stack starting at a given frame; for example, $frame[0](5) will print the current frame and 4 ancestors

Notes

- Trace statements can be included on any type of breakpoint or marker.

Break When a Variable Changes Value

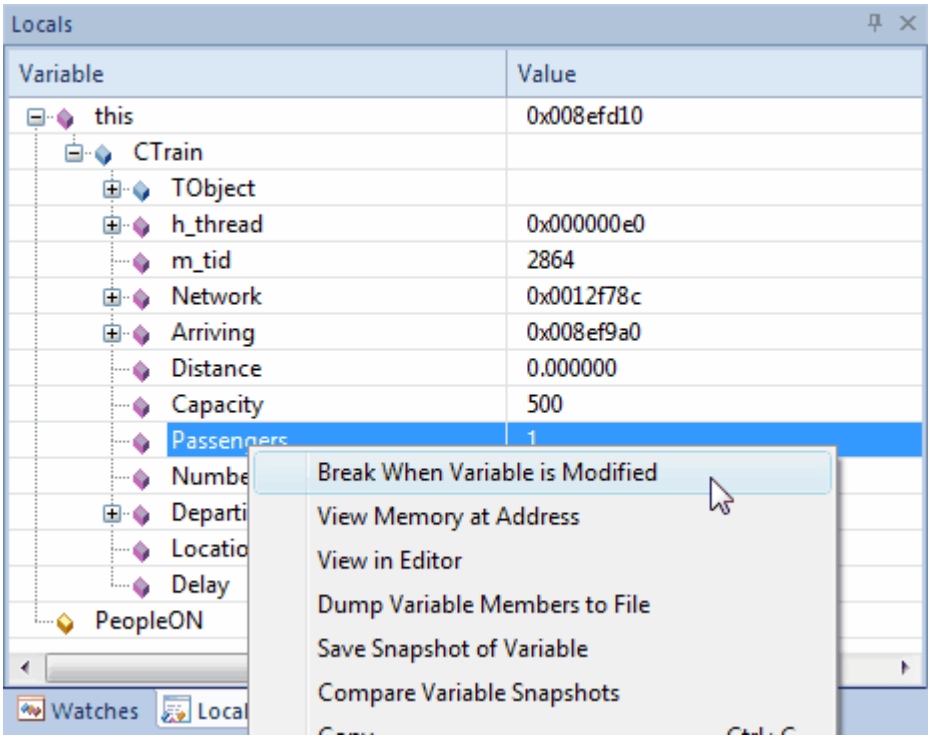
Data breakpoints can be set on a pre-determined memory variable to cause the debugger to halt execution at the line of code that has just caused the value of the variable to change. This can be useful when trying to track down the point at which a variable is modified during program execution, especially if it is not clear how program execution is affecting a particular object state.

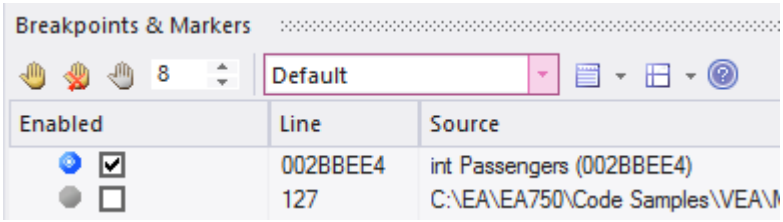
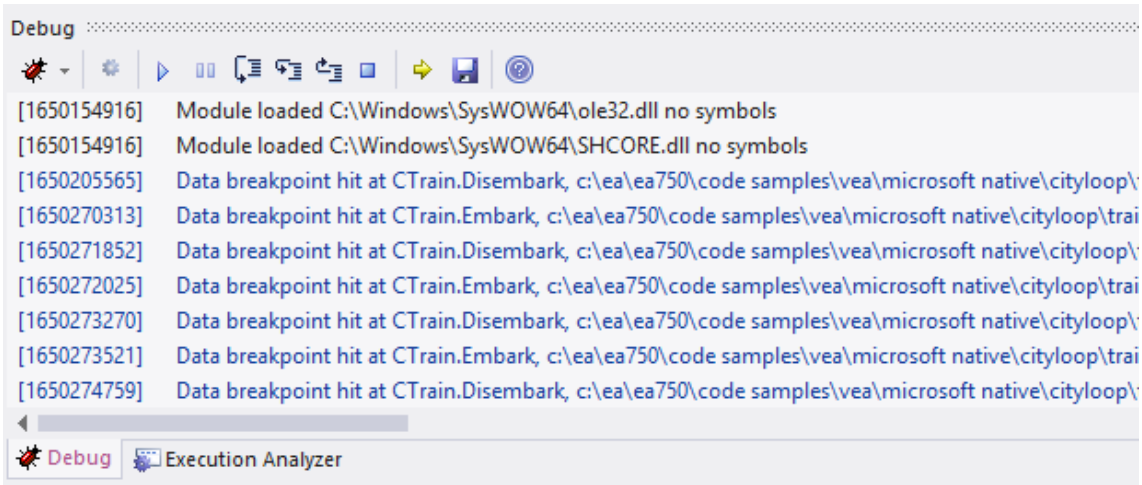
Access

Ribbon	Execute > Windows > Local Variables : Right-click on variable > Break When Variable is Modified or Execute > Windows > Watches : Right-click on variable > Break When Variable is Modified
Other	In a code editor window: Right-click on the variable of interest | Break when item modified

Capture changes to a variable using data

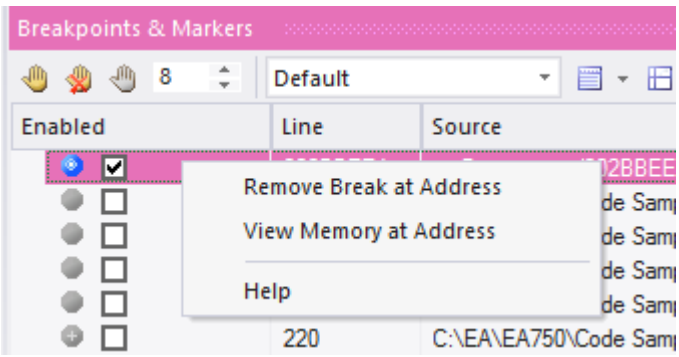
breakpoints

Steps	Detail
1	Set a normal breakpoint in the code so you can choose a variable. Then run the debugger (**F6**).
2	When the program has hit the breakpoint, select the variable of interest and from its context menu, select the 'Break When Variable is Modified' option. 
3	There are no breakpoint indicators in the code, but data breakpoints are easily recognizable in the Breakpoints & **Events window**, being a blue icon with a white diamond. Enterprise Architect displays the name of the variable and its address instead of a

	line number. 
4	With the data breakpoint set, you can disable any other breakpoints you might have. The program will stop at any line of code that changes this variable's value. Now run your program.
5	When this variable is modified, the debugger halts and displays the current line of code in the editor. This is not the line that caused the break, but the line of code following the event. The event is logged to the **Debug window**.  Now we know how and where this value (its State) has changed. For example, the statement at line 58 has just updated the number of Passengers.

```
55 DWORD CTrain::Disembark(int PeopleOFF)
56 {
57     if(Passengers - PeopleOFF > -1)
58         Passengers -= PeopleOFF;
59     else
60         Passengers = 0;
61
62     if(PeopleOFF > 0)
63         return PeopleOFF * 20;
64
65     return 0;
66 }
```

6 Having discovered this and other places where this value is being changed, be sure to get rid of the notification before moving on. You can delete the data breakpoint quickly by selecting it in the **Breakpoints window** and pressing the **Delete key**. You can also use the right-click context menu to do this.



