Enterprise Architect

User Guide Series

Visual Execution Analysis

Author: Sparx Systems & Stephen Maguire

Date: 16/01/2019

Version: 1.0

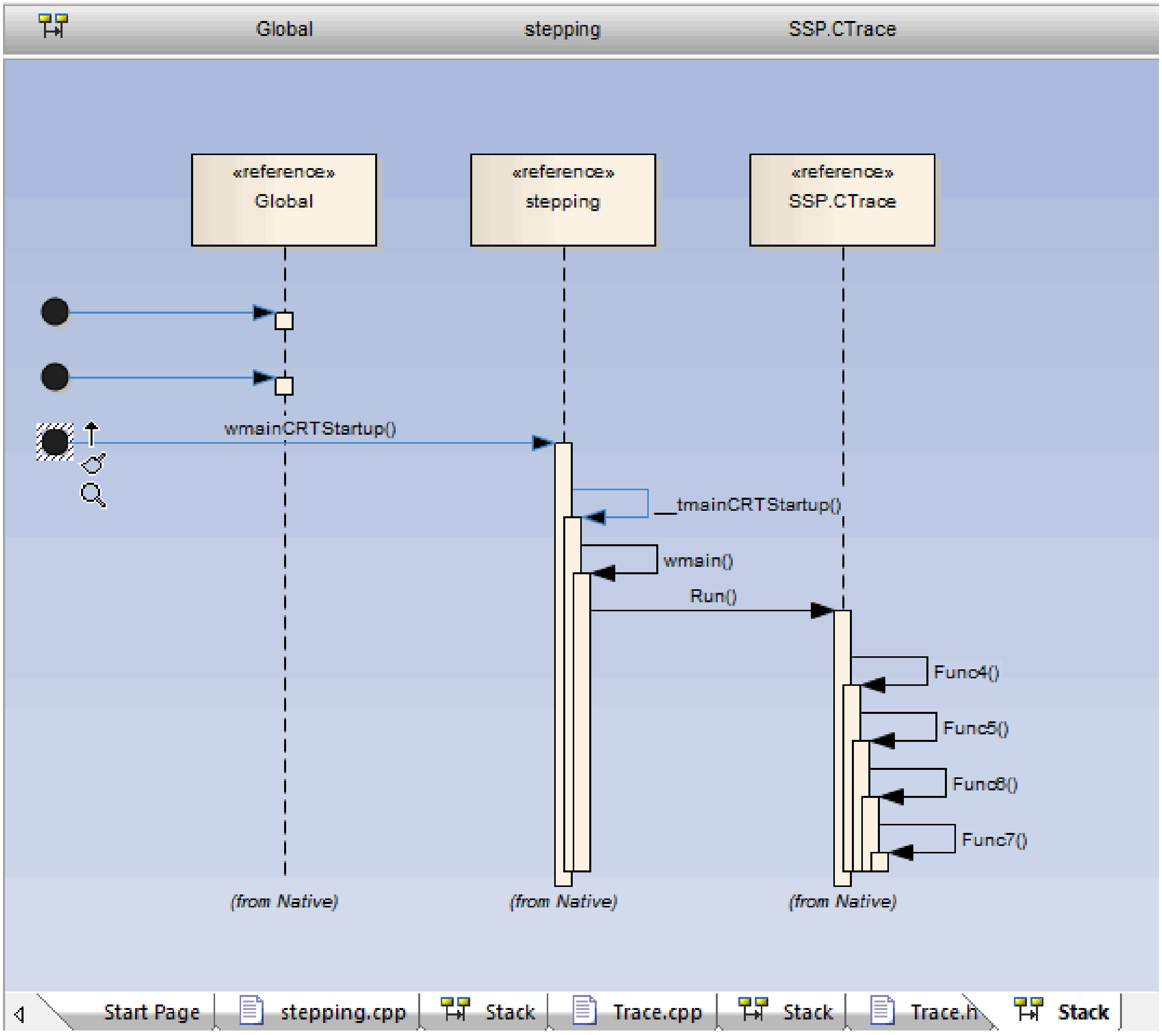
CREATED WITH ENTERPRISE ARCHITECT

Table of Contents

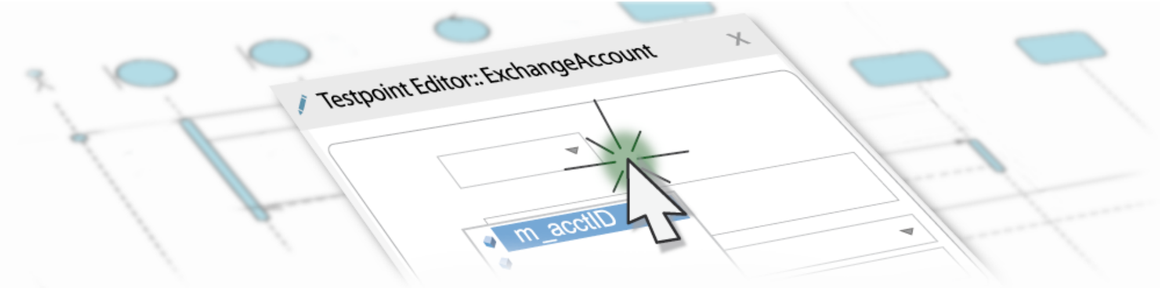
Introduction

Enterprise Architect, in addition to its powerful features as a fully fledged Integrated Development Environment, provides tools for visualizing and analyzing software execution that have a profound effect on the way that code is managed, maintained and documented. Much of this capability is unique to Enterprise Architect which uses its rich visualization and modeling facilities to bring programming code to life effectively allowing the code to be put under the microscope.

Code hot spots and faults can be analyzed and errors and opportunities for speed improvements can be identified quickly and resolutions to errors found. These features can be used while building and testing software products but are also profoundly useful for documenting existing or legacy systems. Commonly there is no documentation for these systems and it prohibitively expensive or even intractable to get the code documented by conventional means. Enterprise Architect can create sophisticated, elegant and valuable models and documents that completely describes the code base including expressive sequence diagrams and static models. This type of documentation is a number of generations more advanced than the simplistic and often opaque documentation generation tools such as JavaDoc.



Visual Execution Analysis



The Visual Execution Analyzer (VEA) is made up of an advanced and powerful suite of tools that allow you to build, debug, record, profile, simulate and otherwise construct and verify your software development while keeping the code tightly integrated with your model. Enterprise Architect has rich support for a wide range of popular compilers and platforms, in particular the Java, .Net and Microsoft Windows C++ environments. Software development becomes a highly streamlined visual experience, quite unlike working in traditional environments.

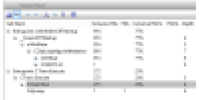
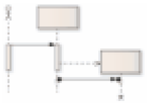
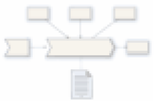
Enterprise Architect is itself modeled, built, compiled, debugged, tested, managed, profiled and otherwise constructed totally within the Visual Execution Analyzer built into Enterprise Architect. While the VEA can be used to complement other tool suites, it also shines when used as the primary development IDE in close coupling with the model and project management capabilities provided by Enterprise Architect.

Access

Ribbon	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+F12

Execution Analyzer Features

Feature	Description
Build & Debug	Using Analyzer Scripts linked to Model Packages, it is possible to tightly integrate the code/build/debug cycle into Enterprise Architect. For Java, .Net and Microsoft C++ in particular, it is simple to link to project code bases and take over the building and debugging within Enterprise Architect's Model Driven Development Environment. In addition to standard debugging features, the strong coupling with the model and the use of advanced debugging features such as Action Points makes Enterprise Architect the ideal platform from which to both design and construct your software application.
Simulation	Bring your behavioral models to life with instant, real-time behavioral model execution. Coupled with tools to manage triggers, events, guards, effects, breakpoints and simulation variables, plus the ability to visually track execution at run-time, the Simulator is a powerful means of 'watching the wheels turn' by visualizing execution of your behavioral models.
Profiling	Lift the hood on software performance and see what is actually going on. Quickly gain a clear picture of why certain tasks behave poorly or worse than expected.

	Whether its Microsoft .NET, native C++ or Java, use profiles to effectively judge changes in performance over your software lifecycle.
Recording Execution 	Record the execution of code without the need for instrumentation. Control the amount of detail through filters and stack depth. Generate beautiful Sequence diagrams and diagrams that illustrate Class collaboration. Use recording to create Test Domain diagrams that can be used with the VEA Testpoints feature.
Testing 	Create and manage test scripts for model elements. Explore the Testing interface, supporting unit, integration, scenario, system, inspection and acceptance tests. Employ programming by contract methodology with the Testpoints facility.
Object Workbench 	Workbench Class behavior on the fly, by instantiating them in the Object Workbench and then invoking their operations. You can even pass objects on the workbench as parameters to other workbench objects.
Visual Execution Analyzer Samples 	Try our sample patterns to set up and explore some of the powerful features of the Visual Execution Analyzer.

Benefits of the Execution Analyzer

The Execution Analyzer provides an integrated development and testing environment for multiple platforms, including Microsoft .NET, Java, Native C++, Mono and Android. It includes a feature-rich debugger, execution recording and profiling, and Testpoint management.

It helps you to generate Sequence, Test Domain Class and Collaborative Class diagrams from a single recording. This is a great way to understand and document your application.

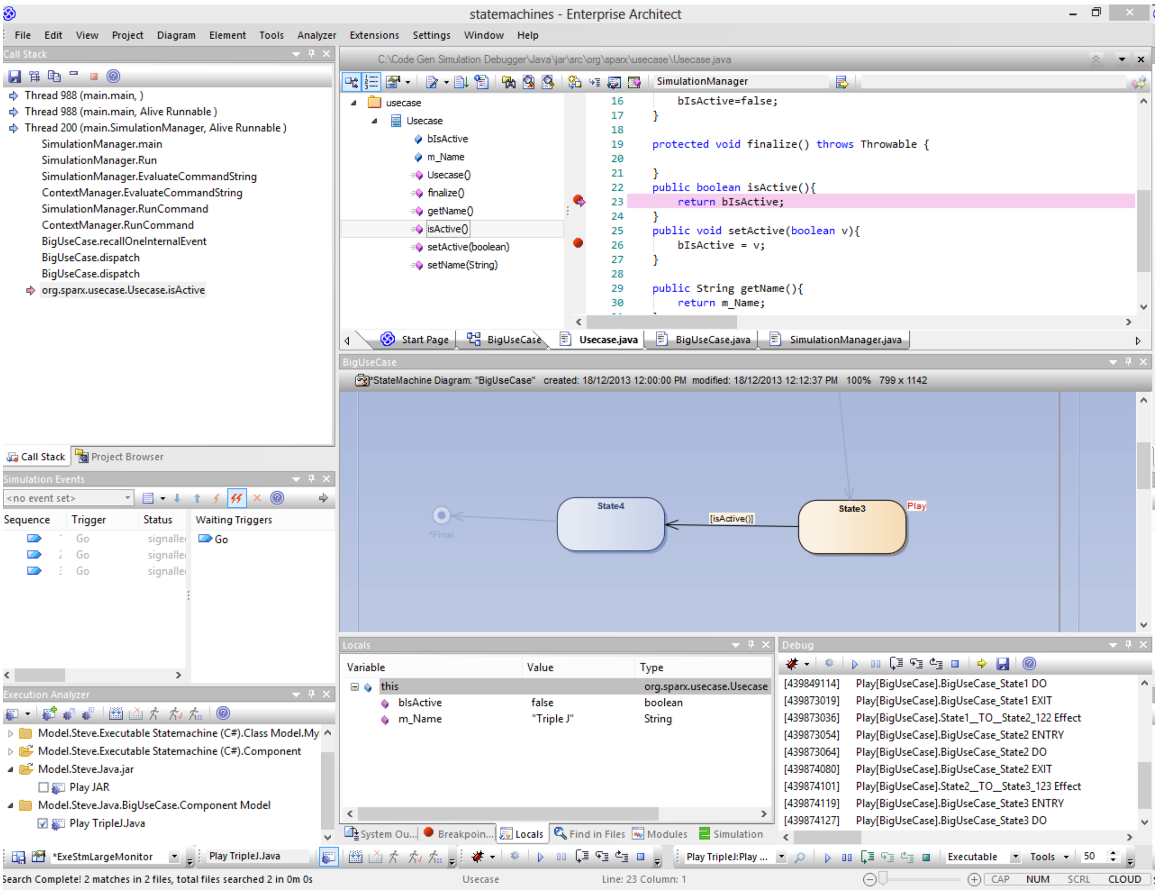
- Visualize program execution
- Optimize existing system resources and understand resource allocation
- Verify that the system is following the rules as designed
- Produce high quality documentation that more accurately reflects system behavior
- Understand how and why systems work
- Train new employees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

Operations

Operation
Simulate UML behavior models to verify their logical and design correctness, for: • Activities • Interactions and Sequences • StateMachines
Record executing programs and represent the behavior as a UML Sequence diagram; recording is supported for: • Microsoft Windows Native C, C++, Visual Basic • Microsoft .NET Family (C#, J#, VB) • Java • Mono • Android • PHP
Quickly view / report on behaviors of running applications. Profiling is supported for these platforms: • Microsoft Native C, C++, Visual Basic • Microsoft .NET Family (C#, J#, VB) (including any unmanaged / managed code mix) • Java • Mono
Testpoints Management provides a facility to define the constraints on a Class model as contracts. The contracts provide the assets on which to create Test domains. A single Testpoint domain can then be used to test and report the behavior of multiple applications. You can also use the Execution Analyzer to record a Use Case and generate a Test Domain diagram with very little effort. Any existing Testpoints are automatically linked to the generated domain or the Test Domain diagram can be used as the context for new contract compositions. How an application behaves for a given Test domain can be seen immediately in real time! Results are displayed in the Testpoint report window every time a contract is passed or failed. The decoupling of test measurement from the code-base has a number of benefits, one of which is aiding the reconciliation of multiple systems with a common Test domain, rather than one another. The Testpoint system supports these contracts: • Class invariants • Method pre-conditions • Method post-conditions • Line conditions
Create and work with objects created within the Enterprise Architect modeling environment using a dynamic Object Workbench. • Create objects from Class model • Invoke methods and view results • Workbench Class collaboration • Pass objects as parameters to other objects • Full debugging features including recording
Run nUnit and jUnit tests for Java and Microsoft .NET Record and document results.

Execution Recording and Profiling both acquire a collection of relevant code files, which you can reverse-engineer to the current model in a single operation.

Build & Debug



Thread 988 (main.main,)
Thread 988 (main.main, Alive Runnable)
Thread 200 (main.SimulationManager, Alive Runnable)
SimulationManager.main
SimulationManager.Run
SimulationManager.EvaluateCommandString
ContextManager.EvaluateCommandString
SimulationManager.RunCommand
ContextManager.RunCommand
BigUseCase.recallOneInternalEvent
BigUseCase.dispatch
BigUseCase.dispatch
org.sparx.usecase.Usecase.isActive

```
        bIsActive=false;
    }

    protected void finalize() throws Throwable {

    }
    public boolean isActive(){
        return bIsActive;
    }
    public void setActive(boolean v){
        bIsActive = v;
    }

    public String getName(){
        return m_Name;
```

*StateMachine Diagram: "BigUseCase" created: 18/12/2013 12:00:00 PM modified: 18/12/2013 12:12:37 PM 100% 799 x 1142

Sequence	Trigger	Status	Waiting Triggers
	Go	signalle	Go
	Go	signalle	
	Go	signalle	

Variable	Value	Type
this		org.sparx.usecase.Usecase
blsActive	false	boolean
m_Name	"Triple J"	String

```
[439849114]   Play[BigUseCase].BigUseCase_State1 DO
[439873019]   Play[BigUseCase].BigUseCase_State1 EXIT
[439873036]   Play[BigUseCase].State1__TO__State2_122 Effect
[439873054]   Play[BigUseCase].BigUseCase_State2 ENTRY
[439873064]   Play[BigUseCase].BigUseCase_State2 DO
[439874080]   Play[BigUseCase].BigUseCase_State2 EXIT
[439874101]   Play[BigUseCase].State2__TO__State3_123 Effect
[439874119]   Play[BigUseCase].BigUseCase_State3 ENTRY
[439874127]   Play[BigUseCase].BigUseCase_State3 DO
```

Model.Steve.Executable Statemachine (C#).Class Model.My
Model.Steve.Executable Statemachine (C#).Component
Model.Steve.Java.jar
□ Play JAR
Model.Steve.Java.BigUseCase.Component Model
☑ Play TripleJ.Java

Search Complete! 2 matches in 2 files, total files searched 2 in 0m 0s

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.

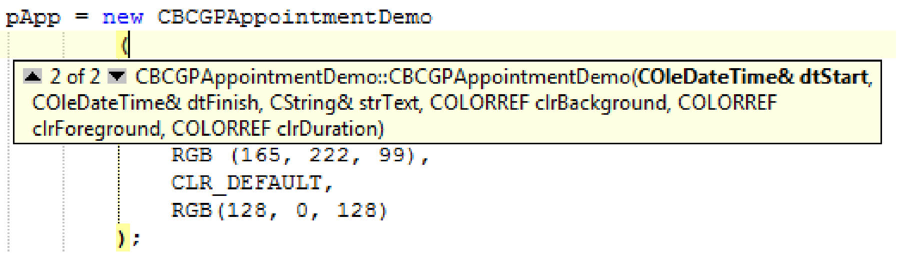
It is recommended that new users take the time to fully understand the use of the Analyzer Scripts and how they tie the model to the code, to the compilers and to other tools necessary for building software.

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.

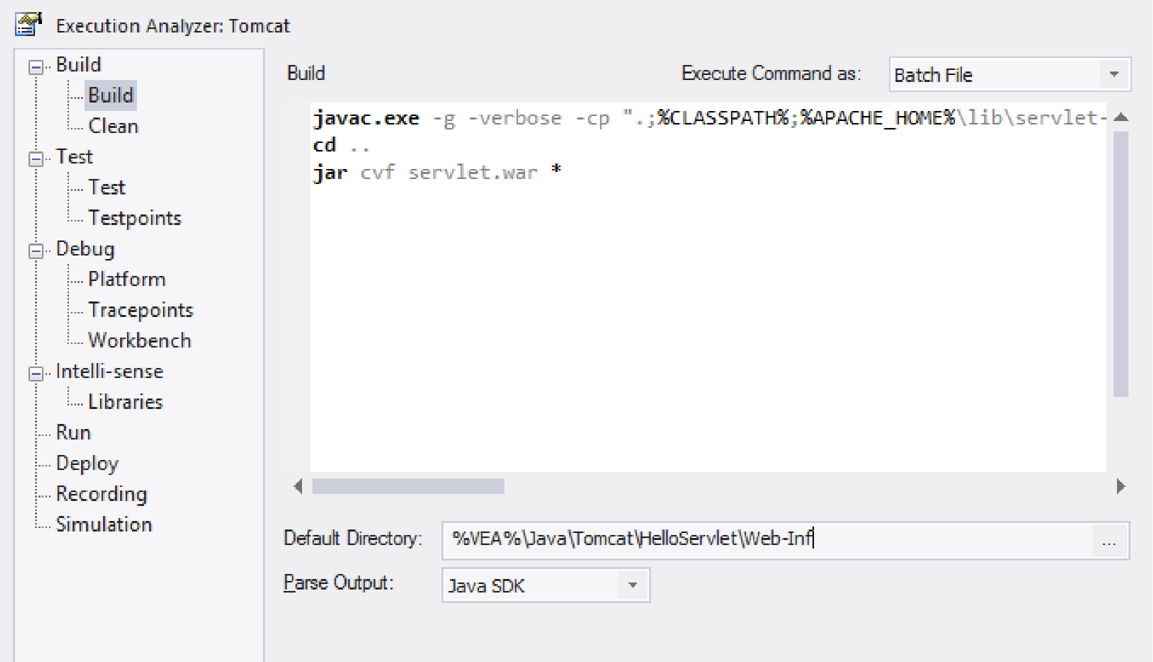
Enterprise Architect's Model Driven Development Environment (MDDE) supports this approach and provides a set of flexible tools to increase productivity and reduce errors. These include the ability to define the architecture and design in models, generate code from these models, synchronize the code with 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
	
	`pApp = new CBCGPAppointmentDemo`
	`(`
	▲ 2 of 2 ▼ CBCGPAppointmentDemo::CBCGPAppointmentDemo(**COleDateTime& dtStart**, COleDateTime& dtFinish, CString& strText, COLORREF clrBackground, COLORREF clrForeground, COLORREF clrDuration)
	`RGB (165, 222, 99),`
	`CLR_DEFAULT,`
	`RGB(128, 0, 128)`
	`);`

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.



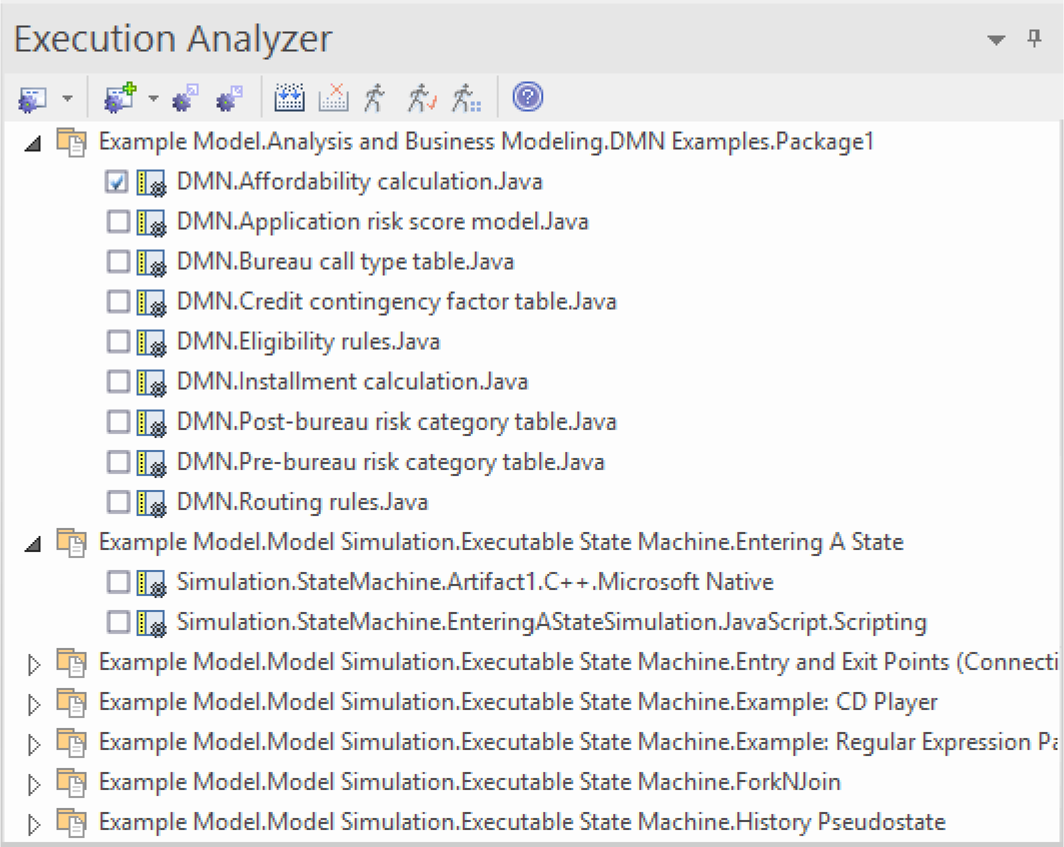
Execute Command as: Batch File

```
javac.exe -g -verbose -cp ".;%CLASSPATH%;%APACHE_HOME%\lib\servlet-
cd ..
jar cvf servlet.war *
```

Default Directory: %VEA%\Java\Tomcat\HelloServlet\Web-Inf

Parse Output: Java SDK

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



Execution Analyzer

◢ Example Model.Analysis and Business Modeling.DMN Examples.Package1
- ☑ DMN.Affordability calculation.Java
- □ DMN.Application risk score model.Java
- □ DMN.Bureau call type table.Java
- □ DMN.Credit contingency factor table.Java
- □ DMN.Eligibility rules.Java
- □ DMN.Installment calculation.Java
- □ DMN.Post-bureau risk category table.Java
- □ DMN.Pre-bureau risk category table.Java
- □ DMN.Routing rules.Java

◢ Example Model.Model Simulation.Executable State Machine.Entering A State
- □ Simulation.StateMachine.Artifact1.C++.Microsoft Native
- □ Simulation.StateMachine.EnteringAStateSimulation.JavaScript.Scripting

▷ Example Model.Model Simulation.Executable State Machine.Entry and Exit Points (Connecti

▷ Example Model.Model Simulation.Executable State Machine.Example: CD Player

▷ Example Model.Model Simulation.Executable State Machine.Example: Regular Expression Pa

▷ Example Model.Model Simulation.Executable State Machine.ForkNJoin

▷ Example Model.Model Simulation.Executable State Machine.History Pseudostate

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. They can however, in a few lines, 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.

Access

Ribbon	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+F12

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 a Package.
	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 'Browse Project' 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.
?	Display the Help topic for this window.

Context Menu Options:

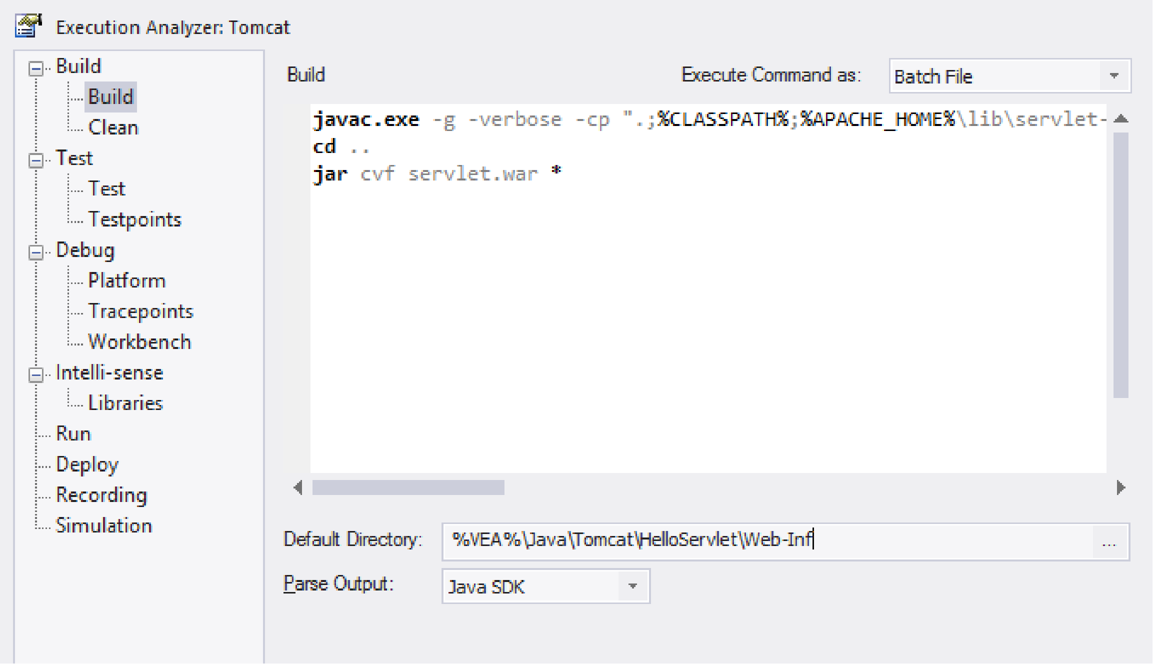
Right-click on the required script or Package to display the context menus.

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.
Select In Project Browser	Highlight the selected Package in the Project Browser. Display the Project Browser, 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' command of the selected script.

Run Executable Statemachine	Start a simulation of the selected Executable Statemachine Artifact.
Start Simulation	Start the simulation referenced by the 'Analyzer Script Simulation' page.
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.
Set as Model Default	Set the selected script as the default script for the model. The icon to the left of the script changes color; any previous model default reverts to normal.
Help	Display the Help topic for this window.

Analyzer Script Editor

The Analyzer Script Editor is a straightforward user interface with a tree view on the left for easy navigation of features, and a content view on the right.



Execute Command as: Batch File

```
javac.exe -g -verbose -cp ".;%CLASSPATH%;%APACHE_HOME%\lib\servlet-
cd ..
jar cvf servlet.war *
```

Default Directory: %VEA%\Java\Tomcat\HelloServlet\Web-Inf

Parse Output: Java SDK

Access

From the 'Execution Analyzer' window, either:

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

Ribbon	Code > Analyzer > Edit Analyzer Scripts Execute > Analyzer
Keyboard Shortcuts	Shift+F12

Execution Analyzer Scripts

Task - Page	Action
Build - Build	Enter 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 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 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.
Run	Enter a script or command to run the 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 look like 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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Context Menu	Project Browser | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	Shift+F12

Execute Command As:

Batch File

Use this option to create a shell script. The script 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"

Build Script

Write your script in the large 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. 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.

Default Directory

The default directory path in which the build script process will run.

Parse Output

This enables you to 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.

Deploy after Build

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

Notes

To execute the Build Script, click on the Package in the Project Browser and either:

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

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer > select and run script
Context Menu	Project Browser | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	Shift+F12

Aspects

Aspect	Detail
Action	Enter the command to be executed when you select 'Clean' from the script context menu.
Example	devenv.com /Clean Debug MyProject.sln

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 press Alt+3 while selecting the test case Class element.

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Context Menu	Project Browser | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	Shift+F12

Actions

Execute Command As:

Process

Enter the path to a program or batch file to run, followed by any parameters.

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

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.

Default Directory

Preset to the Build default directory.

Parse Output

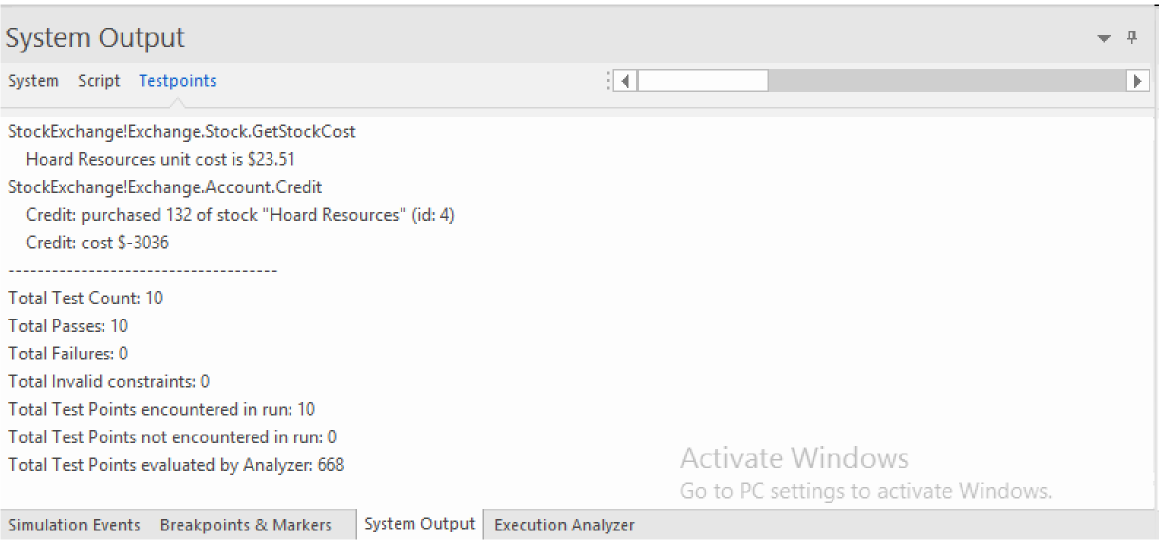
When a parser is selected, output of nUnit and jUnit tests can be parsed, saved and

	managed from the model; (Alt+3). Be aware that output is only captured when a parser is selected.
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.



```
StockExchange!Exchange.Stock.GetStockCost
   Hoard Resources unit cost is $23.51
StockExchange!Exchange.Account.Credit
   Credit: purchased 132 of stock "Hoard Resources" (id: 4)
   Credit: cost $-3036
------------------------------------
Total Test Count: 10
Total Passes: 10
Total Failures: 0
Total Invalid constraints: 0
Total Test Points encountered in run: 10
Total Test Points not encountered in run: 0
Total Test Points evaluated by Analyzer: 668
```

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Context Menu	Project Browser | Right-click on Package | Execution 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	Enter 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 doing so provides access to a great many benefits. Here are some:

- 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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+F12

Notes

- An Analyzer script is not necessary for debugging Enterprise Architect model scripts (JavaScript, VBScript etc.)

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. 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
- Debugging under Windows Vista (x64) - if you encounter problems debugging with Enterprise Architect on a 64-bit platform, you should build a platform specific configuration in Visual Studio; that is, do not specify the AnyCPU configuration, specify either Win32 or x64 explicitly

UAC-Enabled Operating Systems

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

The Enterprise Architect Visual Execution 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

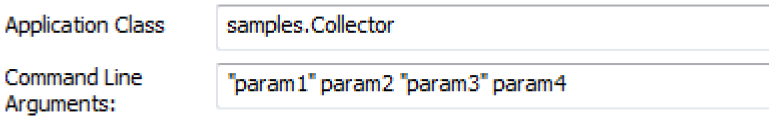
This section describes how to set up Enterprise Architect for debugging Java applications and Web Servers.

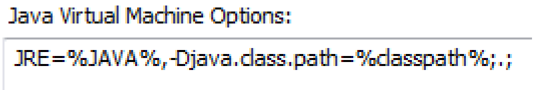
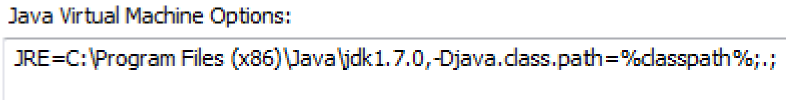
General Setup for Java

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());  Application Class: samples.Collector Command Line Arguments: "param1" param2 "param3" param4
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 (JRE) as the path to be searched for the jvm.dll; this is the DLL supplied as part of the Java runtime environment or Java JDK from Sun Microsystems™. 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:

	 Java Virtual Machine Options: JRE=%JAVA%,-Djava.class.path=%classpath%;.; - Or an absolute path to the JDK installation directory and an environment variable classpath:  Java Virtual Machine Options: JRE=C:\Program Files (x86)\Java\jdk1.7.0,-Djava.class.path=%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 researched from 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

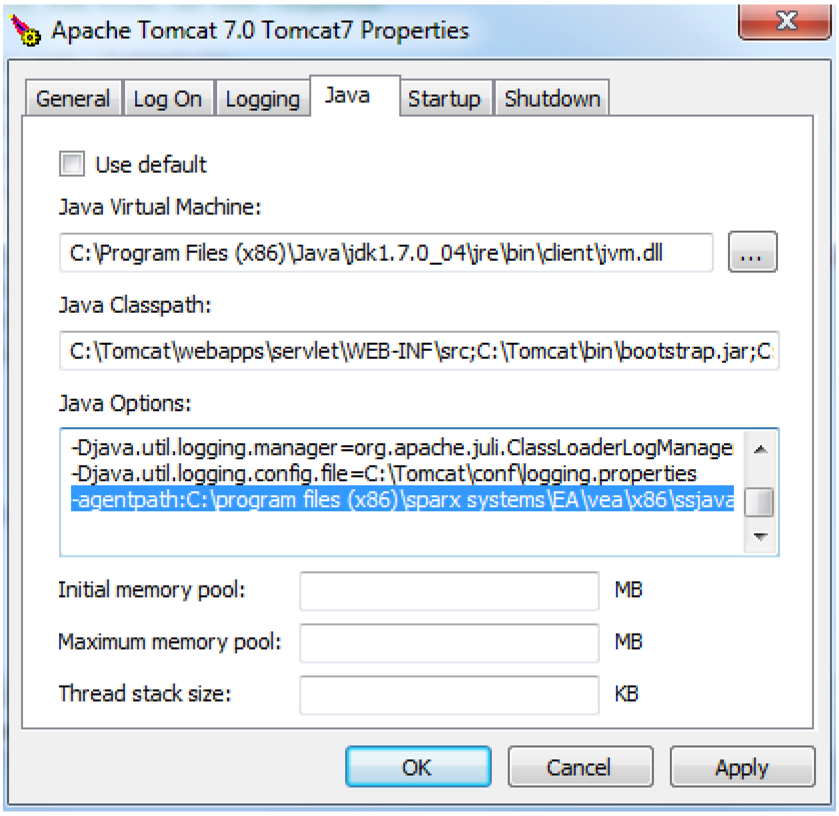
Therefore, the root of this path is included in the JBOSS_CLASSPATH.