Notes

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

Trace When Variable Changes Value

When your code executes, it might change the value of a variable. It is possible to capture such changes and the variable's new value, on the **Debug window**. You can then double-click on the change record to display the line of code that caused the change, in the Code Editor.

Access

Ribbon	Execute > Windows > Local Variables : Right-click on variable > Trace When Variable is Modified or Execute > Windows > Watches : Right-click on variable > Trace When Variable is Modified
Other	In Code Editor | Right-click on variable | Trace When Variable Modified

Set up Trace

The variable you are tracing must be in scope, so to identify and select it, set a normal breakpoint on the line of code

where you know that the variable will exist. When the debugger reaches this breakpoint, locate the variable and use its context menu to enable the trace.

To locate a variable:

- If you see the variable in the source code, hover over it, right-click and select the 'Display variable' option; Enterprise Architect will locate it
- If the variable is in scope (a local, or 'this' or a member of 'this'), look for it in the Locals Window ('Execute > Windows > Local Variables')
- If the variable is global (C, C++), display the **Watches window** ('Execute > Windows > Watches') and search for it by name
- If the variable is a Class static member, display the Watches window ('Execute > Windows > Watches') and enter its fully qualified name

Once trace is enabled, you can disable all other breakpoints and let the program run. Each time the variable changes value, it will be logged to the 'Output' tab of the debugger. Check the change in value and double-click on the line to display the code in the Code Editor.

Notes

- The debugger does not halt when the change event occurs, it only logs the change
- This facility is available on the Microsoft Native and Java

platforms

- Microsoft .NET does not support breakpoints on values

Detecting Memory Address Operations

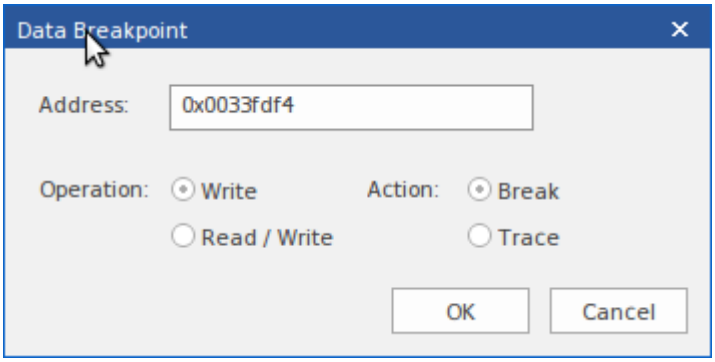
Being able to detect where and when an area of memory is being read or written can be a great help for investigators, even when the code base is well understood. Without this tool, a C++ developer could have a potentially daunting task of tracking where and when a global variable is accessed, and debugging those functions. Data breakpoints allow a C++ programmer to track when a variable / memory location is read or when it is written. When the operation is detected, the debugger will halt the execution and the line of code following the operation will be displayed in the code editor.

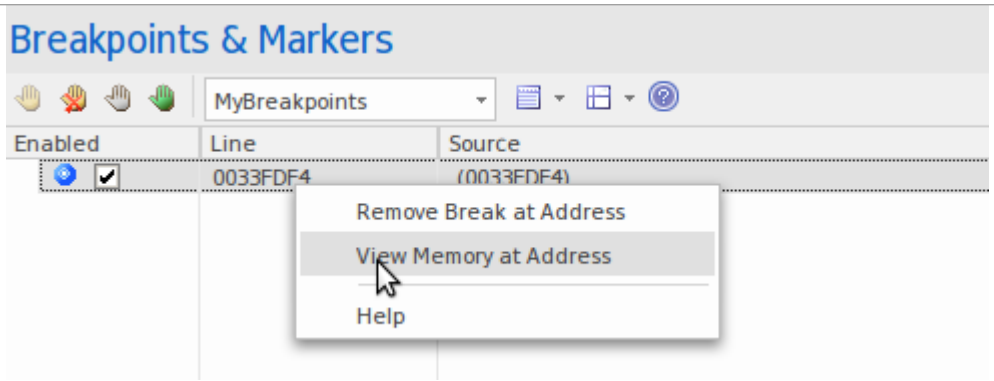
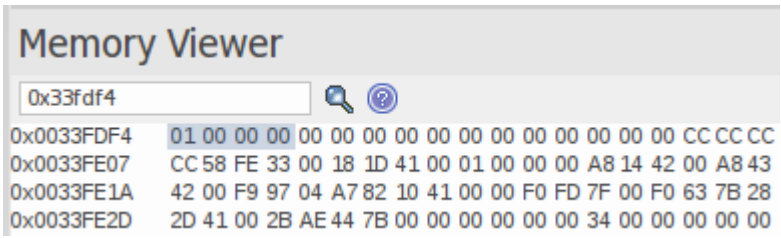
Access

Ribbon	Execute > Windows > Breakpoints

Detect operation on memory address

Step	Action

1	Click the button.
2	Enter the memory address to watch. You can copy an address from the Locals (Local Variables) window. 
3	Select the operation to detect. If you select 'Write', the debugger will break when the address is written to. If you choose 'Read / Write', the debugger will notify you when the address is read or when it is written.
4	Select the action to perform. If you choose 'Break', the debugger will halt the program and the line of code will be shown in the editor. If you choose 'Trace', the debugger will not halt execution, but log any operation on the address as it occurs. This output is displayed in the **Debugger** Window.
5	The data breakpoint is added to the **Breakpoints and Markers window**.

	
6	You can use the context menu on the data breakpoint to check the value at the memory address. 
7	To delete a data breakpoint, select it in the **Breakpoints and Markers window** and press the **Delete key**. Alternatively, deselect the checkbox next to it. Data breakpoints are deleted when they are disabled; they do not persist as other breakpoints do.