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_CLASSPATH%;%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.



□ Use default

Java Virtual Machine:

C:\Program Files (x86)\Java\jdk1.7.0_04\jre\bin\client\jvm.dll

Java Classpath:

C:\Tomcat\webapps\servlet\WEB-INF\src;C:\Tomcat\bin\bootstrap.jar;C

Java Options:

```
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManage
-Djava.util.logging.config.file=C:\Tomcat\conf\logging.properties
-agentpath:C:\program files (x86)\sparx systems\EA\vea\x86\ssjava
```

Initial memory pool: MB

Maximum memory pool: MB

Thread stack size: KB

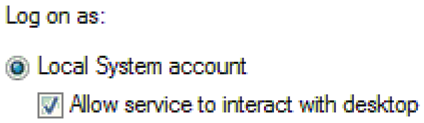
These three points are important:

- 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.



Log on as:

● Local System account

☑ Allow service to interact with desktop

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 Set up for .NET
- Debugging an Unmanaged Application
- Debug COM Interop
- Debug ASP .NET

General Set up for .NET

This is the general set up 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 startup.
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 plugin 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 Plugins 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, under Windows Vista the name of the IIS process is w3wp.exe.

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 Linux as well as on 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. This is achieved through 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 import 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

Debugger Configuration (Windows)

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 is run.
Application Path	The full path of the Mono application.

Command Line Arguments	Any parameters to pass to the program. Surround the parameters in double quotes if they contain spaces.

Debugger Configuration (Linux)

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.
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 Wine / Windows root path of the source code • remotepath: the native Linux root path of the source code • 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 Debugger window and socket messages are logged to the specified output file

The DebugRun Page

This page is optional and is only useful where Mono and Enterprise Architect will be running on the same machine. What it provides is 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.

Starting Mono with Debugger Support using an Analzyer Script

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

Linux:

cd path-to-program

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

Windows:

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 and

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

Starting Mono with Debugger Support from 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:

Linux:

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

Windows:

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

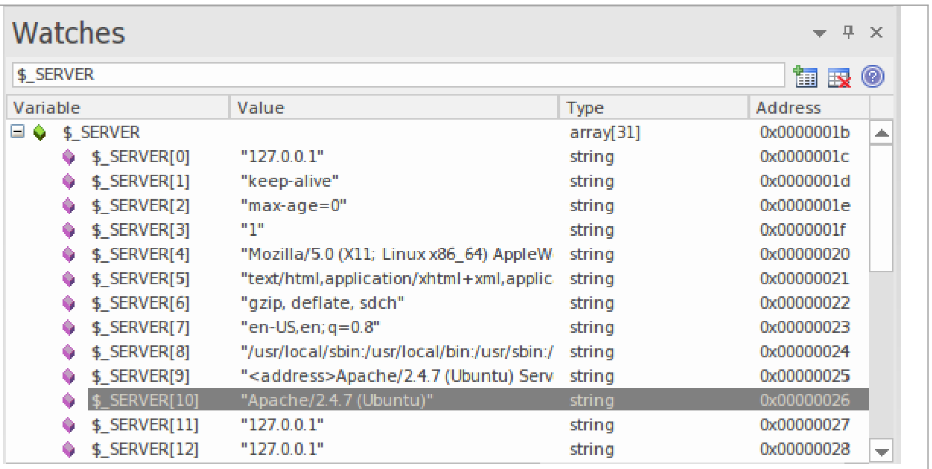
The Enterprise Architect PHP Debugger enables you to debug PHP.exe scripts. This section discusses basic set up and the various debugging scenarios that are commonly encountered; the scenarios concern themselves with the mapping of file paths, which is critical to the success of a remote debugging session.

- 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 Machine Apache Server	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 • 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.

Watches

$_SERVER

Variable	Value	Type	Address
$_SERVER		array[31]	0x0000001b
$_SERVER[0]	"127.0.0.1"	string	0x0000001c
$_SERVER[1]	"keep-alive"	string	0x0000001d
$_SERVER[2]	"max-age=0"	string	0x0000001e
$_SERVER[3]	"1"	string	0x0000001f
$_SERVER[4]	"Mozilla/5.0 (X11; Linux x86_64) AppleW	string	0x00000020
$_SERVER[5]	"text/html,application/xhtml+xml,applic	string	0x00000021
$_SERVER[6]	"gzip, deflate, sdch"	string	0x00000022
$_SERVER[7]	"en-US,en;q=0.8"	string	0x00000023
$_SERVER[8]	"/usr/local/sbin:/usr/local/bin:/usr/sbin:/	string	0x00000024
$_SERVER[9]	"<address>Apache/2.4.7 (Ubuntu) Serv	string	0x00000025
$_SERVER[10]	"Apache/2.4.7 (Ubuntu)"	string	0x00000026
$_SERVER[11]	"127.0.0.1"	string	0x00000027
$_SERVER[12]	"127.0.0.1"	string	0x00000028

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Context Menu	Project Browser | Right-click on Package | Execution Analyzer
Keyboard Shortcuts	Shift+F12

Set up the GNU Debugger

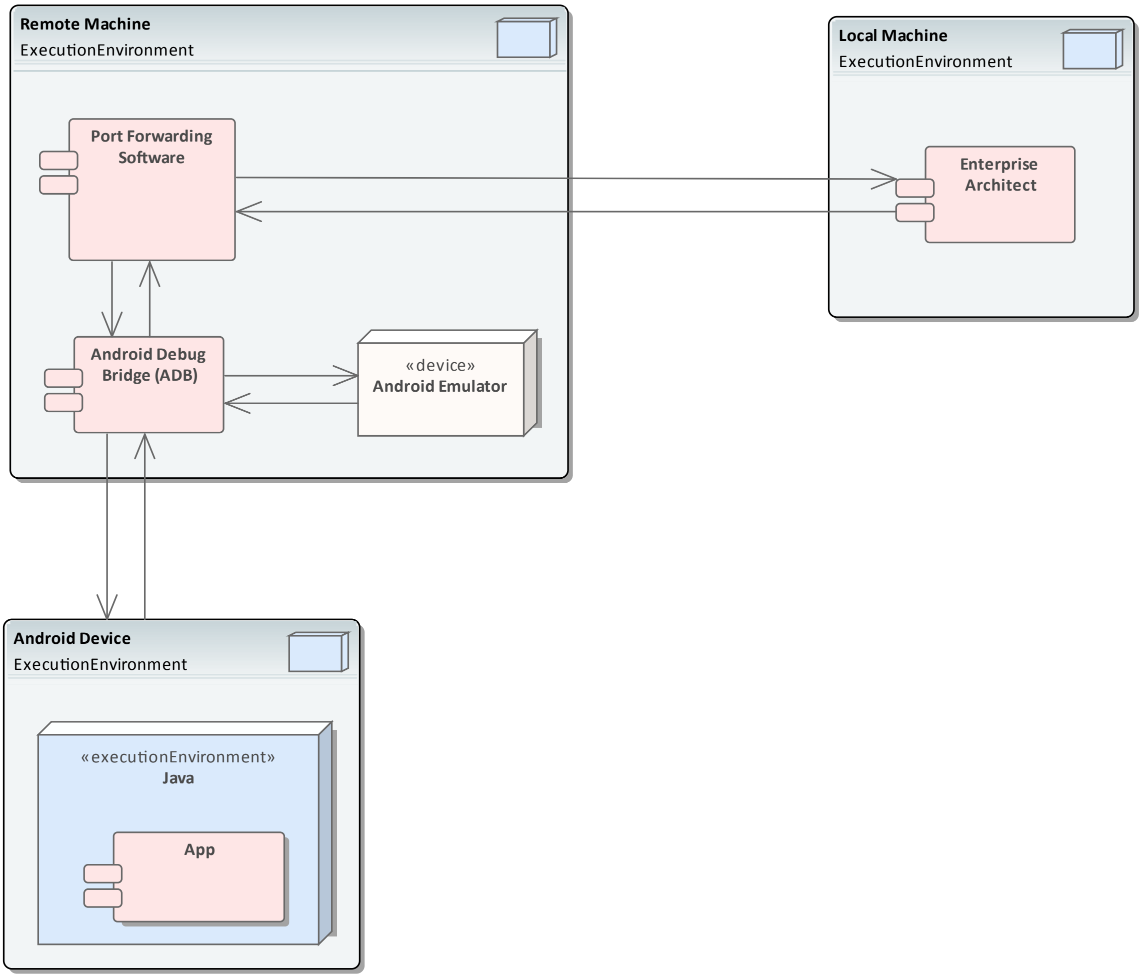
Task	Details
Set up Script	An Analyzer Script is a basic requirement for debugging in Enterprise Architect; you create a script using the Execution Analyzer toolbar. On the 'Platform' page of the Execution Analyzer Script Editor, in the 'Debugger' field click on the drop-down arrow and select 'GDB'.
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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer > select and run script
Context Menu	Project Browser | 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 function	If selected, any Trace statements executed during the debug session run are prefixed with the current function call.

Workbench Set Up

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > 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 Set Up

This is the general set up 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 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 startup.
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.

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+F12

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 'Code > Build and Run > 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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+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 > Explore > Browse > 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

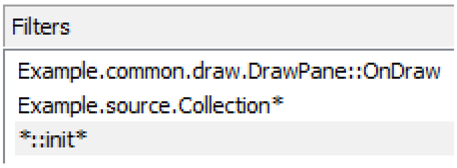
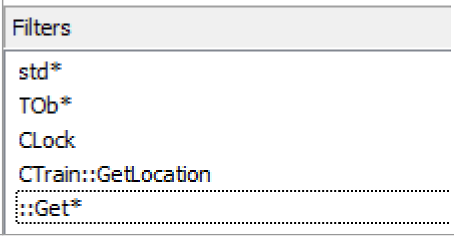
On the Execution Analyzer window, either:

- 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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > 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>)  Filters Example.common.draw.DrawPane::OnDraw Example.source.Collection* *::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 std* TOb* CLock CTrain::GetLocation ::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.

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 'Services' page or
- Click on in the window Toolbar, select the Package in which to create a new script, and select the 'Services' page

Ribbon	Execute > Run > Analyzer or Code > Configure > Analyzer > Edit Analyzer Scripts
Keyboard Shortcuts	Shift+F12

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	Code > Configure > Analyzer > Edit Analyzer Scripts Execute > Run > Analyzer
Keyboard Shortcuts	Shift+F12

Build Application

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

Access

Ribbon	Code > Build and Run > Build Execute > Run > Build
Keyboard Shortcuts	Ctrl+Shift+F12
Other	'Build' toolbar > Execution Analyzer window |

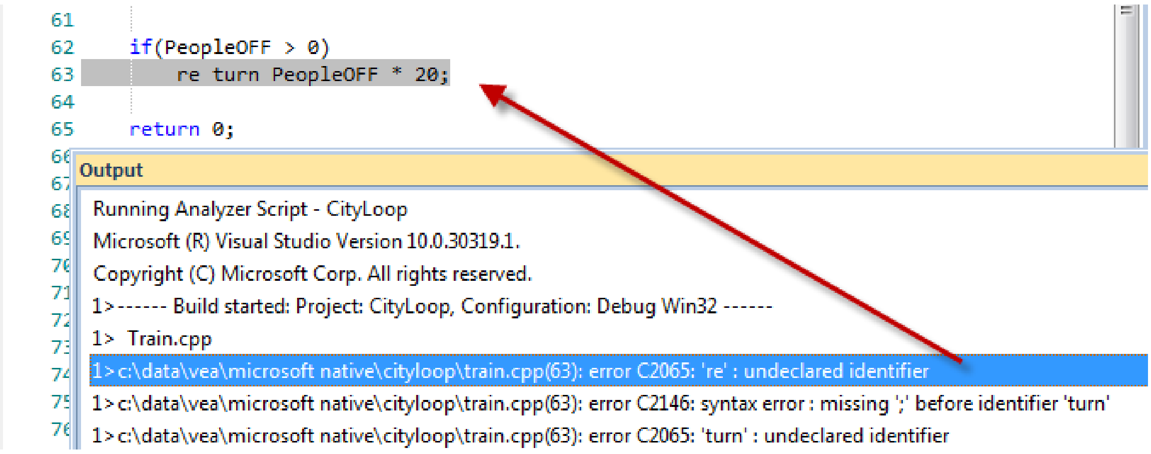
Action

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.

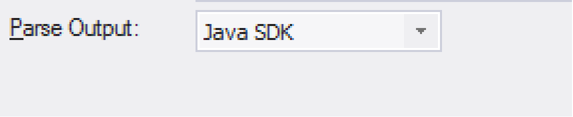


```
    if(PeopleOFF > 0)
        re turn PeopleOFF * 20;

    return 0;
```

```
Running Analyzer Script - CityLoop
Microsoft (R) Visual Studio Version 10.0.30319.1.
Copyright (C) Microsoft Corp. All rights reserved.
1>------ Build started: Project: CityLoop, Configuration: Debug Win32 ------
1>  Train.cpp
1>c:\data\vea\microsoft native\cityloop\train.cpp(63): error C2065: 're' : undeclared identifier
1>c:\data\vea\microsoft native\cityloop\train.cpp(63): error C2146: syntax error : missing ';' before identifier 'turn'
1>c:\data\vea\microsoft native\cityloop\train.cpp(63): error C2065: 'turn' : undeclared identifier
```

Tip

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

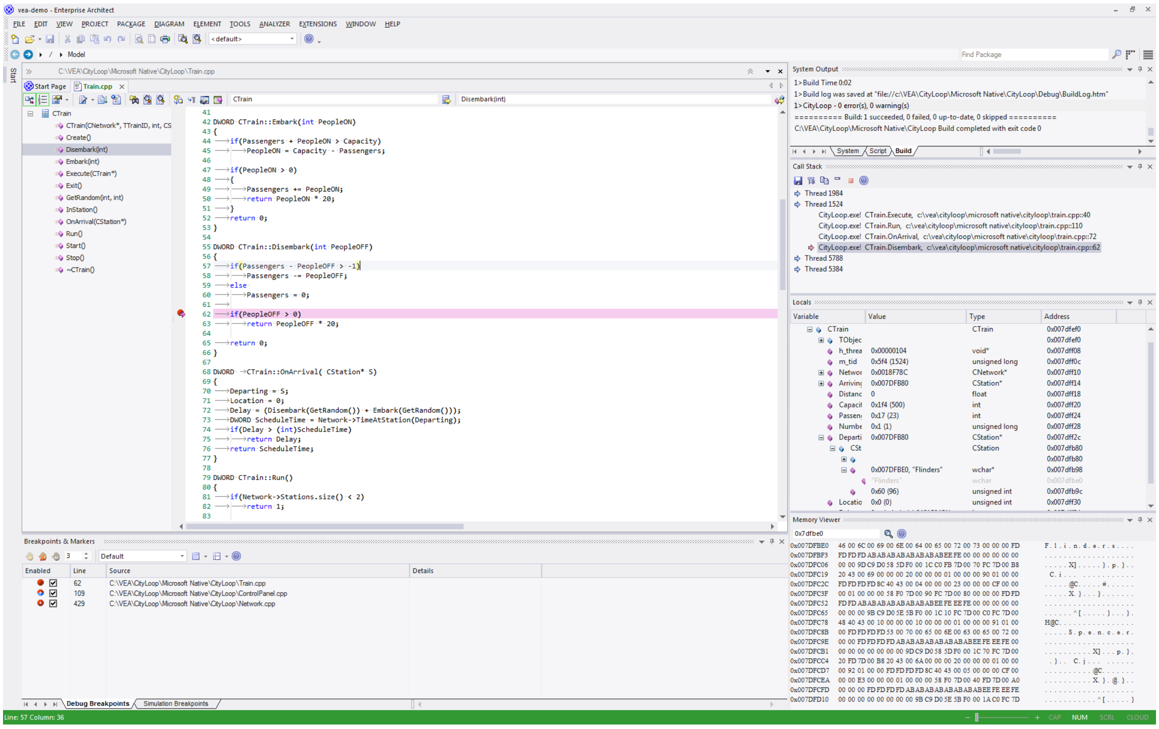


Parse Output: Java SDK

Access

Ribbon	Start > Explore > Browse > System Output
Keyboard Shortcuts	Ctrl+Shift+8

Debugging



```
DWORD CTrain::Embark(int PeopleON)
{
	if(Passengers + PeopleON > Capacity)
		PeopleON = Capacity - Passengers;

	if(PeopleON > 0)
	{
		Passengers += PeopleON;
		return PeopleON * 20;
	}
	return 0;
}

DWORD CTrain::Disembark(int PeopleOFF)
{
	if(Passengers - PeopleOFF > -1)
		Passengers -= PeopleOFF;
	else
		Passengers = 0;
	
	if(PeopleOFF > 0)
		return PeopleOFF * 20;

	return 0;
}

DWORD 	CTrain::OnArrival( CStation* S)
{
	Departing = S;
	Location = 0;
	Delay = (Disembark(GetRandom()) + Embark(GetRandom()));
	DWORD ScheduleTime = Network->TimeAtStation(Departing);
	if(Delay > (int)ScheduleTime)
		return Delay;
	return ScheduleTime;
}

DWORD CTrain::Run()
{
	if(Network->Stations.size() < 2)
		return 1;

```

```
1>Build Time 0:02
1>Build log was saved at "file://c:\VEA\CityLoop\Microsoft Native\CityLoop\Debug\BuildLog.htm"
1>CityLoop - 0 error(s), 0 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
C:\VEA\CityLoop\Microsoft Native\CityLoop Build completed with exit code 0
```

```
Thread 1984
Thread 1524
    CityLoop.exe!  CTrain.Execute,  c:\vea\cityloop\microsoft native\cityloop\train.cpp::40
    CityLoop.exe!  CTrain.Run,  c:\vea\cityloop\microsoft native\cityloop\train.cpp::110
    CityLoop.exe!  CTrain.OnArrival,  c:\vea\cityloop\microsoft native\cityloop\train.cpp::72
  ⇨ CityLoop.exe!  CTrain.Disembark,  c:\vea\cityloop\microsoft native\cityloop\train.cpp::62
Thread 5788
Thread 5384
```

Variable	Value	Type	Address
CTrain		CTrain	0x007dfef0
TObjec			0x007dfef0
h_threa	0x00000104	void*	0x007dff08
m_tid	0x5f4 (1524)	unsigned long	0x007dff0c
Networ	0x0018F78C	CNetwork*	0x007dff10
Arriving	0x007DFB80	CStation*	0x007dff14
Distanc	0	float	0x007dff18
Capacit	0x1f4 (500)	int	0x007dff20
Passen	0x17 (23)	int	0x007dff24
Numbe	0x1 (1)	unsigned long	0x007dff28
Departi	0x007DFB80	CStation*	0x007dff2c
CSt		CStation	0x007dfb80
			0x007dfb80
	0x007DFBE0, "Flinders"	wchar*	0x007dfb98
	"Flinders"	wchar	0x007dfbe0
	0x60 (96)	unsigned int	0x007dfb9c
Locatio	0x0 (0)	unsigned int	0x007dff30

0x7dfbe0

```
0x007DFBE0   46 00 6C 00 69 00 6E 00 64 00 65 00 72 00 73 00 00 00 FD   F.l.i.n.d.e.r.s....
0x007DFBF3   FD FD FD AB AB AB AB AB AB AB AB EE FE 00 00 00 00 00 00   ...................
0x007DFC06   00 00 9D C9 D0 58 5D F0 00 1C C0 FB 7D 00 70 FC 7D 00 B8   .....X].....}.p.}..
0x007DFC19   20 43 00 69 00 00 00 20 00 00 00 01 00 00 00 90 01 00 00    C.i... ...........
0x007DFC2C   FD FD FD FD 8C 40 43 00 04 00 00 00 23 00 00 00 CF 00 00   .....@C....#......
0x007DFC3F   00 01 00 00 00 58 F0 7D 00 90 FC 7D 00 80 00 00 00 FD FD   .....X.}...}.......
0x007DFC52   FD FD AB AB AB AB AB AB AB AB EE FE EE FE 00 00 00 00 00   ...................
0x007DFC65   00 00 00 9B C9 D0 5E 5B F0 00 1C 10 FC 7D 00 C0 FC 7D 00   ......^[.....}...}.
0x007DFC78   48 40 43 00 10 00 00 00 10 00 00 00 01 00 00 00 91 01 00   H@C................
0x007DFC8B   00 FD FD FD FD 53 00 70 00 65 00 6E 00 63 00 65 00 72 00   .....S.p.e.n.c.e.r.
0x007DFC9E   00 00 FD FD FD FD AB AB AB AB AB AB AB AB EE FE EE FE 00   ...................
0x007DFCB1   00 00 00 00 00 00 00 9D C9 D0 58 5D F0 00 1C 70 FC 7D 00   ..........X]...p.}.
0x007DFCC4   20 FD 7D 00 B8 20 43 00 6A 00 00 00 20 00 00 00 01 00 00    .}.. C.j... ......
0x007DFCD7   00 92 01 00 00 FD FD FD FD 8C 40 43 00 05 00 00 00 CF 00   ..........@C.......
0x007DFCEA   00 00 E3 00 00 00 01 00 00 00 58 F0 7D 00 40 FD 7D 00 A0   ..........X.}.@.}..
0x007DFCFD   00 00 00 FD FD FD FD AB AB AB AB AB AB AB AB EE FE EE FE   ...................
0x007DFD10   00 00 00 00 00 00 00 00 9B C9 D0 5E 5B F0 00 1A C0 FC 7D   ...........^[.....}
```

3

Enabled	Line	Source	Details
● ☑	62	C:\VEA\CityLoop\Microsoft Native\CityLoop\Train.cpp	
● ☑	109	C:\VEA\CityLoop\Microsoft Native\CityLoop\ControlPanel.cpp	
● ☑	429	C:\VEA\CityLoop\Microsoft Native\CityLoop\Network.cpp	

Line: 57 Column: 36

Enterprise Architect is more than a drawing tool. Every feature that you might expect in an IDE is also available. Comprehensive debugging environments and tools for many major platforms are provided. By integrating 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 execution can be done 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
- JavaSscript, VBScript and JScript
- PHP scripts on Apache web servers
- Remote Linux GDB processes using Enterprise Architect on Windows (how's that for interoperability?)
- 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. (Your artifact is safe!)

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 supported 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 > Analyze > Debugger > Open Debugger Execute > Run > Start > Run
Keyboard Shortcuts	Alt+8 (displays the Debug window) F6 (begins execution of the application being debugged)
Other	Right-click on Project Browser caption bar menu | Analyzer Toolbars | Debugging

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 > Analyze > 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 > Analyze > 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 > Analyze > 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 like breakpoints, but in Enterprise Architect they have special powers. You set any marker or breakpoint in the Source Code editor. They are visible in the left margin, and 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. 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 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 personal set allocated to each user of any model. It is private.

Access

Ribbon	Execute > Windows > 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 marker set	Check the Default ▾ field in the Breakpoints & Events window toolbar. 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

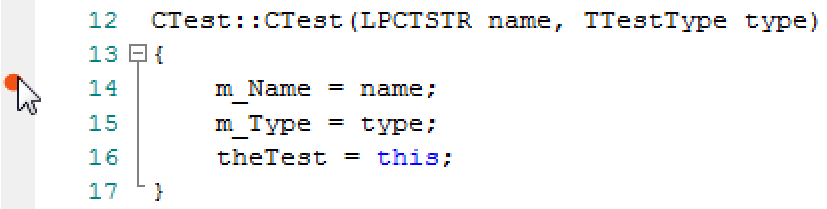
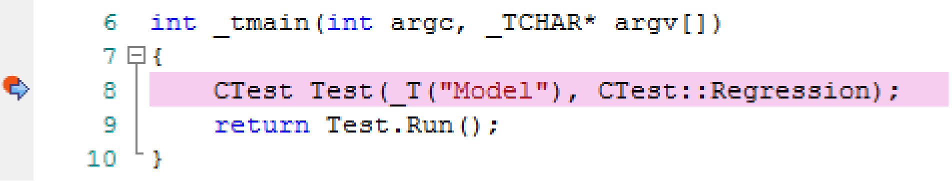
State	Remarks
🟠	*Debug Running:* Bound *Debug Not Running:* Enabled
●	*Debug Running:* Disabled *Debug Not Running:* Disabled
?	*Debug Running:* Not bound - this usually means that a module is yet to be loaded. Also, dlls are unloaded from time to time. *Debug Not Running:* N/a
!	*Debug Running:* Failed - 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.
	
	`CTest::CTest(LPCTSTR name, TTestType type)`
	`{`
	`    m_Name = name;`
	`    m_Type = type;`
	`    theTest = this;`
	`}`
	If the code is currently halted at a breakpoint, that point is indicated by a blue arrow next to the marker.
	
	`int _tmain(int argc, _TCHAR* argv[])`
	`{`
	`    CTest Test(_T("Model"), CTest::Regression);`
	`    return Test.Run();`
	`}`
	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.

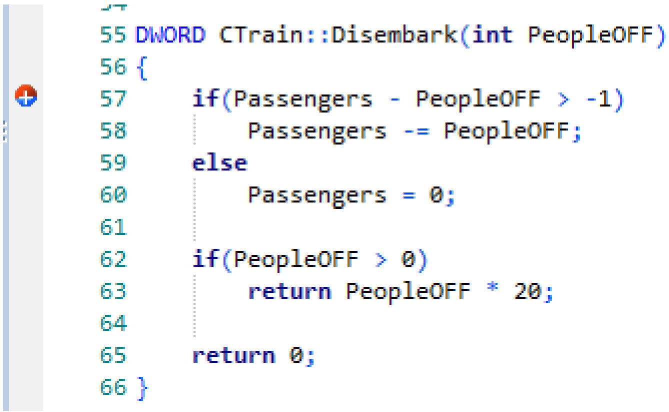
Tracepoint Markers are set in the code editor. Like breakpoints, they are placed on a line of code. When that line of code executes, the debugger evaluates the statement, the result of which is logged to the Debug window (or to file if overridden by the Analyzer script).

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

Step	Action
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. 

```
DWORD CTrain::Disembark(int PeopleOFF)
{
    if(Passengers - PeopleOFF > -1)
        Passengers -= PeopleOFF;
    else
        Passengers = 0;

    if(PeopleOFF > 0)
        return PeopleOFF * 20;

    return 0;
}
```

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

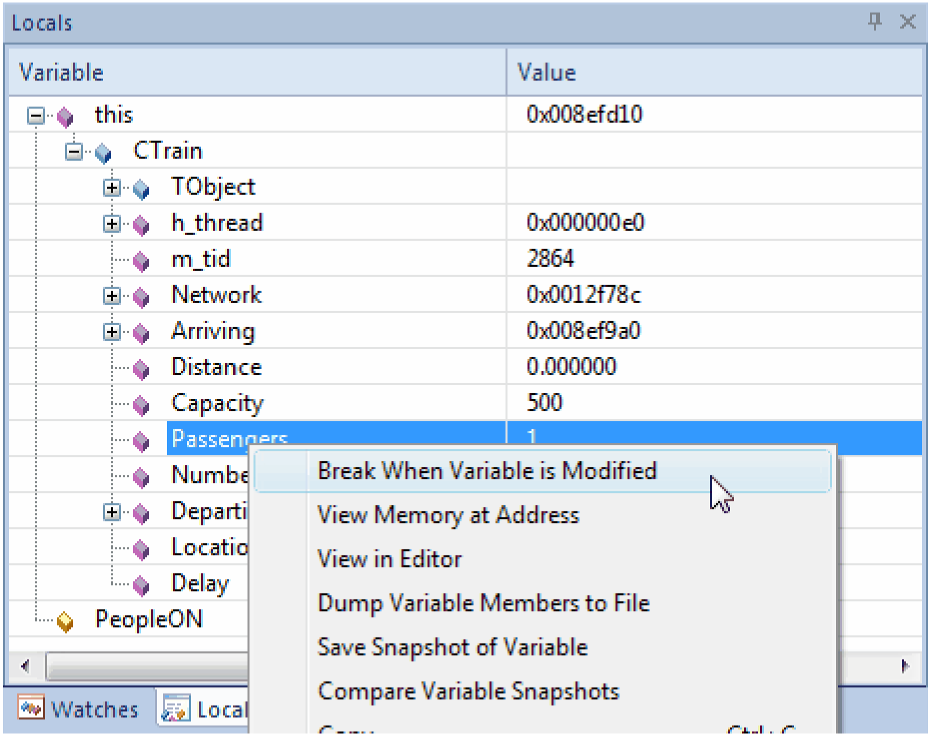
Data breakpoints can be set on a pre-determined memory variable to cause the debugger to halt execution at the line of code that has just caused the value of the variable to change. This can be useful when trying to track down the point at which a variable is modified during program execution, especially 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

<table>
<tr><th>Steps</th><th>Detail</th></tr>
<tr><td>1</td><td>Set a normal breakpoint in the code so you can choose a variable. Then run the debugger (F6).</td></tr>
<tr><td>2</td><td>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.



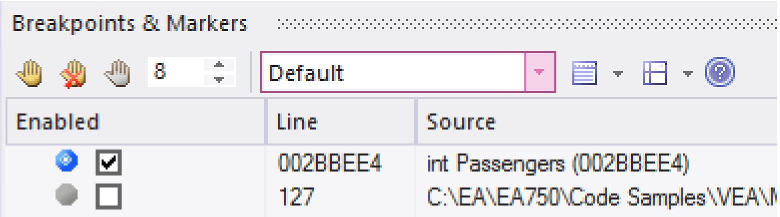
Locals

<table>
<tr><th>Variable</th><th>Value</th></tr>
<tr><td>this</td><td>0x008efd10</td></tr>
<tr><td>CTrain</td><td></td></tr>
<tr><td>TObject</td><td></td></tr>
<tr><td>h_thread</td><td>0x000000e0</td></tr>
<tr><td>m_tid</td><td>2864</td></tr>
<tr><td>Network</td><td>0x0012f78c</td></tr>
<tr><td>Arriving</td><td>0x008ef9a0</td></tr>
<tr><td>Distance</td><td>0.000000</td></tr>
<tr><td>Capacity</td><td>500</td></tr>
<tr><td>Passengers</td><td>1</td></tr>
<tr><td>Numbe</td><td></td></tr>
<tr><td>Departi</td><td></td></tr>
<tr><td>Locatio</td><td></td></tr>
<tr><td>Delay</td><td></td></tr>
<tr><td>PeopleON</td><td></td></tr>
</table>

Break When Variable is Modified
View Memory at Address
View in Editor
Dump Variable Members to File
Save Snapshot of Variable
Compare Variable Snapshots
Watches
Local

</td></tr>
</table>

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.



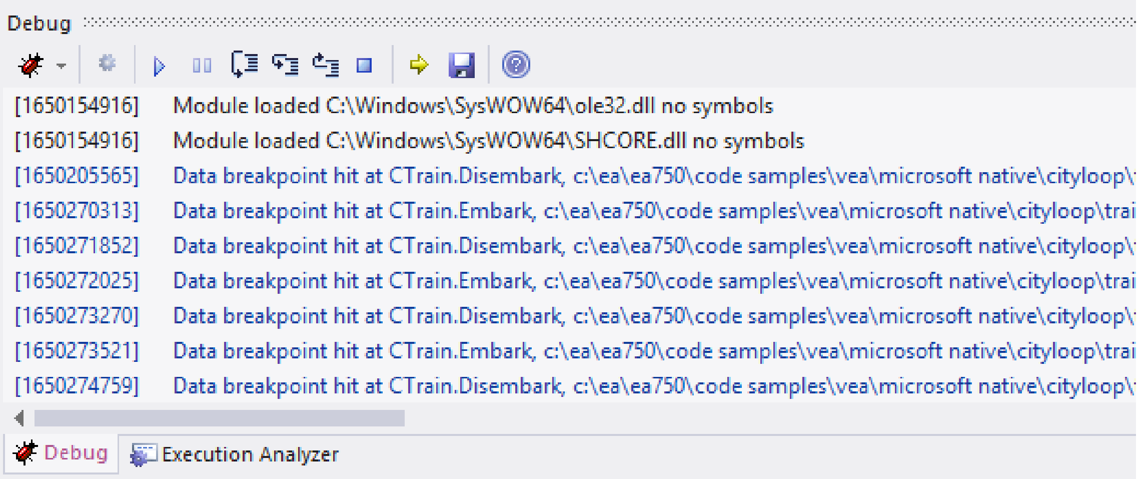
Breakpoints & Markers

8 Default

Enabled	Line	Source
● ☑	002BBEE4	int Passengers (002BBEE4)
● □	127	C:\EA\EA750\Code Samples\VEA\M

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.

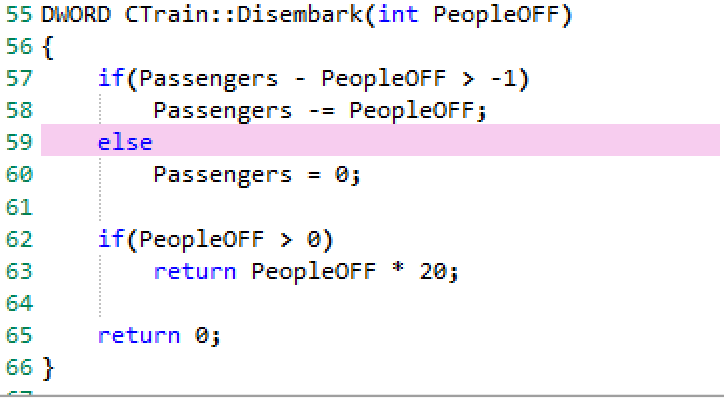
5 When this variable is modified, the debugger halts and displays the current line of code in the editor. This is not the line that caused the break, but the line of code following the event. The event is logged to the Debugger window.



Debug

```
[1650154916]   Module loaded C:\Windows\SysWOW64\ole32.dll no symbols
[1650154916]   Module loaded C:\Windows\SysWOW64\SHCORE.dll no symbols
[1650205565]   Data breakpoint hit at CTrain.Disembark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\
[1650270313]   Data breakpoint hit at CTrain.Embark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\trai
[1650271852]   Data breakpoint hit at CTrain.Disembark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\
[1650272025]   Data breakpoint hit at CTrain.Embark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\trai
[1650273270]   Data breakpoint hit at CTrain.Disembark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\
[1650273521]   Data breakpoint hit at CTrain.Embark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\trai
[1650274759]   Data breakpoint hit at CTrain.Disembark, c:\ea\ea750\code samples\vea\microsoft native\cityloop\
```