System Requirements

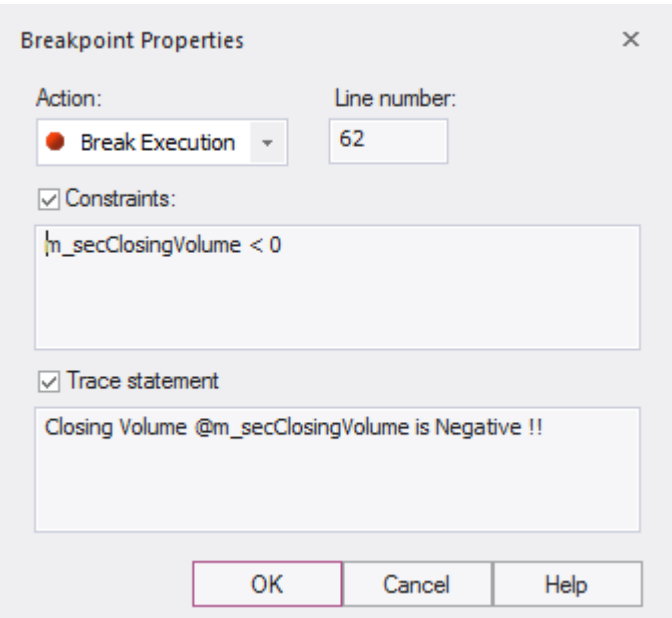
Memory address breakpoints are supported in the C/C++ native debugger.

Breakpoint Properties

Breakpoints have a number of additional properties that determine what occurs when executing the line of code that the breakpoint applies to.

These properties define:

- The action to be performed
- The line of code that the breakpoint applies to
- Constraints that determine whether or not the action is performed when the breakpoint is hit
- Trace information to be output when the breakpoint is hit



Access

There are several ways to display the 'Breakpoint Properties'

dialog:

Code Editor	• Right-click on a breakpoint marker | Properties or • Ctrl+Click on breakpoint marker or • Right-click on code that has a breakpoint marker | Breakpoint | Properties
Breakpoints & Markers window	• Right-click on breakpoint | Properties

Options

Field	Details
Action	The behavior when the breakpoint is hit.
Line	The line of source code that this breakpoint applies to.
Stack Height	For Stack Capture markers, the number of caller frames to record. To record the entire Stack, set the value to 0.

Constraints	Defines the condition under which the breakpoint action will be taken. For normal breakpoints this would be the condition that halts execution. In this example, for a normal breakpoint, execution would stop at this line when the condition evaluates to **True**. Constraints are evaluated each time the line of code is executed. (this.m_FirstName="Joe") AND (this.m_LastName="Smith")
Trace statement	A message output to the **Debug window** when the breakpoint is hit. Variables currently in scope can be included in a trace statement output by prefixing the variable name with a $ token for string variables, or an @ token for primitive types such as int or long. For example: Account $pAccount->m_sName has a balance of @pAccount->m_fBalance

Failure to Bind Breakpoint

A breakpoint failure occurs if there is a problem in binding the breakpoint. Breakpoint failures are most often caused by source files being changed without the application being rebuilt. Breakpoints can sometimes bind to a different line, causing them to be moved. If a breakpoint cannot be bound to the binary at this line or the three lines following it, it is displayed with a question mark.

A warning message displays in the 'Details' column of the Breakpoints & **Events window**, identifying the type of problem:

- The source file for the breakpoint does not match the source file used to build the application image
- The time date stamp on the file is greater than that of the image

A warning message is also output to the **Debug window**.

Debug a Running Application

Rather than starting a process explicitly from within Enterprise Architect, you might want to debug an application (process) that is already running on your system.

In this case you can use the debugging capability to attach to the process that is already running. Provided you have the appropriate debug information written into the running process, and/or associated debug files (such as .PDB files), the debugger binds to that process and initiates a debug session.

You can also 'detach' from the process after you have completed your inspection and leave the process to run as normal.

Access

Ribbon	Execute > Run > Start > Attach to Process
Debugger Window	The debugger window toolbar has an **Attach button**

Stages

Stage	Description
Show Processes	When you select to debug another process, the 'Attach To Process' dialog displays. 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.
Select Debugger	When you select a process, you might have to choose the debugger from the **Debugger** dropdown list; however, if the selected Package has already been configured in an Analyzer Script, then the debugger listed in the script is preset on the dialog.
Process Selection	Once you double-click on a process containing debug information, and Enterprise Architect is attached to the process: • Any breakpoints encountered are detected by the debugger • The process is halted when a breakpoint is encountered, and

	• The information is available in the **Debug window**
Detach From Process	To detach from a process, click on the (Debug Stop) button.

View the Local Variables

The **Locals window** displays variables of the executing system. Whether you are recording C#, debugging Java, C++ or VBScript, debugging an Executable StateMachine, or running a simulation, this window is where the system's variables are located. Current values are only displayed when a program is halted. This occurs when a breakpoint is encountered during debugging, when you step over a line of code or when you step between States in a simulation.

Access

Ribbon	Execute > Windows > Local Variables Simulate > Dynamic Simulation > Local Variables
Context Menu	In Code Editor | Right-click on any variable identifier > Display Variable

Icons

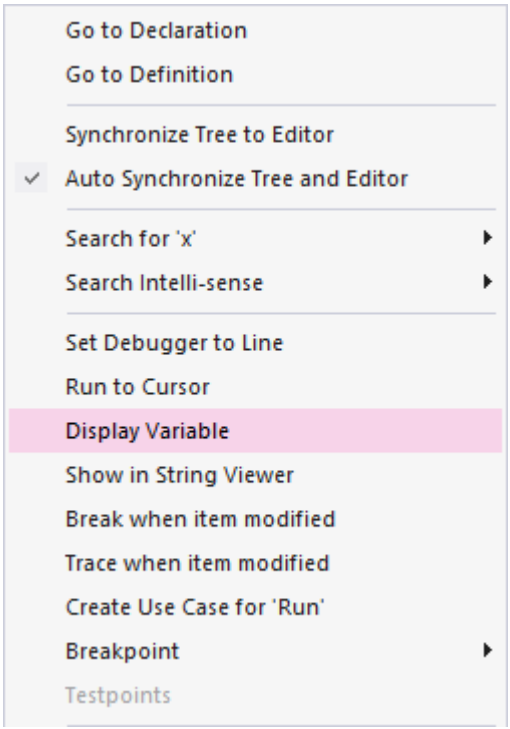
The value and type of any in-scope variable is displayed in a tree; each variable has a colored box icon that identifies the

type of variable:

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

Finding variables

The easiest way to find a variable is to first locate it in the code editor and use the right-click context menu on the variable, selecting 'Display Variable'. Enterprise Architect will find and reveal any variable in scope, including deeply nested members. If the variable is found in a different scope (global, file, module, static), it will be displayed in the **Watches window** (see *View Variables in Other Scopes*).