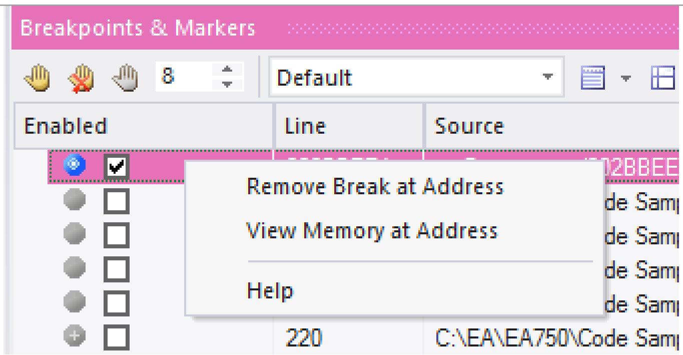
Now we know how and where this value (its State) has changed. For example, the statement at line 58 has just updated the number of Passengers.



```
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

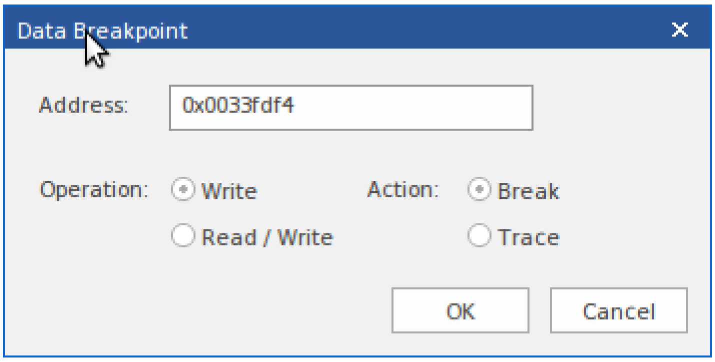
Detecting Memory Address Operations

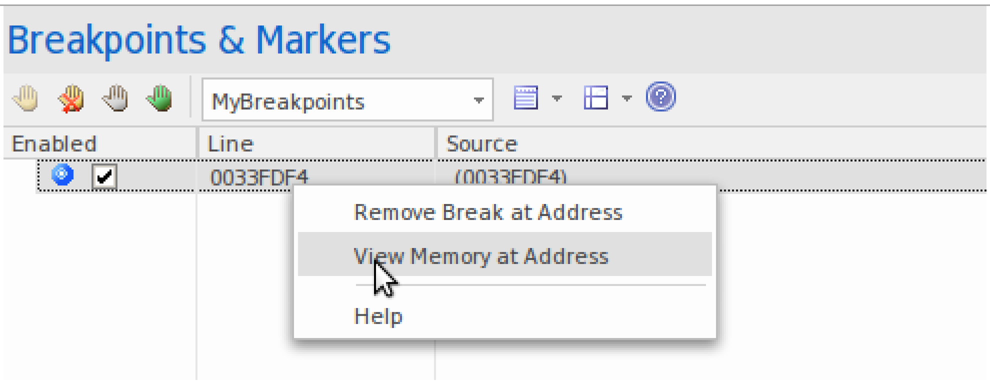
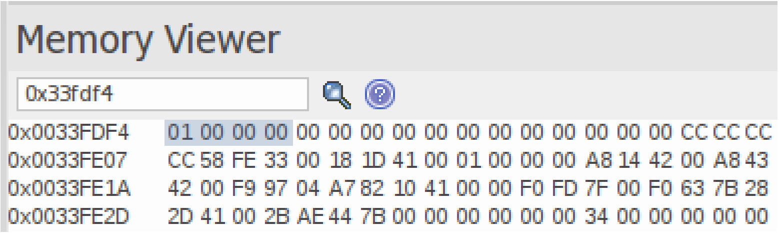
Being able to detect where and when an area of memory is being read or written can be a great help for investigators, even when the code base is well understood. Without this tool, a C++ developer could have a potentially daunting task of tracking where and when a global variable is accessed, and debugging 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.  Address: 0x0033fdf4 Operation: ● Write ○ Read / Write Action: ● Break ○ Trace
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.

	 Breakpoints & Markers MyBreakpoints Enabled Line Source 0033FDF4 (0033FDF4) Remove Break at Address View Memory at Address Help
6	You can use the context menu on the data breakpoint to check the value at the memory address.  Memory Viewer 0x33fdf4 0x0033FDF4 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 CC CC CC 0x0033FE07 CC 58 FE 33 00 18 1D 41 00 01 00 00 00 A8 14 42 00 A8 43 0x0033FE1A 42 00 F9 97 04 A7 82 10 41 00 00 F0 FD 7F 00 F0 63 7B 28 0x0033FE2D 2D 41 00 2B AE 44 7B 00 00 00 00 00 00 34 00 00 00 00 00
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

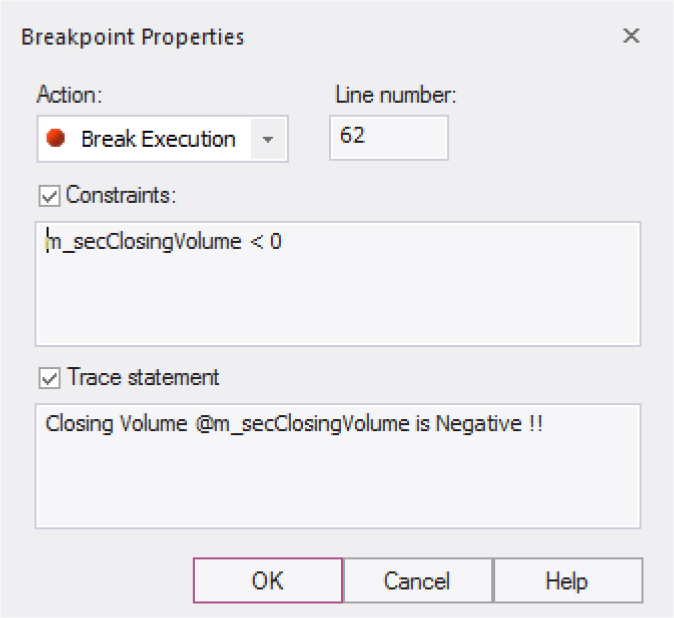
Memory address breakpoints are supported in the C/C++ native debugger.

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



Breakpoint Properties

Action: ● Break Execution

Line number: 62

☑ Constraints:

```
m_secClosingVolume < 0
```

☑ Trace statement

```
Closing Volume @m_secClosingVolume is Negative !!
```

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 or Execute > Analyze > Debugger > Attach to Process
Other	Debug window toolbar : ⚙

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 > Core > Local Variables
Context Menu	In Code Editor | Right-click on any variable identifier > Display Variable

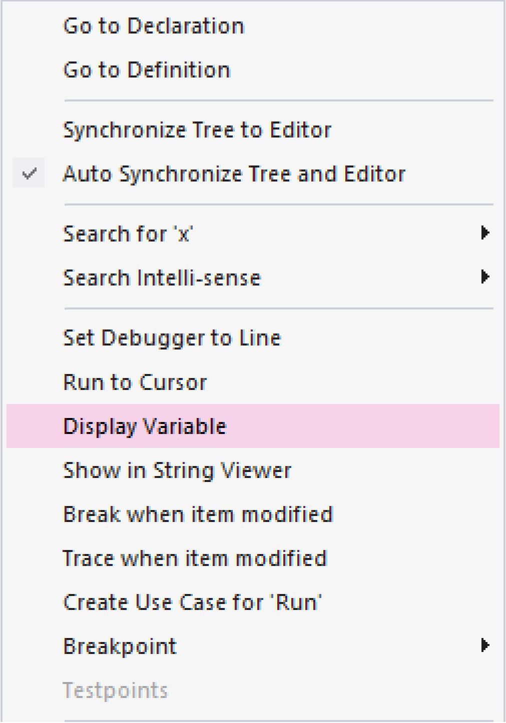
Icons

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*).

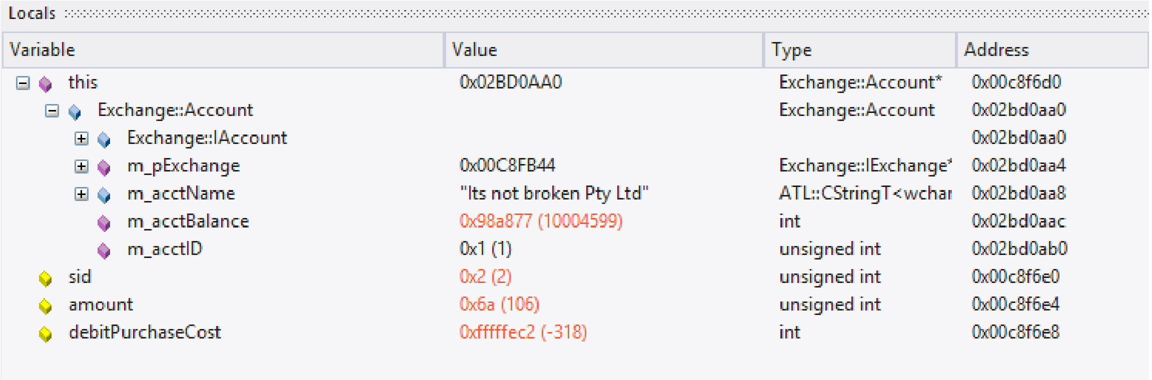


Persistent View

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.



Variable	Value	Type	Address
⊟ this	0x02BD0AA0	Exchange::Account*	0x00c8f6d0
⊟ Exchange::Account		Exchange::Account	0x02bd0aa0
⊞ Exchange::IAccount			0x02bd0aa0
⊞ m_pExchange	0x00C8FB44	Exchange::IExchange*	0x02bd0aa4
⊞ m_acctName	"Its not broken Pty Ltd"	ATL::CStringT<wchar	0x02bd0aa8
m_acctBalance	0x98a877 (10004599)	int	0x02bd0aac
m_acctID	0x1 (1)	unsigned int	0x02bd0ab0
sid	0x2 (2)	unsigned int	0x00c8f6e0
amount	0x6a (106)	unsigned int	0x00c8f6e4
debitPurchaseCost	0xfffffec2 (-318)	int	0x00c8f6e8

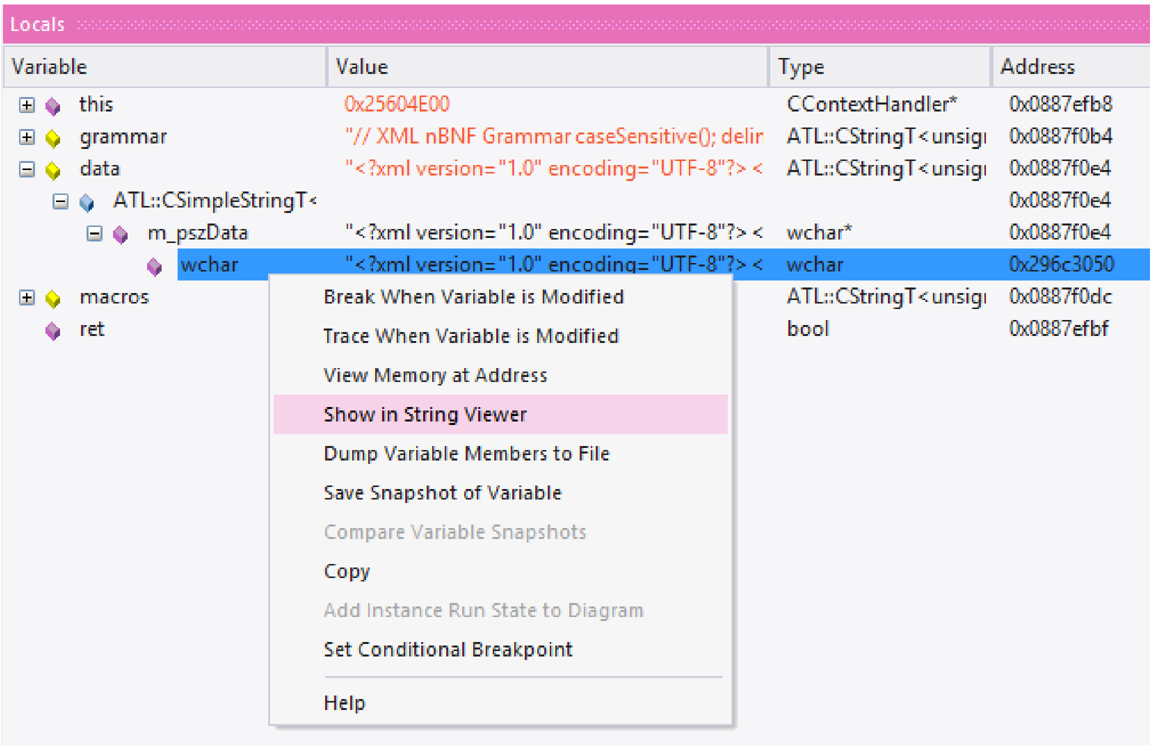
Context Menu

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'.

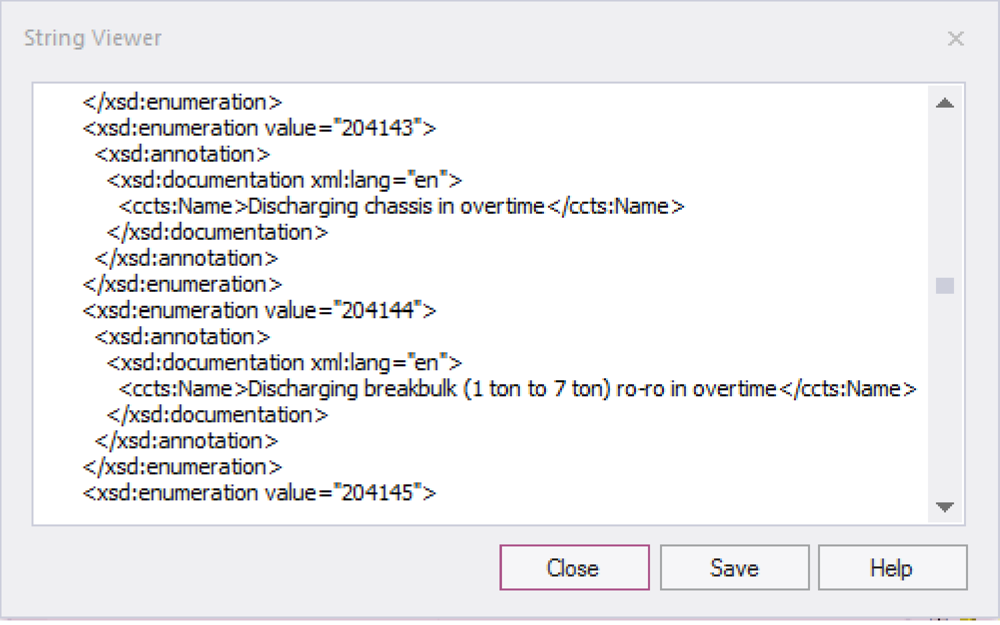


Locals

Variable	Value	Type	Address
⊞ this	0x25604E00	CContextHandler*	0x0887efb8
⊞ grammar	"// XML nBNF Grammar caseSensitive(); delir	ATL::CStringT<unsigı	0x0887f0b4
⊟ data	"<?xml version="1.0" encoding="UTF-8"?> <	ATL::CStringT<unsigı	0x0887f0e4
⊟ ATL::CSimpleStringT<			0x0887f0e4
⊟ m_pszData	"<?xml version="1.0" encoding="UTF-8"?> <	wchar*	0x0887f0e4
wchar	"<?xml version="1.0" encoding="UTF-8"?> <	wchar	0x296c3050
⊞ macros		ATL::CStringT<unsigı	0x0887f0dc
ret		bool	0x0887efbf

Break When Variable is Modified
Trace When Variable is Modified
View Memory at Address
Show in String Viewer
Dump Variable Members to File
Save Snapshot of Variable
Compare Variable Snapshots
Copy
Add Instance Run State to Diagram
Set Conditional Breakpoint
Help

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



String Viewer

```
    </xsd:enumeration>
    <xsd:enumeration value="204143">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
          <ccts:Name>Discharging chassis in overtime</ccts:Name>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:enumeration>
    <xsd:enumeration value="204144">
      <xsd:annotation>
        <xsd:documentation xml:lang="en">
          <ccts:Name>Discharging breakbulk (1 ton to 7 ton) ro-ro in overtime</ccts:Name>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:enumeration>
    <xsd:enumeration value="204145">
```

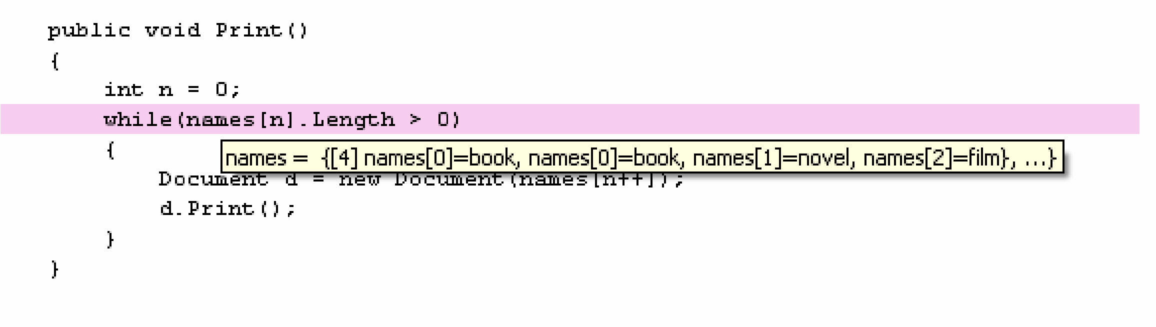
Access

From Code Editor or	

Locals window	Right-click on string variable | Show in String Viewer

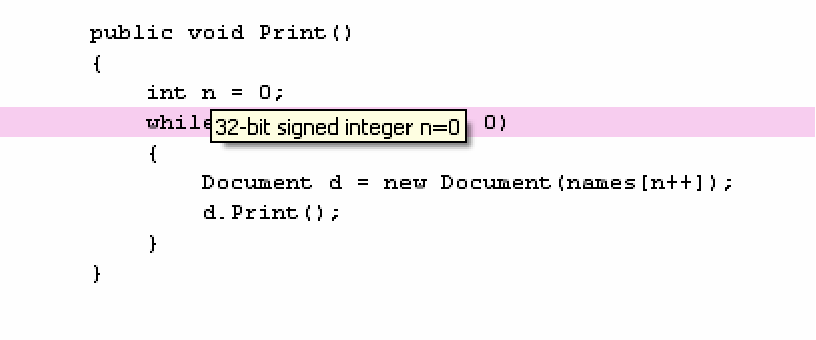
View Debug Variables in Code Editors

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.



```
public void Print()
{
    int n = 0;
    while(names[n].Length > 0)
    {
        Document d = new Document(names[n++]);
        d.Print();
    }
}
```

names = {[4] names[0]=book, names[0]=book, names[1]=novel, names[2]=film}, ...}



```
public void Print()
{
    int n = 0;
    while                    0)
    {
        Document d = new Document(names[n++]);
        d.Print();
    }
}
```

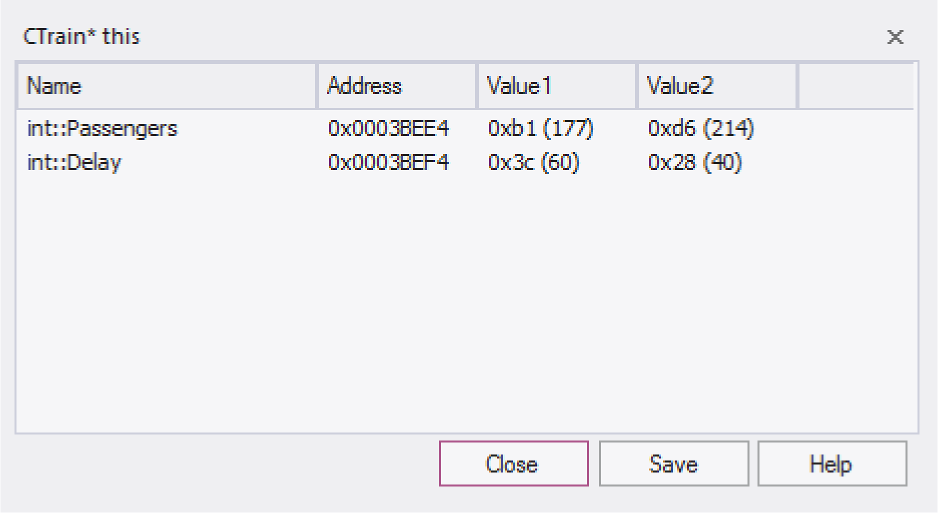
32-bit signed integer n=0

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

Variable Snapshots

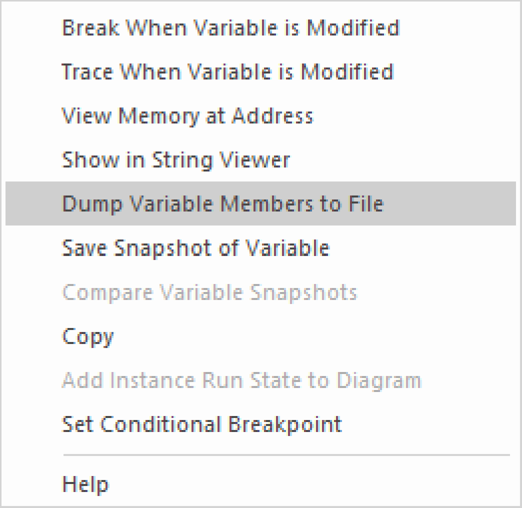
It is possible to take a 'snapshot' of a variable when your program hits a breakpoint and use this snapshot to see how the value of the variable changes at different points in its life. The debugger does not copy the value of the selected variable only; for complex variables it copies the values of the 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

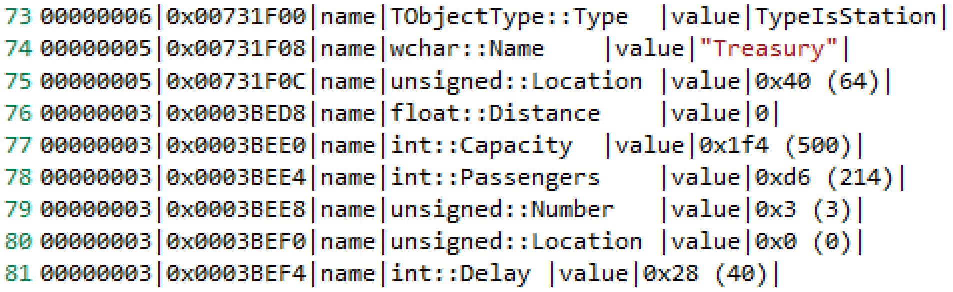
<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>In the Code Editor, set two breakpoints: one at the start of a function and another at the end of the function.</td></tr>
<tr><td>2</td><td>At the start breakpoint, right-click on a variable in the Locals window and select the 'Save Variable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

CTrain* this
<table><tr><th>Name</th><th>Address</th><th>Value1</th><th>Value2</th></tr><tr><td>int::Passengers</td><td>0x0003BEE4</td><td>0xb1 (177)</td><td>0xd6 (214)</td></tr><tr><td>int::Delay</td><td>0x0003BEF4</td><td>0x3c (60)</td><td>0x28 (40)</td></tr></table> </td></tr>
</table>

Save Variable Snapshot to File

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


Break When Variable is Modified
Trace When Variable is Modified
View Memory at Address
Show in String Viewer
Dump Variable Members to File
Save Snapshot of Variable
Compare Variable Snapshots
Copy
Add Instance Run State to Diagram
Set Conditional Breakpoint
Help

This is an excerpt of the file contents.



```
00000006|0x00731F00|name|TObjectType::Type  |value|TypeIsStation|
00000005|0x00731F08|name|wchar::Name     |value|"Treasury"|
00000005|0x00731F0C|name|unsigned::Location |value|0x40 (64)|
00000003|0x0003BED8|name|float::Distance    |value|0|
00000003|0x0003BEE0|name|int::Capacity   |value|0x1f4 (500)|
00000003|0x0003BEE4|name|int::Passengers    |value|0xd6 (214)|
00000003|0x0003BEE8|name|unsigned::Number    |value|0x3 (3)|
00000003|0x0003BEF0|name|unsigned::Location |value|0x0 (0)|
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	/ - division

conditional expressions	+ - 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.

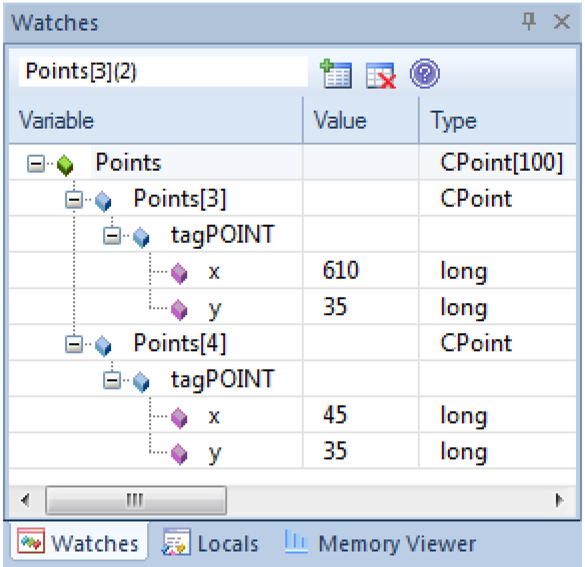
View Elements of Array

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.



Points[3](2)

Variable	Value	Type
Points		CPoint[100]
Points[3]		CPoint
tagPOINT		
x	610	long
y	35	long
Points[4]		CPoint
tagPOINT		
x	45	long
y	35	long

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 Sequence Diagram from the Call Stack	To generate Sequence diagram from the 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

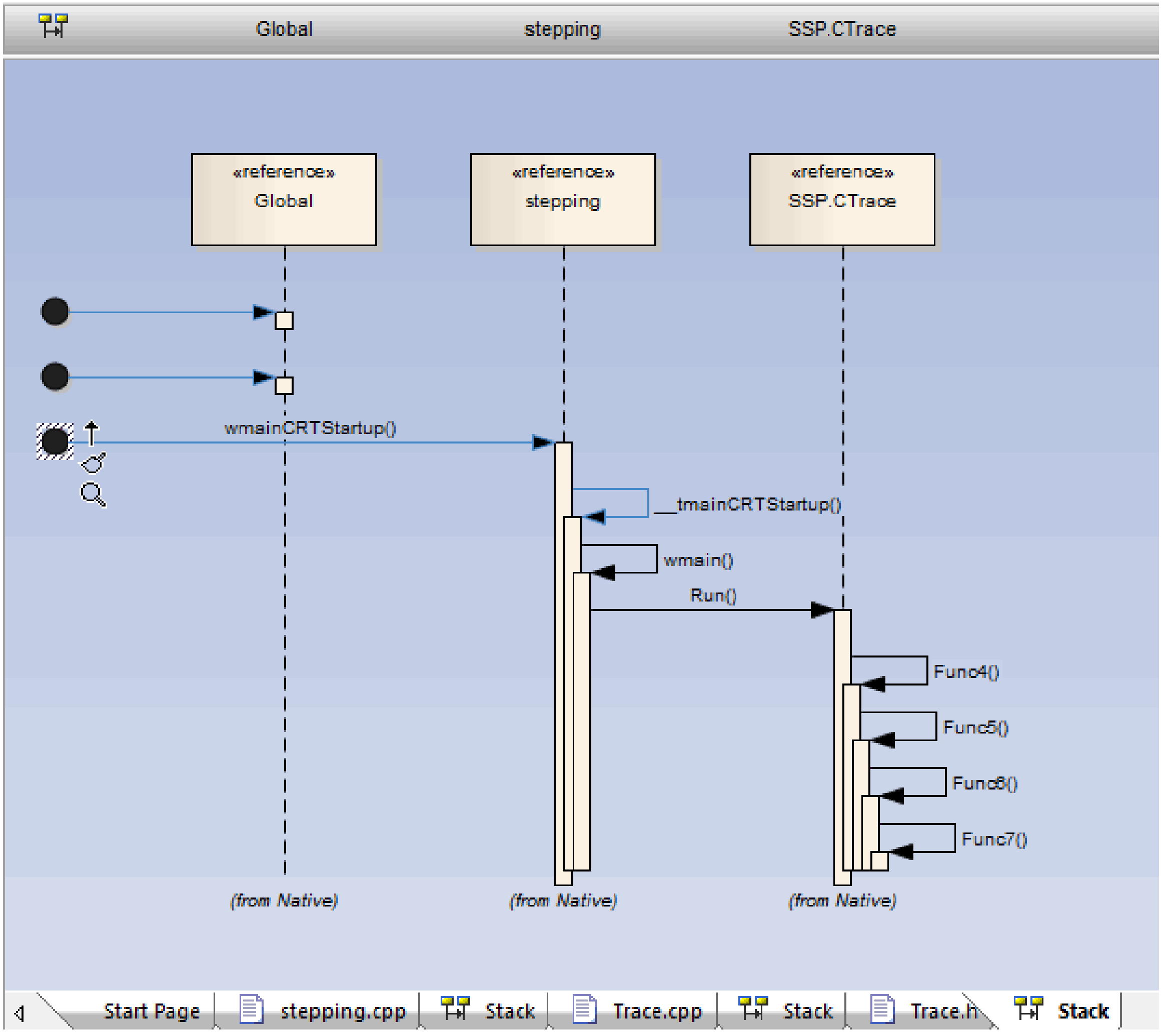
Ribbon	Execute > Windows > Call Stack
Other	Execution Analyzer window toolbar : | Call Stack

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 > Analyze > 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 Debuger

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 > Analyze > 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 > Run > Services Code > Configure > 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

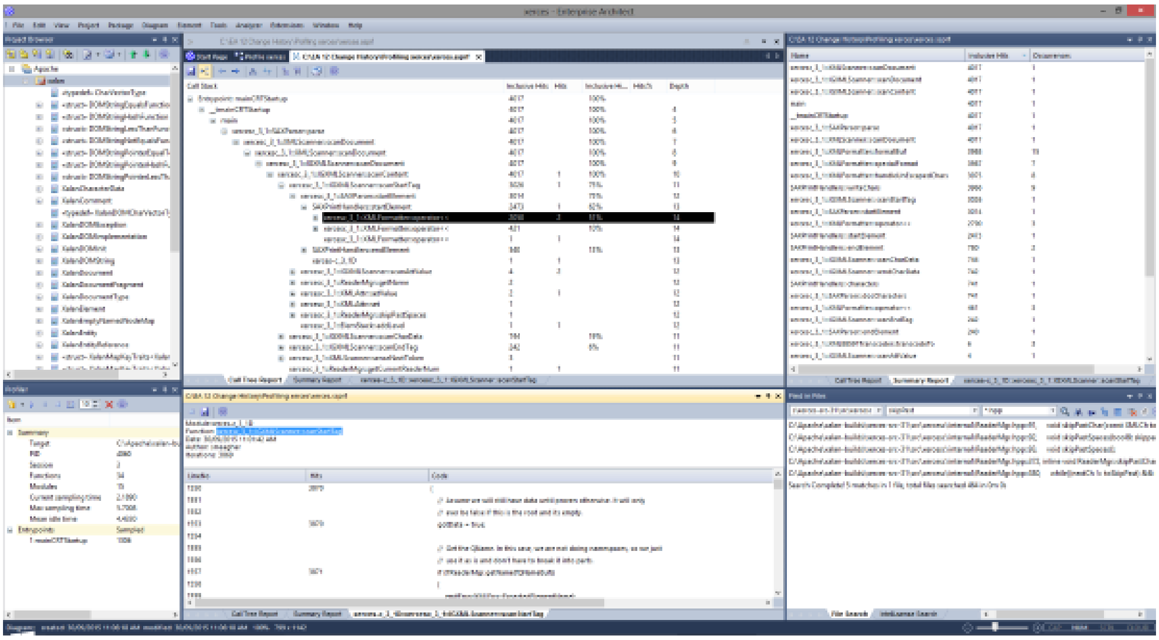
The Agent Service

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.

Profiling

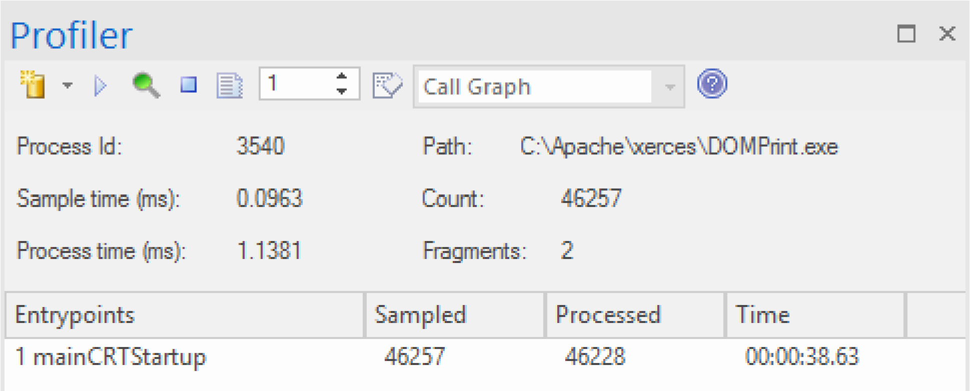


During the lifetime of software applications, it is not uncommon to investigate application tasks that are determined to be performing slower than expected. You might also simply want to know what is going on when you '*press this button*'! You can work this out quite quickly in Enterprise Architect by using its Profiler. Results can usually be produced in a few seconds and you will quickly be able to see the actions that are consuming the application and the functions involved. In the Execution Analyzer, The feature employs two separate strategies; *Process Sampling* and *Process Hooking*. In one, samples are taken at regular intervals to identify CPU-intensive patterns, while in the other, the process is hooked to record demands made on memory. Data is analyzed to produce a weighted Call Graph. Behaviors are usually identifiable as root nodes (entrypoints) in the graph, or branches near these points. All reports can be reviewed on demand. They can be saved to file within the model, both as Artifact elements and as Team Library posts.

Access

Ribbon	Execute > Analyze > Profiler
Other	Execution Analyzer toolbar : Analyzer Windows | Profiler

Call Sampling



Process Id: 3540 Path: C:\Apache\xerces\DOMPrint.exe

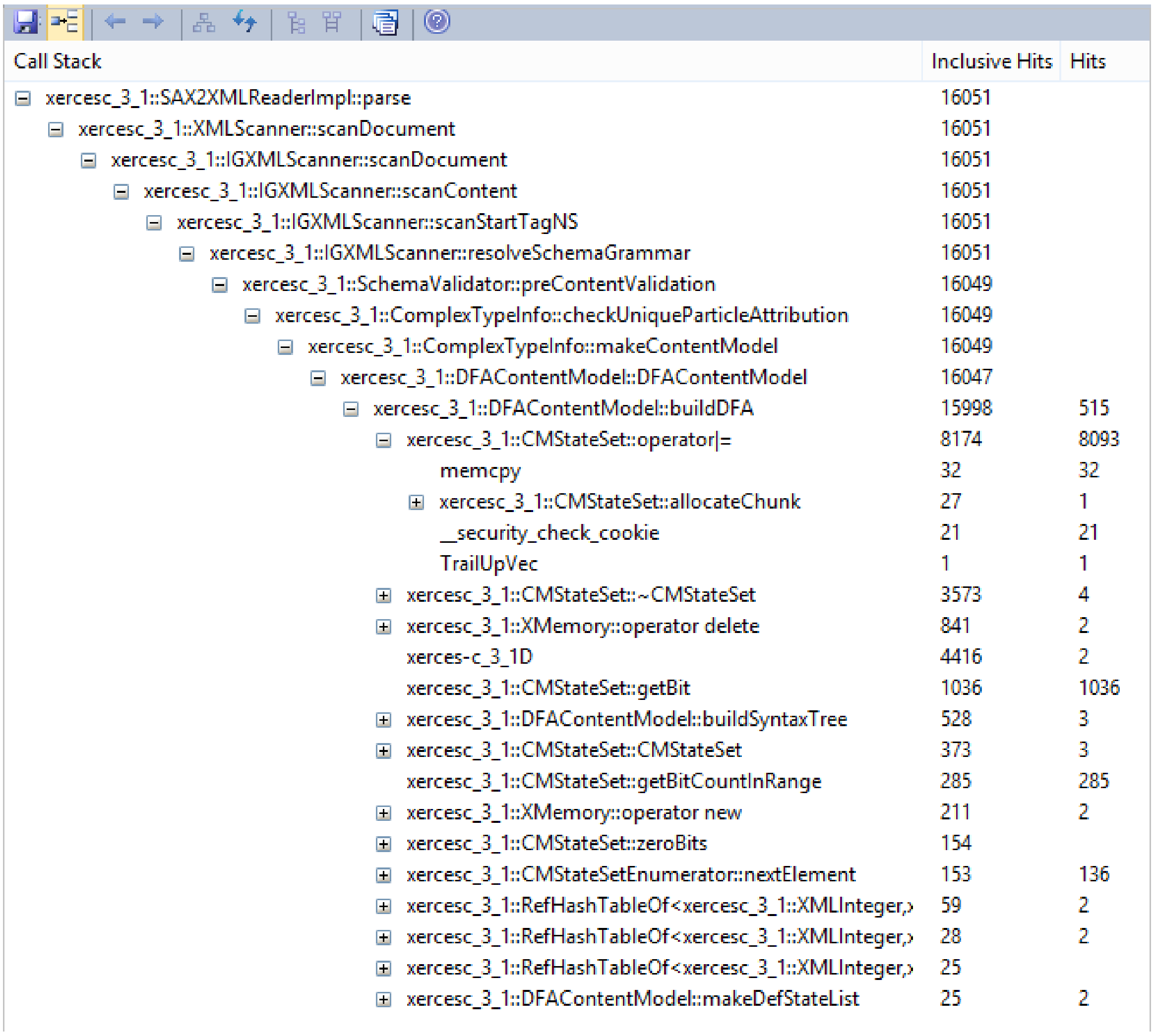
Sample time (ms): 0.0963 Count: 46257

Process time (ms): 1.1381 Fragments: 2

Entrypoints	Sampled	Processed	Time
1 mainCRTStartup	46257	46228	00:00:38.63

The Profiler is controlled using its toolbar buttons. Here you can attach the Profiler to an existing process (or JVM), or launch the application for the active Analyzer Script. The Profiler window displays the details of the target process as it is profiled. These details provide feedback, letting you see the number of samples taken. You also have options for pausing and resuming capture, clearing captured data and generating reports. You can gain access to the reporting feature by pausing the capture - the reporting feature is disabled whilst data capture is in progress.