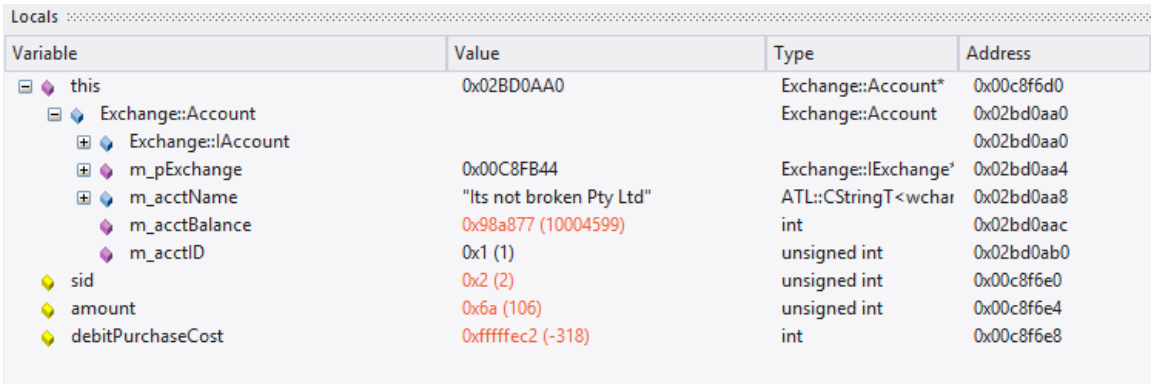


Persistent View

The examination of variables usually involves digging around in the tree to expose the values of interest. It can be annoying then, having gone through that trouble, to step to the next line of code, only to have those variables buried from sight again due to a change in context. The **Locals window** has a persistent view that lingers for a while after a run or step command. When you step through a function in Enterprise Architect, the variables structure persists line after line. This makes stepping through a function quick and easy.

What changed

As part of the persistent view, the **Locals window** tracks changes to values and highlights them.



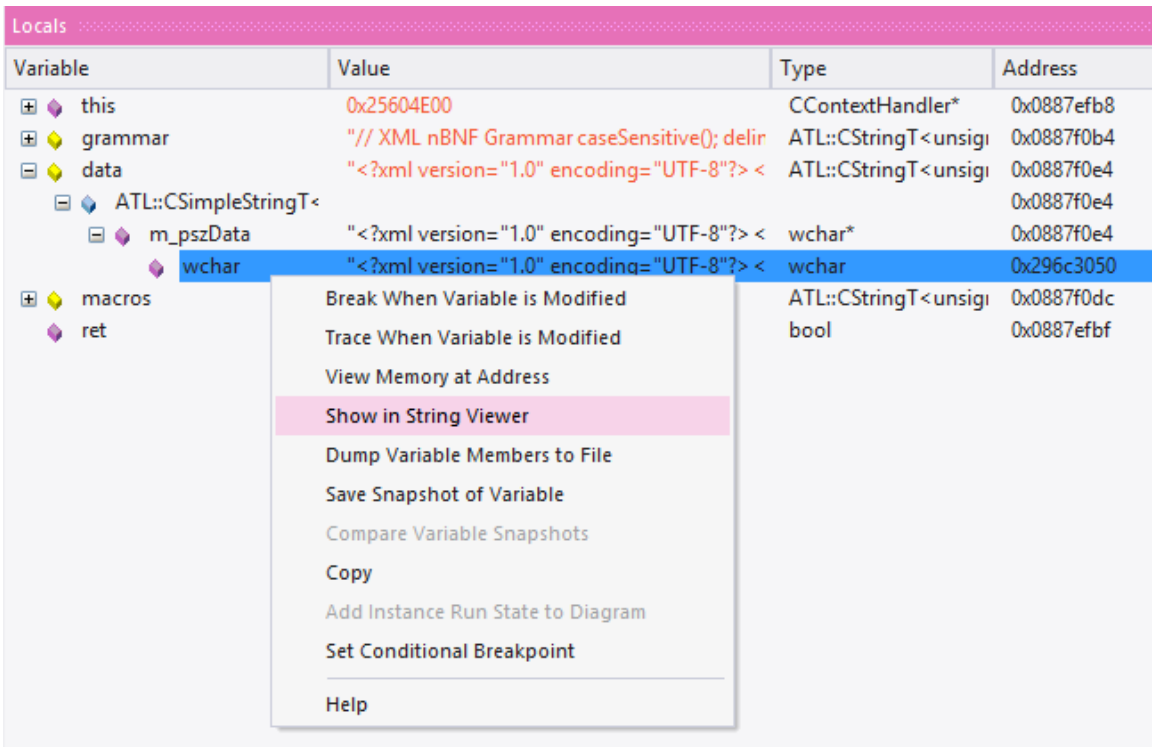
Context Menu

Facility	Detail
Break When Variable is Modified	Set data breakpoints on the selected memory variable to halt debugger execution at the line of code that has just caused the value of the variable to change.
View Memory at Address	Display the raw values in memory at the selected address, in hex and ASCII.
Show in String Viewer	Display the variable string in the 'String Viewer' dialog.

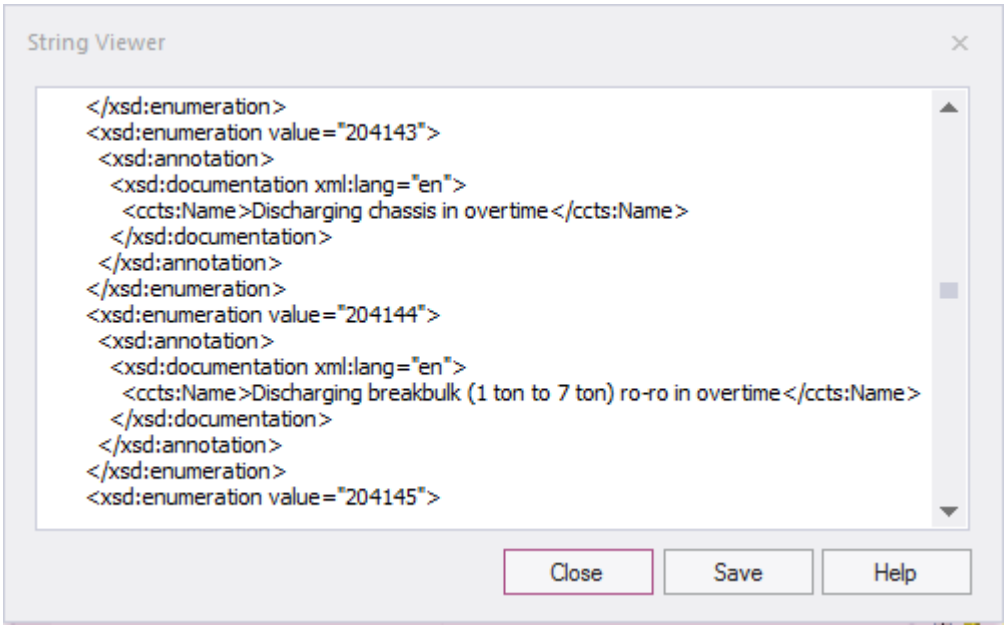
Dump Variable Members to File	Capture and store the selected variables to a separate location; a browser displays to select the appropriate .txt file name and file path.
Save Snapshot of Variable	Capture the value of a variable at a specific point in the life of that variable.
Compare Variable Snapshots	Compare the values of a variable at different points in the life of that variable.
Copy	Copy the selected variable to the Enterprise Architect clipboard.
Add Instance Run State to Diagram	If you have opened a model diagram containing an Object of the Class for which the source code is being debugged, this option updates that Object with the Run State represented by the variable value.
Set Conditional Breakpoint	Add a breakpoint at the current execution position with a constraint for this variable matching its current value.