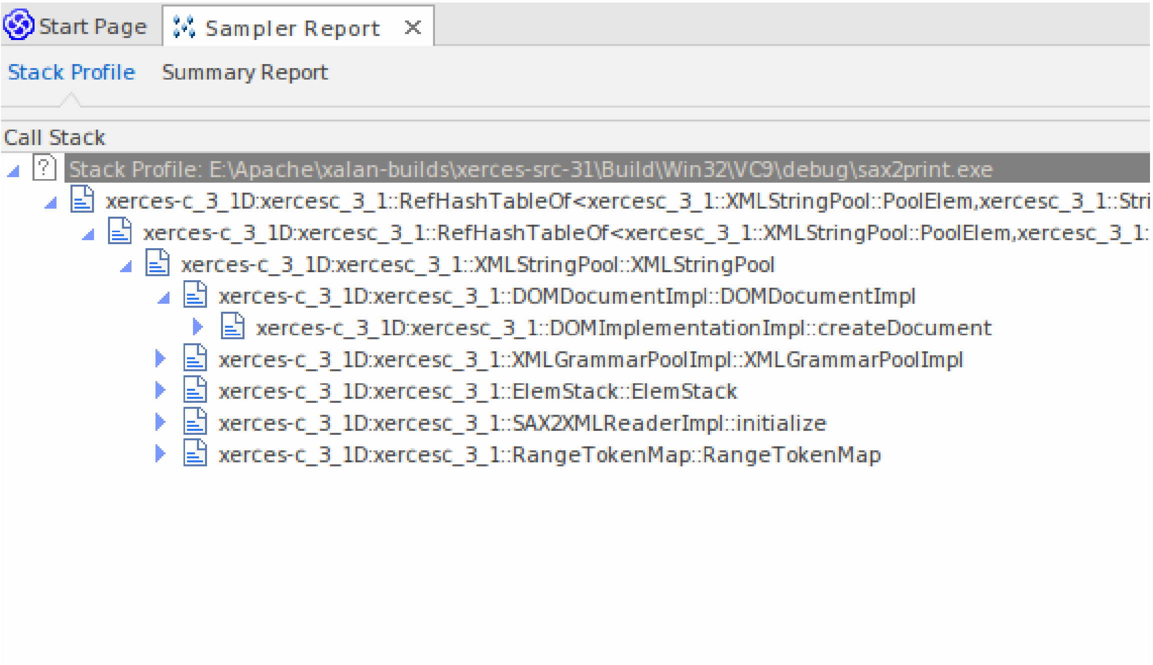
Weighted Call Graph



Call Stack	Inclusive Hits	Hits
⊟ xercesc_3_1::SAX2XMLReaderImpl::parse	16051	
⊟ xercesc_3_1::XMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanContent	16051	
⊟ xercesc_3_1::IGXMLScanner::scanStartTagNS	16051	
⊟ xercesc_3_1::IGXMLScanner::resolveSchemaGrammar	16051	
⊟ xercesc_3_1::SchemaValidator::preContentValidation	16049	
⊟ xercesc_3_1::ComplexTypeInfo::checkUniqueParticleAttribution	16049	
⊟ xercesc_3_1::ComplexTypeInfo::makeContentModel	16049	
⊟ xercesc_3_1::DFAContentModel::DFAContentModel	16047	
⊟ xercesc_3_1::DFAContentModel::buildDFA	15998	515
⊟ xercesc_3_1::CMStateSet::operator|=	8174	8093
memcpy	32	32
⊞ xercesc_3_1::CMStateSet::allocateChunk	27	1
__security_check_cookie	21	21
TrailUpVec	1	1
⊞ xercesc_3_1::CMStateSet::~CMStateSet	3573	4
⊞ xercesc_3_1::XMemory::operator delete	841	2
xerces-c_3_1D	4416	2
xercesc_3_1::CMStateSet::getBit	1036	1036
⊞ xercesc_3_1::DFAContentModel::buildSyntaxTree	528	3
⊞ xercesc_3_1::CMStateSet::CMStateSet	373	3
xercesc_3_1::CMStateSet::getBitCountInRange	285	285
⊞ xercesc_3_1::XMemory::operator new	211	2
⊞ xercesc_3_1::CMStateSet::zeroBits	154	
⊞ xercesc_3_1::CMStateSetEnumerator::nextElement	153	136
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	59	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	28	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	25	
⊞ xercesc_3_1::DFAContentModel::makeDefStateList	25	2

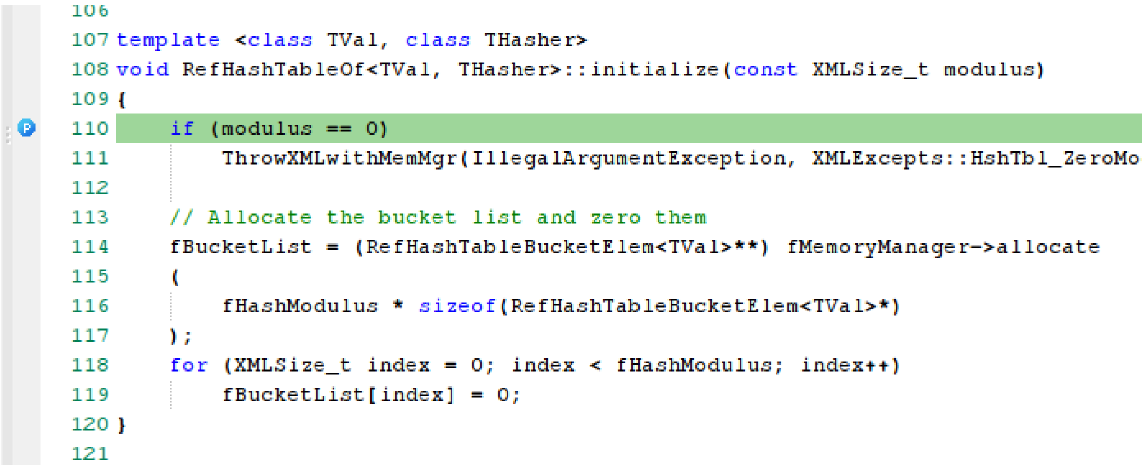
This detailed report shows the unique set of Call Stacks/behaviors as a weighted Call Graph. The weight of each branch is depicted by a hit count, which is the total hits of that branch plus all branches from this point. By following the hit trail, a user can quickly identify the areas of code that occupied the program the most during the capture period.

Stack Profile



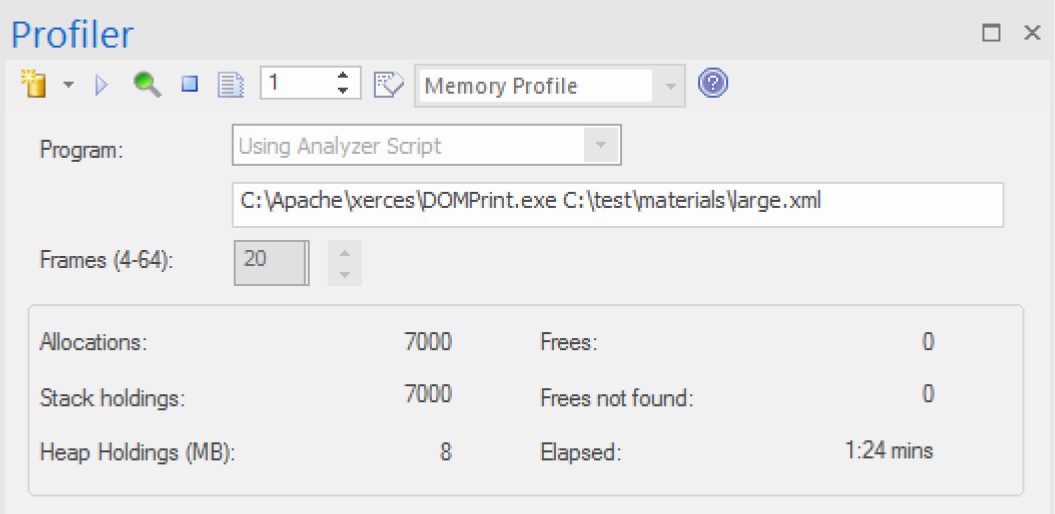
```
Stack Profile: E:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\sax2print.exe
  xerces-c_3_1D:xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLStringPool::PoolElem,xercesc_3_1::Stri
    xerces-c_3_1D:xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLStringPool::PoolElem,xercesc_3_1:
      xerces-c_3_1D:xercesc_3_1::XMLStringPool::XMLStringPool
        xerces-c_3_1D:xercesc_3_1::DOMDocumentImpl::DOMDocumentImpl
          xerces-c_3_1D:xercesc_3_1::DOMImplementationImpl::createDocument
        xerces-c_3_1D:xercesc_3_1::XMLGrammarPoolImpl::XMLGrammarPoolImpl
        xerces-c_3_1D:xercesc_3_1::ElemStack::ElemStack
        xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::initialize
        xerces-c_3_1D:xercesc_3_1::RangeTokenMap::RangeTokenMap
```

Stack Profiles are a taken to discover the different ways (stacks) and the count of ways that a particular function is invoked during the running of the program. Unlike other the other profiler modes, this profile is activated through the use of a Profile Point, which is a special kind of breakpoint marker. The marker is set in the source code like any other breakpoint. When the breakpoint is encountered by the program, the stack is captured. When you later produce the report, the stacks are analyzed and a weighted call graph produced. The graph shows the unique stacks that were involved in that function during the time the profiler was running, The 'Hit Count' column indicates the count of times that same stack occurred.



```

template <class TVal, class THasher>
void RefHashTableOf<TVal, THasher>::initialize(const XMLSize_t modulus)
{
    if (modulus == 0)
        ThrowXMLwithMemMgr(IllegalArgumentException, XMLExcepts::HshTbl_ZeroMo

    // Allocate the bucket list and zero them
    fBucketList = (RefHashTableBucketElem<TVal>**) fMemoryManager->allocate
    (
        fHashModulus * sizeof(RefHashTableBucketElem<TVal>*)
    );
    for (XMLSize_t index = 0; index < fHashModulus; index++)
        fBucketList[index] = 0;
}

```

Memory Profiles



Program: Using Analyzer Script

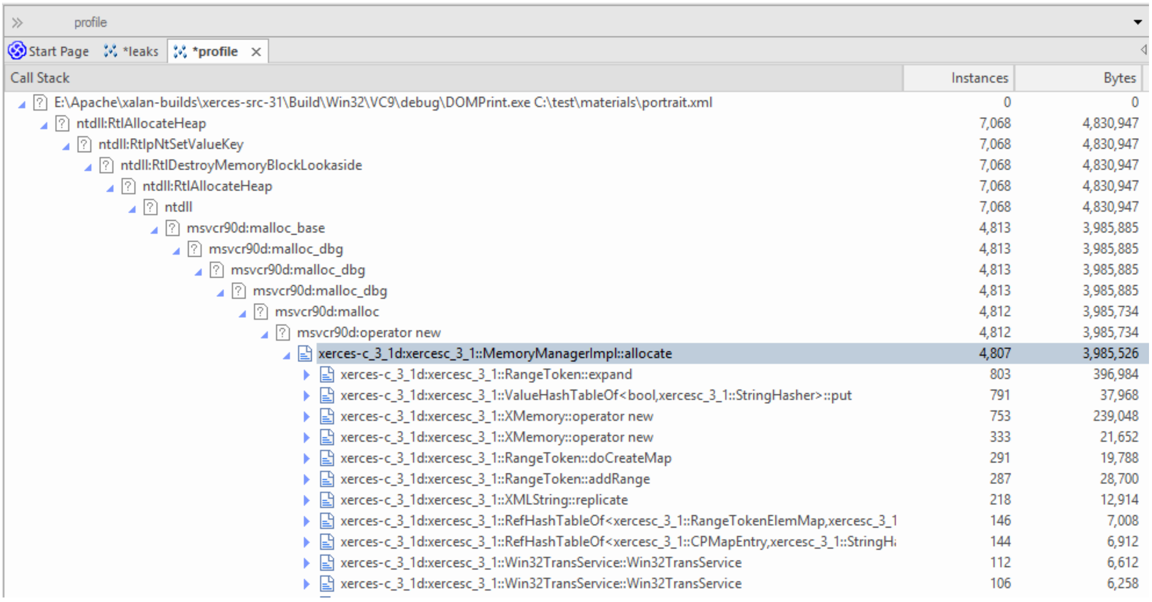
C:\Apache\xerces\DOMPrint.exe C:\test\materials\large.xml

Frames (4-64): 20

Allocations:	7000	Frees:	0
Stack holdings:	7000	Frees not found:	0
Heap Holdings (MB):	8	Elapsed:	1:24 mins

The Memory profile tracks allocations, ignoring when memory is freed. It uses this information to rate the executing code's demands for memory, in terms not of the amount of memory but of the frequency of demands. The *Allocations* figure is the total number of memory allocations requested. The *Stack Holdings* is the number of stack traces taken at those times, and the *Heap Holding* figure is the total amount of memory obtained by these calls. Note that profiling can be turned on and off on demand. There is also no need to rebuild your program to get it to work as there is no linkage involved.

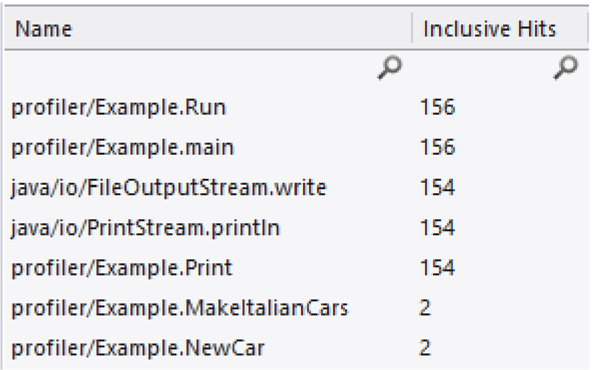
Memory Graph



Call Stack	Instances	Bytes
E:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\DOMPrint.exe C:\test\materials\portrait.xml	0	0
ntdll:RtlAllocateHeap	7,068	4,830,947
ntdll:RtlpNtSetValueKey	7,068	4,830,947
ntdll:RtlDestroyMemoryBlockLookaside	7,068	4,830,947
ntdll:RtlAllocateHeap	7,068	4,830,947
ntdll	7,068	4,830,947
msvcr90d:malloc_base	4,813	3,985,885
msvcr90d:malloc_dbg	4,813	3,985,885
msvcr90d:malloc_dbg	4,813	3,985,885
msvcr90d:malloc_dbg	4,813	3,985,885
msvcr90d:malloc	4,812	3,985,734
msvcr90d:operator new	4,812	3,985,734
xerces-c_3_1d:xercesc_3_1::MemoryManagerImpl::allocate	4,807	3,985,526
xerces-c_3_1d:xercesc_3_1::RangeToken::expand	803	396,984
xerces-c_3_1d:xercesc_3_1::ValueHashTableOf<bool,xercesc_3_1::StringHasher>::put	791	37,968
xerces-c_3_1d:xercesc_3_1::XMemory::operator new	753	239,048
xerces-c_3_1d:xercesc_3_1::XMemory::operator new	333	21,652
xerces-c_3_1d:xercesc_3_1::RangeToken::doCreateMap	291	19,788
xerces-c_3_1d:xercesc_3_1::RangeToken::addRange	287	28,700
xerces-c_3_1d:xercesc_3_1::XMLString::replicate	218	12,914
xerces-c_3_1d:xercesc_3_1::RefHashTableOf<xercesc_3_1::RangeTokenElemMap,xercesc_3_1	146	7,008
xerces-c_3_1d:xercesc_3_1::RefHashTableOf<xercesc_3_1::CPMapEntry,xercesc_3_1::StringH	144	6,912
xerces-c_3_1d:xercesc_3_1::Win32TransService::Win32TransService	112	6,612
xerces-c_3_1d:xercesc_3_1::Win32TransService::Win32TransService	106	6,258

This example is of a report produced from Profiling a demonstration program in the Xerces project from Apache. The program iterates over the Document Object Model (DOM) for a provided XML file.

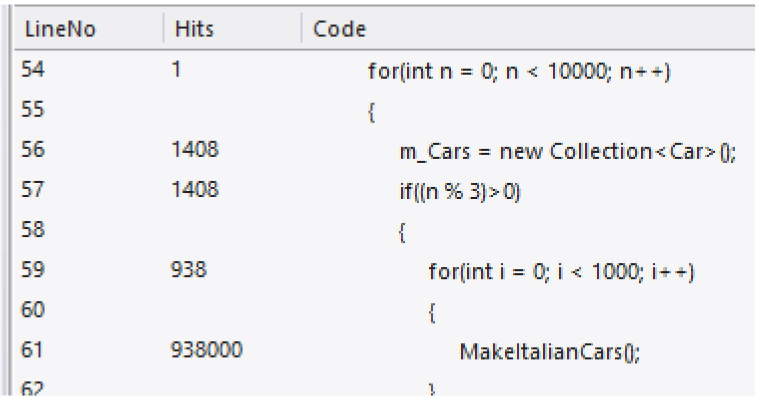
Function Summary Report



Name	Inclusive Hits
profiler/Example.Run	156
profiler/Example.main	156
java/io/FileOutputStream.write	154
java/io/PrintStream.println	154
profiler/Example.Print	154
profiler/Example.MakeItalianCars	2
profiler/Example.NewCar	2

This summary report lists the functions and only those functions executed during the sample period. Functions are listed by total invocations, with a function that presents twice in separate Call Stacks appearing before a function that appears just the once.

Function Line Report



LineNo	Hits	Code
54	1	for(int n = 0; n < 10000; n++)
55		{
56	1408	m_Cars = new Collection<Car>();
57	1408	if((n % 3)>0)
58		{
59	938	for(int i = 0; i < 1000; i++)
60		{
61	938000	MakeItalianCars();
62		}

This detailed report shows the source code for a function line by line displaying beside it the total times each was executed. We uncovered code using this report, that exposed case statements in code that never appeared to be executed.

Support

The Profiler is supported for programs written in C, C++, Visual Basic, Java and the Microsoft .NET languages. Memory profiling is currently available for native C and C++ programs.

Notes

- The Profiler is available in Enterprise Architect Professional editions and above
- The Profiler can also be used under WINE (Linux and Mac) for Profiling standard Windows applications deployed in a WINE environment

System Requirements

Using the Profiler, you can analyze applications built for these platforms:

- Microsoft ™ Native (C++, C, Visual basic)
- Microsoft .NET (supporting a mix of managed and unmanaged code)
- Java

Microsoft Native applications

For C, C++ or Visual Basic applications, the Profiler requires that the applications are compiled with the Microsoft ™ Native compiler and that for each application or module of interest, a PDB file is available. The Profiler can sample both debug and release configurations of an application, provided that the PDB file for each executable exists and is up to date.