View Content Of Long Strings

For efficiency, the **Locals window** only shows partial strings. However, you can display the entire contents of a string variable using the 'String Viewer'.



This example shows the value of a variable holding the contents of an XML schema file.

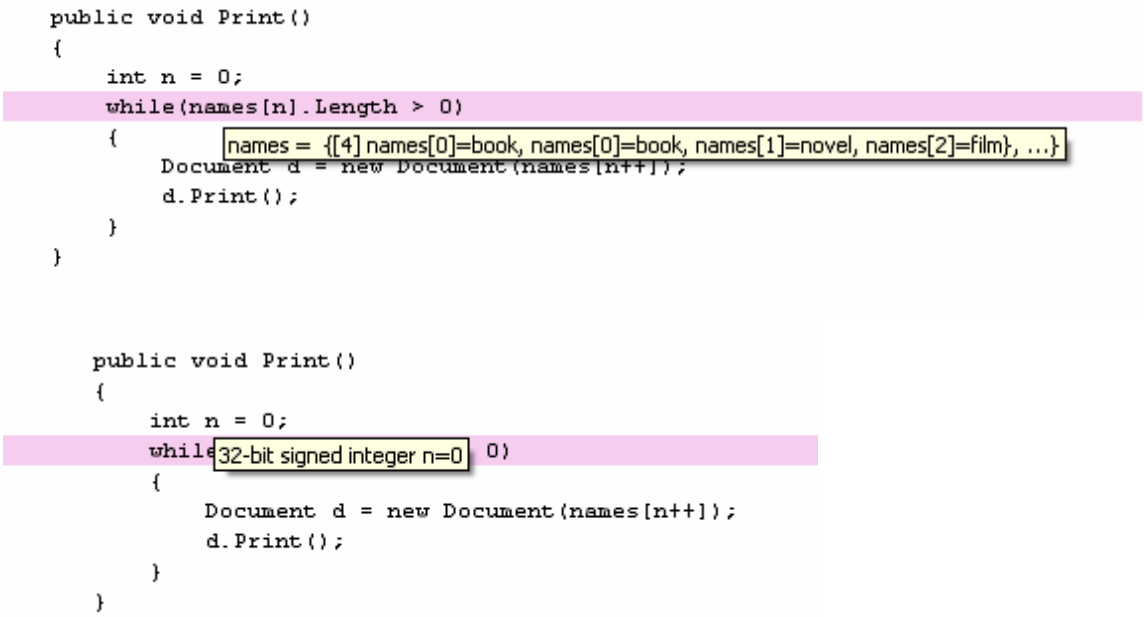


Access

From Code Editor or Locals window	Right-click on string variable | Show in String Viewer

View Debug Variables in Code Editors

When a breakpoint occurs, you will see all the local variables in that window. You can also inspect variables in the **Source Code Editor** by hovering your mouse over the reference. Here are some examples.



Note: The variable does not have to be one of the local variables. It can have a file or module scope.

Variable Snapshots

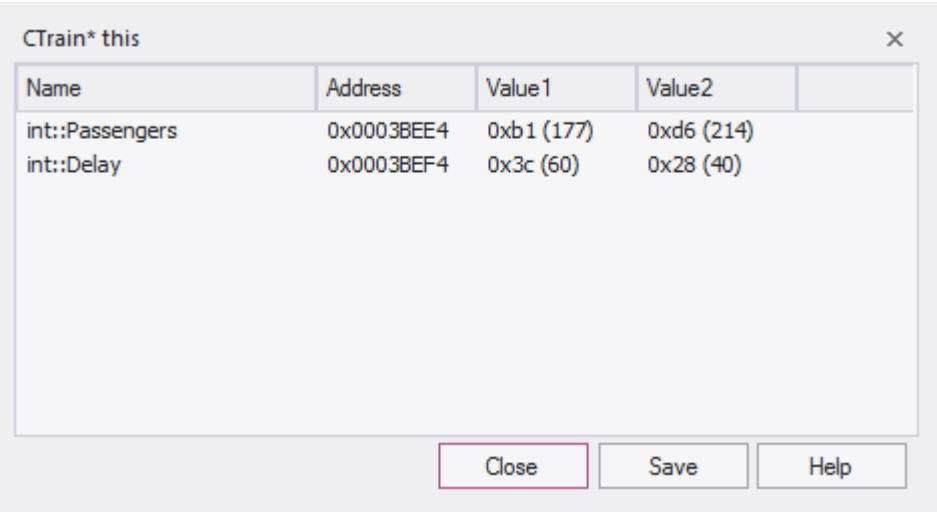
It is possible to take a 'snapshot' of a variable when your program hits a breakpoint and use this snapshot to see how the value of the variable changes at different points in its life. The debugger does not copy the value of the selected variable only; for complex variables it copies the values of the selected variable and of each of its hierarchy of members until it can no longer find any more debug information for a member or no more members can be found.

Capture Variable Snapshot

Step	Action
1	In the Code Editor, set two breakpoints: one at the start of a function and another at the end of the function.
2	At the start breakpoint, right-click on a variable in the **Locals window** and select the 'Save Variable Snapshot' menu option.
3	Run the application.
4	When the end breakpoint is reached, right-click on

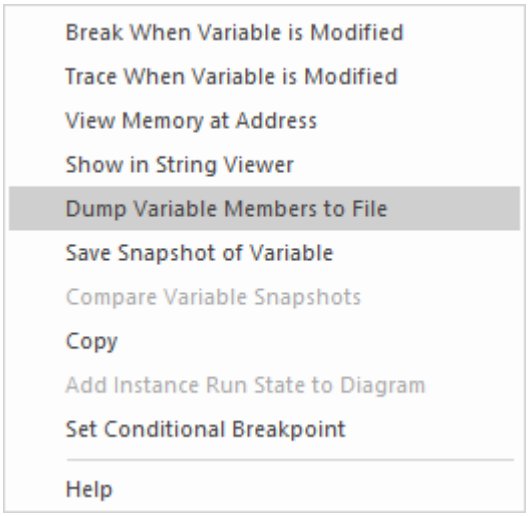
the variable in the **Locals window** and select the 'Compare Variable Snapshots' option.

A dialog displays that shows the original value from the first snapshot and the current value from the second snapshot as illustrated in this diagram taken from the EA.Example model.



Save Variable Snapshot to File

You can save the state of a variable to file using its right-click context menu.



This is an excerpt of the file contents.

```
73 00000006|0x00731F00|name|TObjectType::Type  |value|TypeIsStation|
74 00000005|0x00731F08|name|wchar::Name    |value|"Treasury"|
75 00000005|0x00731F0C|name|unsigned::Location |value|0x40 (64)|
76 00000003|0x0003BED8|name|float::Distance    |value|0|
77 00000003|0x0003BEE0|name|int::Capacity  |value|0x1f4 (500)|
78 00000003|0x0003BEE4|name|int::Passengers    |value|0xd6 (214)|
79 00000003|0x0003BEE8|name|unsigned::Number   |value|0x3 (3)|
80 00000003|0x0003BEF0|name|unsigned::Location |value|0x0 (0)|
81 00000003|0x0003BEF4|name|int::Delay |value|0x28 (40)|
```

Actionpoints

Actionpoints are breakpoints that can perform actions. When a breakpoint is hit, the actionpoint script is invoked by the debugger, and the process continues to run. Actionpoints are sophisticated debugging tools, and provide expert developers with an additional command suite. With them, a developer can alter the behavior of a function, capture the point at which a behavior changes, and modify/detect an object's state. To support these features, Actionpoints can alter the value of primitive local and member variables, can define their own 'user-defined-variables' and alter program execution.

User-Defined Variables in Actionpoints and Breakpoints

User Defined Variables (UDVs):

- Provide the means for setting a **UDV** primitive or string in **Actionpoint** statements
- Can be used in condition statements of multiple markers/breakpoints
- Can be seen easily in the same **Local Variables window**
- The final values of all UDVs are logged when debugging ends.

In the UDV syntax, the UDV name:

- Must be preceded by a # (hash) character
- Is case-insensitive

Actionpoint Statements

Actionpoint statements can contain set commands and goto commands.

set command

Sets variable values. An **Actionpoint** statement can contain multiple 'set' commands, all of which should precede any 'goto' command.

The 'set' command syntax is:

set LHS = RHS

Where:

- **LHS** = the name of the variable as a:
  - user defined variable (**UDV**) such as #myval
  - local or member variable such as strName or this.m_strName
- **RHS** = the value to assign:
  - As a literal or local variable
  - If a literal, as one of: integer, boolean, floating point, char or string

set command - Variable Examples

UDV Examples	Local Variable Examples
set #mychar = 'a'	set this.m_nCount=0
set #mystr = "a string"	set bSuccess=false
set #myint = 10	
set #myfloat = 0.5	
set #mytrue = true	

goto command

goto command - switches execution to a different line

number in a function. An **Actionpoint** statement can contain only one goto command, as the final command in the statement.

The goto command syntax is:

goto L

Where **L** is a line number in the current function.

Integer operators

Where a **UDV** exists and is of type int, it can be incremented and decremented using the ++ and -- operators. For example:

1. Create a UDV and set its value and type to a local integer variable.
   AP1: set #myint = nTotalSoFar
2. Increment the UDV.
   AP2: #myint++
3. Decrement the UDV.
   AP3: #myint--

Timer operations

Actionpoints can report elapsed time between two points. There is only one timer available, which is reset or started with the startTimer command. The current elapsed time can

then be printed with the printTimer command. Finally, the total elapsed time is printed and the timer ended with the endTimer command.

Example Actionpoint Conditions

With Literals and constants:

- (#mychar='a')
- (#mystr <> "")
- (#myint > 10)
- (#myfloat > 0.0)

With Local Variables:

- (#myval == this.m_strValue)
- (#myint <> this->m_nCount)
- (#myint != this->m_nCount)

Instruction Recording

Instruction recording can be useful for detecting changes to a known behavior; the point in execution (B) that diverges from a previous execution(s) (A). The commands are:

- recStart - starts recording or starts comparing if a previous recording exists

- recStop - ends recording
- recPause - pause recording
- recResume - resumes recording

The **recStart** command begins recording instructions. Executed instructions are then stored. When a **recStop** command is encountered, the recording is saved. There can only be one saved recording at any one time between two **Actionpoints**. When a **recStart** is encountered and a previous recording exists, the debugger will begin comparing each subsequent instruction with its recording. It could perform many comparisons. If and when a difference is detected, the debugger will break and the line of code where the behavior changed will be displayed in the code editor. The iteration of the comparison is also printed.

The recording is stored in memory by default, but it can also be stored to a file with the command syntax:

recStart filesspec

For example:

recStart c:\mylogs\onclickbutton.dat

When a **recStart** command is encountered that specifies a file, and that file exists, it is loaded into memory and the debugger will immediately enter comparison mode.

Expressions

There is no implicit precedence in Breakpoint, **Actionpoint**

and **Testpoint** conditional expressions. In complex expressions, the use of parentheses is mandatory. See these examples:

Type	Example
Actionpoint UDV example	(#myint=1) AND (#mystr="Germany")
Local variables examples	(this.m_nCount > 10) OR (nCount%1) (this.m_nCount > 10) OR (bForce)
Equality operators in conditional expressions	<> - Not Equal != - Not Equal == - Equal = - Equal
Assignment operator in Actionpoint	= - Assigns RHS to LHS
Arithmetic operators in conditional expressions	/ - division + - plus - - minus * - multiplication % - modulus

Logical operators in conditional expressions	AND - both must be true OR - one must be true && - both must be true || - one must be true ^ - exclusive OR (only one must be true)

View Variables in Other Scopes

Access

Ribbon	Execute > Windows > Watches
Other	**Execution Analyzer window** toolbar : | Watches

Views

View	Description
Watches	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 - 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 add a watch, type the name of the

	variable to watch in the toolbar, and press the **Enter key**. To examine a static Class member variable in C++, Java or Microsoft .NET, enter its fully qualified name: CMyClass::MyStaticVar To examine a C++ data symbol with module or file scope, just enter its name. Variables are evaluated by looking at the current scope; that is, the module of the current stack frame (you can change the scope at a breakpoint by double-clicking the frame in the **Call Stack**). If the global variable exists in a different module, you can examine the variable by prefixing the module name to the variable modulename!variable_name
History	The history of items entered is maintained. Previously entered names or expressions can be selected again using the Up arrow key and **Down arrow** key inside the toolbar text box. The history will also persist for the user across any instance of Enterprise Architect or model on the same machine.

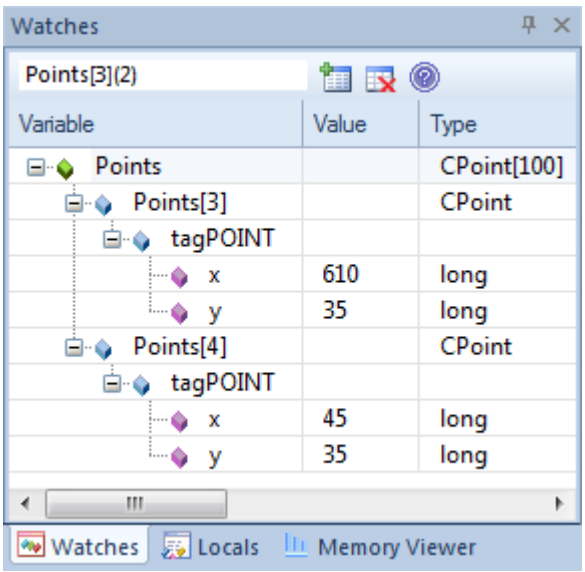
View Elements of Array

You can use the **Watches window** to inspect one or more specific elements of an array.

In the field to the left of the Watches window toolbar, type the variable name of the array followed by the start element and the number of elements to display. The start element is enclosed in square brackets and the count of elements is enclosed in parentheses; that is:

variable[start_element](count_of_elements)

For example, Points[3](2) displays the fourth and fifth elements of the Points array, as illustrated.



If you entered Points[3] the Watches window would show the third array element only.

Access

Ribbon	Execute > Windows > Watches
Other	**Execution Analyzer window** toolbar : | Watches

View the Call Stack

The **Call Stack** window is used to display all currently running threads in a process. It can be used to identify which thread is operational, immediately before program failure occurs.

When a Simulation is active, the Call Stack will show the current execution context for the running simulation. This will include a separate context stack for each concurrent simulation 'thread'.

A stack trace is displayed whenever a thread is suspended, through one of the step actions or through encountering a breakpoint. The Call Stack window can record a history of stack changes, and enables you to generate Sequence diagrams from this history.

Access

Ribbon	Execute > Windows > **Call Stack**
Other	**Execution Analyzer window** toolbar : **Call Stack**

Use to

- View stack history to understand the execution of a process
- View threads
- Save a call stack for later use
- Record call stack changes for Sequence diagram generation
- Generate a Sequence diagram from the call stack
- View the related code line in the **Source Code Editor**

Facilities

Facility	Description
Indicators	• A pink 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
Save a Call Stack to a .TXT File	Not currently available.

Record a Thread in a Debug Session	To record the execution of a thread and direct the recording to the Record & Analyze window, right-click on the thread in the **Call Stack** and select the appropriate context menu option: • 'Record' - to manually record the current thread during the debug session Used in conjunction with the 'step' buttons of the debugger; each function that is called due to a step command is logged to the Record & Analyze window • 'Auto-Record' - to perform auto-recording during a debug session When you select this icon, the Analyzer begins recording and does not stop until either the program ends, you stop the debugger or you click on the 'Stop' icon
Stop Recording	If you have started a manual or automatic recording of a thread you can stop it before completion; select the thread (indicated by a red arrow) and either: • Click on the (Stop Recording) button in the toolbar or • Right-click and select the 'Stop' option
Generate a	To generate Sequence diagram from the

Sequence Diagram from the Call Stack	**Call Stack** trace, either: • Click on the (**Generate Sequence Diagram** of Stack) button, or • Right-click and select the 'Generate Sequence Diagram' option
Copy Stack to Recording History	To add the stack details immediately to the Record & Analyze window (for later generation of Sequence diagrams) either: • Click on the button, or • Right-click and select the 'Copy Stack to Record History' option
Toggle Stack Depth	To toggle between showing the full stack and showing only frames with source, click on the (Toggle Stack Depth) button.
Display Related Code in Source Code Editor	Double-click on a thread/frame to display the related line of code in the **Source Code Editor**; local variables are also refreshed for the selected frame.

Create Sequence Diagram of Call Stack

The **Call Stack** window records a history of stack changes from which you can generate Sequence diagrams.

Access

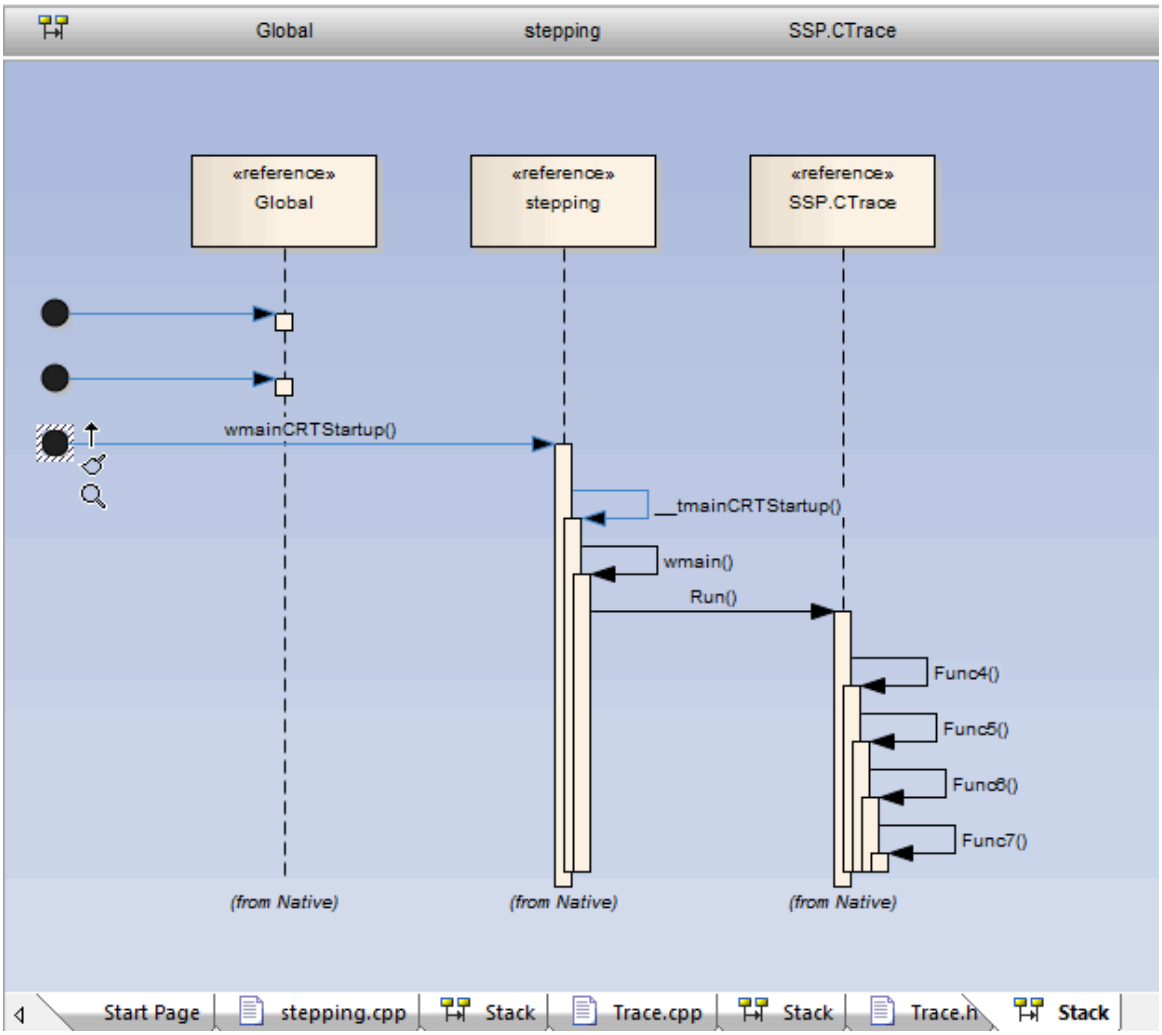
Ribbon	Execute > Windows > **Call Stack**
Other	**Execution Analyzer window** toolbar : **Call Stack**

Use to

- Record **Call Stack** changes for Sequence diagram generation
- Generate a Sequence diagram from the 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**.


Global
stepping
SSP.CTrace
«reference»
Global
«reference»
stepping
«reference»
SSP.CTrace
wmainCRTStartup()
__tmainCRTStartup()
wmain()
Run()
Func4()
Func5()
Func6()
Func7()
(from Native)
(from Native)
(from Native)
Start Page
stepping.cpp
Stack
Trace.cpp
Stack
Trace.h
Stack

Inspect Process Memory

Using the Memory Viewer, you can display the raw values of memory in hex and ASCII. You can manually define the memory address in the 'Address' field (top right), or right-click on a variable in the **Locals window** or **Watches window** and select the 'View Memory at Address' option.

Access

Ribbon	Execute > Windows > Memory Viewer
Other	**Execution Analyzer window** toolbar : | Memory Viewer From **Locals window** or **Watches window** : Right-click on a variable | View Memory at Address

Notes

- The Memory Viewer is available for debugging Microsoft Native Code Applications (C, C++, VB) running on

Windows or within WINE on Linux

Show Loaded Modules

For .NET and native Windows applications, you can list the DLL's loaded by the debugged process, using the **Modules window**. This list can also include associated symbolic files (PDB files) used by the debugger.

Access

Ribbon	Execute > Windows > Modules

Modules Window display

Column	Description
Path	Shows the file path of the loaded module.
Load Address	Shows the base memory address of the loaded module.
Modified Date	Shows the local file date and the time the module was modified.

Debug Symbols	Shows: • 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	Indicates the validity of the symbol file; if the value is false, the symbol file is out of date.
Symbol Path	Shows the file path of the symbol file, which must be present for debugging to work.
Modified Date	Shows the local file date and time the symbol file was created.

Process First Chance Exceptions

Access

Ribbon	Execute > Tools > **Debugger** > Process First Chance Exceptions
Other	**Debug window** toolbar : | Process First Chance Exceptions

Processing Elements

Element	Description
Debug Process	When an application is being debugged and the debugger is notified of an exception, the application is paused and the debugger responds in the way it is configured to do; it either: • Resumes the application and leaves the exception to the application to manage, or • Keeps the application suspended and

	passes the exception to the appropriate routines for automatic resolution or manual intervention
Second Chance Exceptions	The Enterprise Architect debugger defaults to the first listed behavior. If the application can handle the exception, it continues to process; if it cannot handle the exception, the debugger is notified again and this time it must suspend the application and resolve the exception condition. In this behavior, because the debugger has encountered the exception twice, it is known as a second-chance exception; in this case, if the exception does not halt execution, it is ignored and you avoid spending time on conditions that do not impact the overall outcome of processing. You might work this way on large or complex systems that invariably involve exception conditions somewhere in the processing paths.
First Chance Exceptions	However, if you want to examine every exception that occurs as soon as it occurs, you can set the debugger to adopt the second behavior. Because the debugger responds to the

	exception on first contact, it is known as a first-chance exception. You might work this way with individual functions or routines that must work cleanly or not at all.
Selection	Select the 'Process First Chance Exceptions' option to debug exceptions on first contact. Deselect the option to process exceptions only if the application fails when they occur.

Just-in-time Debugger

You can register the Enterprise Architect debugger as the operating system Just-in-time debugger, to be invoked when an application running outside Enterprise Architect on the system either encounters an exception or crashes. When you do so, an application crash will cause Enterprise Architect to be opened, and the source and reason for the crash displayed.

Access

Ribbon	Execute > Tools > **Debugger** > Set as JIT Debugger

Services

Enterprise Architect provides two services to facilitate remote script execution and remote debugging. The services primarily support Enterprise Architect running on Linux to allow users to run native Linux shell scripts and debug Linux programs. The Satellite service supports Analyzer Scripts while the Agent service supports debugging.

Access

Ribbon	Execute > Tools > Services

The Satellite Service

The Satellite service is responsible for executing Analyzer Scripts on the machine on which it is running. The feature can help Linux users to execute native Linux programs and shell commands directly, bypassing Wine. The service can be managed from the ribbon. It can also be run independently from a terminal.

The Linux Shell

The default shell used by Enterprise Architect is 'bash'. To override the Linux Shell used by Enterprise Architect, open a Linux terminal, run 'wine regedit ' and add a string value to this registry key:

HKEY_CURRENT_USER\Software\Sparx Systems\EA400\EA\Options

where:

- key name: "LINUX"
- key value: *path*

and *path* is the Linux path to the shell program "/bin/bash", for example.

Permissions

Under Linux you must check that the service programs have the appropriate permissions. The programs are located under the Enterprise Architect installation folder. The sub directory "**VEA**/x86/linux". Check that each of the programs in this directory has the execute permission set for the owner.

Notes

- The Satellite services are enabled in the Unified and Ultimate Editions of Enterprise Architect

The Agent Service

The Agent service is responsible for managing debugging sessions for Enterprise Architect's GDB debugger. The service allows Enterprise Architect users to debug Linux programs. The service can be managed from the ribbon. It can also be run independently from a terminal.

The Services Menu

Start Satellite Service	Starts the service. The service listens on the Satellite Port configured in any Analyzer Script Services Page.
Stop Satellite Service	Stops the service.
Test Satellite Service	Tests whether the service is running or not.

Start Agent Service	Starts the service. The service listens on the Agent Port configured in an Analyzer Script Services Page.
Stop Agent Service	Stops the service.
Test Agent Service	Tests whether the service is running or not.