Microsoft .NET applications

For Microsoft .NET applications, the Profiler requires that the appropriate Microsoft .NET framework is installed, and that for each application or module to be analyzed, a PDB file is available.

Java

For Java, the Profiler requires that the appropriate JDK from Oracle is installed.

The classes of interest should also have been compiled with debug information. For example: "java -g *.java"

- New instance of application VM is launched from Enterprise Architect - no other action is required
- Existing application VM is attached to from within Enterprise Architect - the target Java Virtual Machine has to have been launched with the Enterprise Architect profiling agent

These are examples of command lines to create a Java VM with a specific JVMTI agent:

1. java.exe -cp "%classpath%;.\" -agentpath:"C:\Program Files (x86)\Sparx Systems\EA\vea\x86\ssamplerlib32" myapp
2. java.exe -cp "%classpath%;.\" -agentpath:"C:\Program Files (x86)\Sparx Systems\EA\vea\x64\ssamplerlib64" myapp

(Refer to the JDK documentation for details of the -agentpath VM startup option.)

Getting Started

The Profiler can be used to investigate performance issues, providing three separate tools for you to choose from, namely:

- Call Graph
- Memory Profile
- Memory Leaks

You select the tools from the Profiler toolbar.

Tools

Tool	Description
Call Graph	Analyzes performance by taking samples during an activity in a program. Each sample represents a stack. The samples are taken at intervals controlled using the toolbar. In this scenario, poor performance is rated by the patterns of behavior that repeat the most during the sample time period. This figure is used to weight the Call Graph produced.
Memory Profile	Analyzes performance by hooking the memory allocations made by a program. In this scenario, poor performance is rated by the activities making the most requests for memory. This figure is used to weight the Call Graph produced.
Memory Leaks	Analyzes memory leaks by hooking the memory operations performed by a program. What is produced is a Call Graph presenting the Call Stacks that allocated memory for which a free operation was not detected.

Access

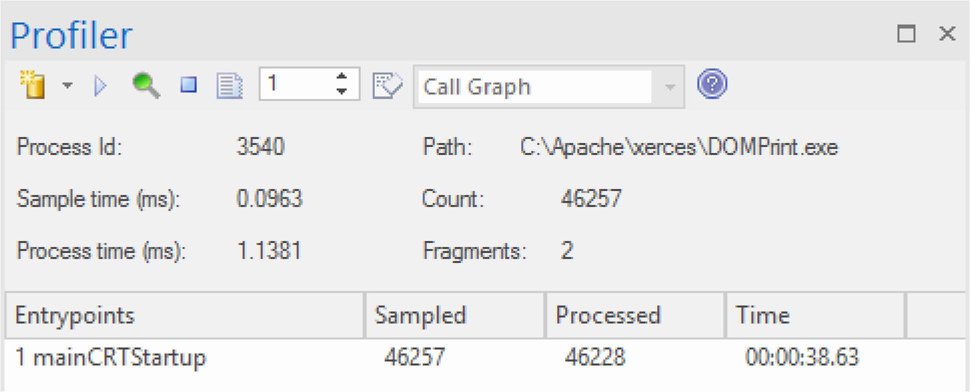
Ribbon	Code > Configure > Analyzer > Open Profiler Execute > Analyze > Profiler > Open Profiler
Other	Execution Analyzer toolbar : Analyzer Windows | Profiler

Toolbar Buttons

Button	Action
▾	Profiler options
▷	Launches the configured application to be profiled. By default, this is the application configured in the active Analyzer Script.

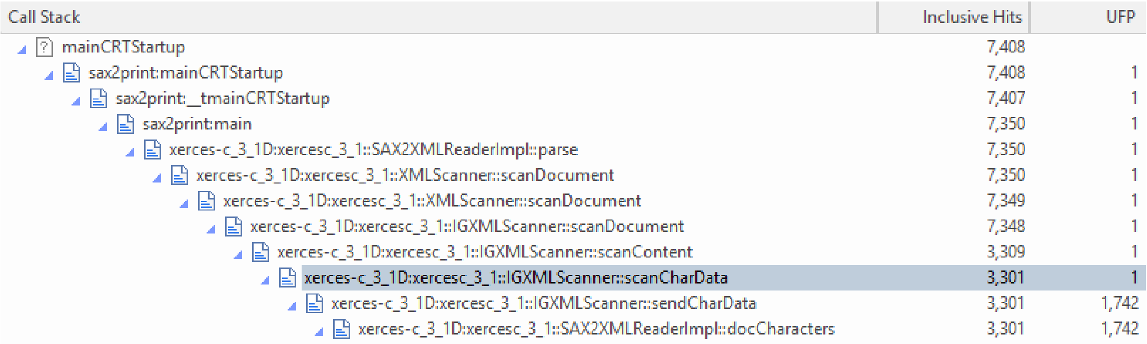
⏸ ⏸	While the Profiler is running, pauses and resumes capture. When capture is paused, the Report button and Discard Data button are active.
■	Stops the Profiler process; if any samples have been collected, the Report button is enabled.
🗎	Generates a report from the current data collection.
5 ▲▼	For the call sampler, sets the interval, in milliseconds, at which samples are taken of the target process; the range of possible values is 1 - 250. (Sampler strategy only.)
✕	Discards the collected data. You are prompted to confirm the discard.
?	Displays the Help topic for this window.

Call Graph



Process Id: 3540
Path: C:\Apache\xerces\DOMPrint.exe
Sample time (ms): 0.0963
Count: 46257
Process time (ms): 1.1381
Fragments: 2

Entrypoints	Sampled	Processed	Time
1 mainCRTStartup	46257	46228	00:00:38.63



Call Stack	Inclusive Hits	UFP
mainCRTStartup	7,408	
sax2print:mainCRTStartup	7,408	1
sax2print:__tmainCRTStartup	7,407	1
sax2print:main	7,350	1
xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::parse	7,350	1
xerces-c_3_1D:xercesc_3_1::XMLScanner::scanDocument	7,350	1
xerces-c_3_1D:xercesc_3_1::XMLScanner::scanDocument	7,349	1
xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanDocument	7,348	1
xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanContent	3,309	1
xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanCharData	3,301	1
xerces-c_3_1D:xercesc_3_1::IGXMLScanner::sendCharData	3,301	1,742
xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::docCharacters	3,301	1,742

- Quickly discover what a program is doing at any point in time
- Easily identify performance issues
- Be surprised how quickly you can realize improvements
- See your improvements at work and have the evidence
- Support for C/C++, .NET and Java platforms

Usage

The 'Call Graph' option is typically used in situations where an activity is performing slower than expected, but it can also be used simply to better understand the patterns of behavior at play during an activity.

Operation

The Profiler operates by taking samples - or Call Stacks - at regular intervals over a period of time; the interval is set using the Profiler toolbar. You use the Profiler to run a particular program, or you can attach to an existing process. The Profiler capture is controlled, and you can pause and resume capture at any time. You can also elect to have capture initiated immediately when the Profiler is started. If necessary, you can discard any captured samples and start again during the same session. If you cannot continue with the same session, restarting the Profiler is quick and easy.

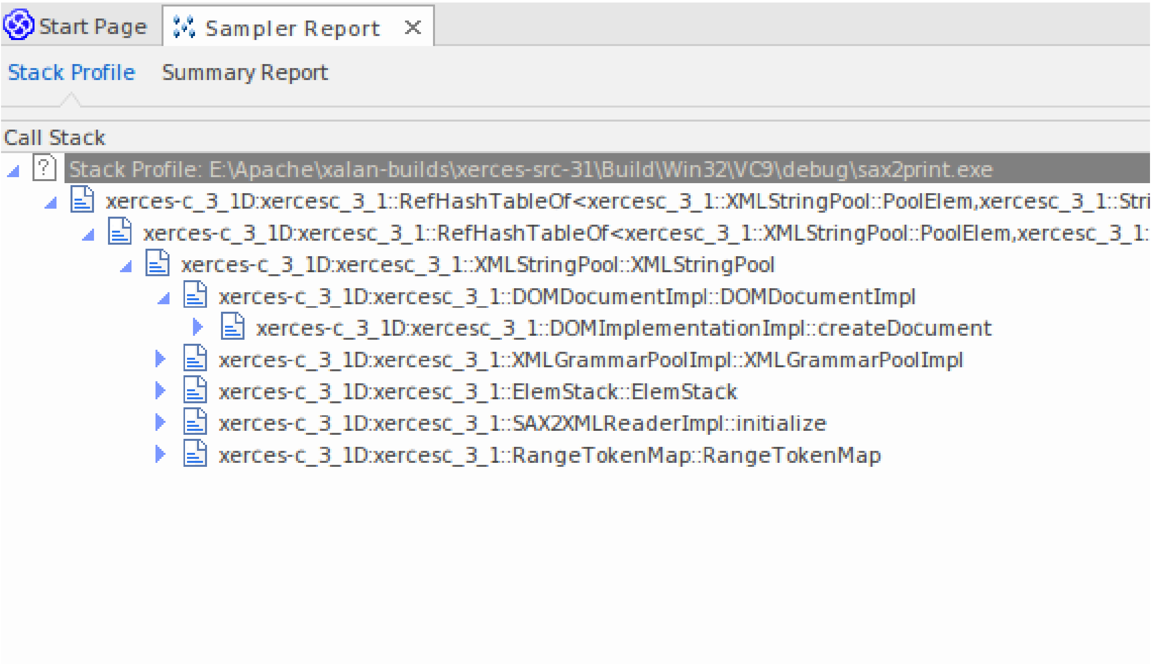
Results

Results can be produced at any time during the session; however, capture must be disabled in order for the Report button to become active. It is up to you to decide how long you let the Profiler run. You might know when an activity is finished, or it might be apparent for other reasons. The reason you are here might be that an activity is not completing at all.

You enable the Report button by either pausing capture or stopping the Profiler altogether.

Results are displayed in a Report view. The report opens with two tabs initially visible: the Call Graph and the Function Summary. The reports can be saved to file, stored in the model as Artifacts or posted in the Team Library.

Stack Profile

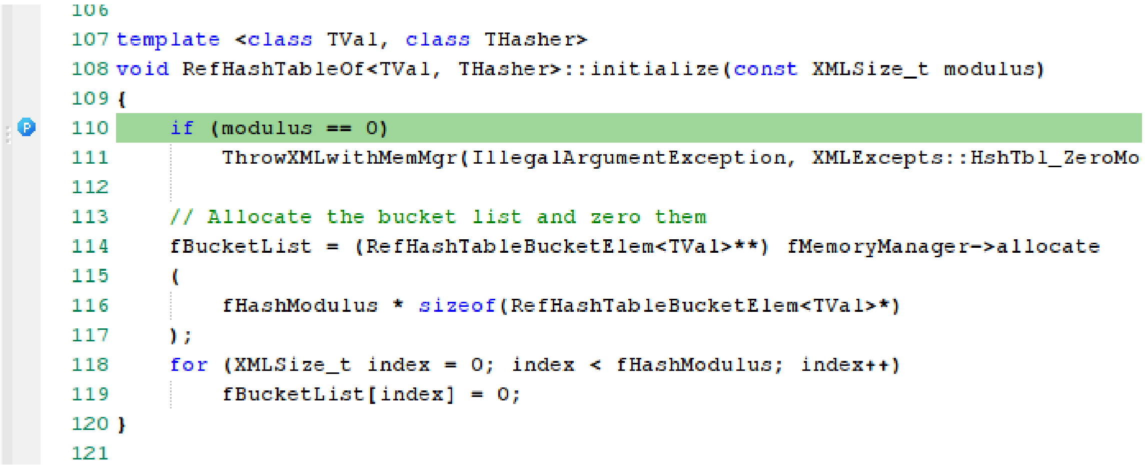


```
Stack Profile: E:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\sax2print.exe
  xerces-c_3_1D:xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLStringPool::PoolElem,xercesc_3_1::Stri
    xerces-c_3_1D:xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLStringPool::PoolElem,xercesc_3_1:
      xerces-c_3_1D:xercesc_3_1::XMLStringPool::XMLStringPool
        xerces-c_3_1D:xercesc_3_1::DOMDocumentImpl::DOMDocumentImpl
          xerces-c_3_1D:xercesc_3_1::DOMImplementationImpl::createDocument
        xerces-c_3_1D:xercesc_3_1::XMLGrammarPoolImpl::XMLGrammarPoolImpl
        xerces-c_3_1D:xercesc_3_1::ElemStack::ElemStack
        xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::initialize
        xerces-c_3_1D:xercesc_3_1::RangeTokenMap::RangeTokenMap
```

Usage

Use the Stack Profile mode to produce a report that shows the unique ways in which a function can be invoked during the running of a program. Determine the parts of the model that rely on this function and their frequency.

Operation



```
template <class TVal, class THasher>
void RefHashTableOf<TVal, THasher>::initialize(const XMLSize_t modulus)
{
    if (modulus == 0)
        ThrowXMLwithMemMgr(IllegalArgumentException, XMLExcepts::HshTbl_ZeroMo

    // Allocate the bucket list and zero them
    fBucketList = (RefHashTableBucketElem<TVal>**) fMemoryManager->allocate
    (
        fHashModulus * sizeof(RefHashTableBucketElem<TVal>*)
    );
    for (XMLSize_t index = 0; index < fHashModulus; index++)
        fBucketList[index] = 0;
}
```

Profiler modes are selected using the Profiler control Toolbar. If a Profiler Point is already created, it is displayed. The Profiler Point is the point at which stack traces are captured. You can set the Profiler Point using the Set button on the control itself, once the mode is selected. After deciding on the Profile Point, build the project to be sure everything is up to date, then start the Profiler. The number of unique stack holdings detected is visible during the run.

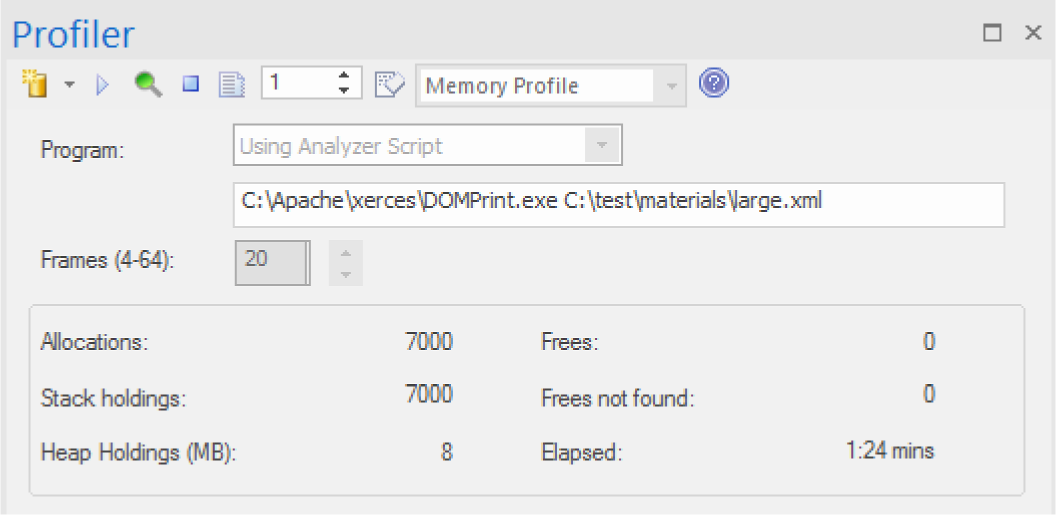
Results

A results can be produced by clicking the report button on the Profiler control Toolbar. This button is enabled when either:

1. Capture is turned off. (using the Pause Button) or
2. The Profiler is stopped. (using the Stop Button)

The results produced are displayed as a weighted call graph, where the lines on the graph represent a unique stack, and weighted to show the higher frequency stacks first. The report can then be saved, either to file or to the model, using the context menu of the report itself.

Memory Profile



Profiler

Program: Using Analyzer Script

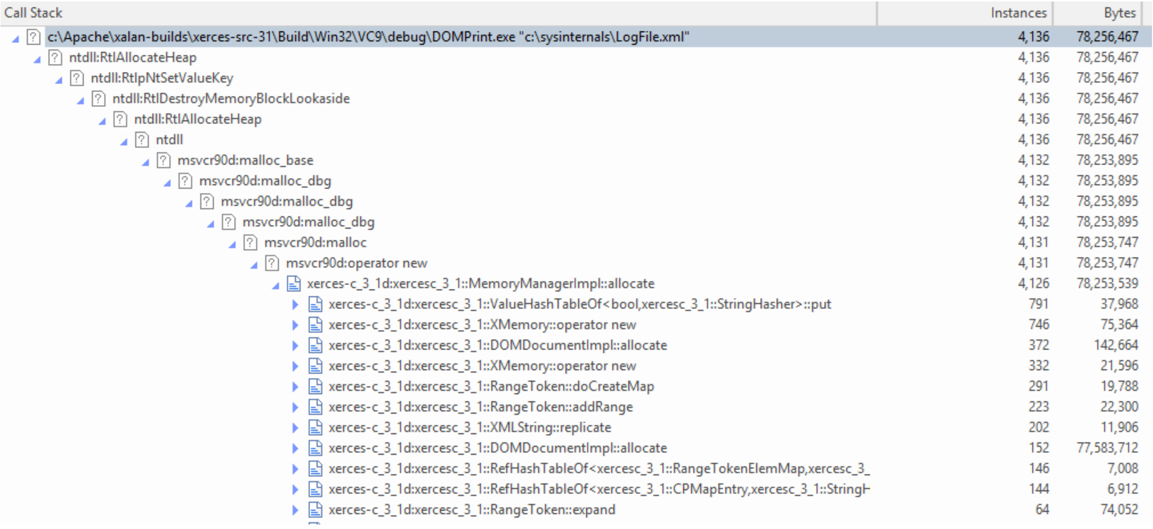
C:\Apache\xerces\DOMPrint.exe C:\test\materials\large.xml

Frames (4-64): 20

Allocations: 7000 Frees: 0

Stack holdings: 7000 Frees not found: 0

Heap Holdings (MB): 8 Elapsed: 1:24 mins



Call Stack	Instances	Bytes
c:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\DOMPrint.exe "c:\sysinternals\LogFile.xml"	4,136	78,256,467
ntdll:RtlAllocateHeap	4,136	78,256,467
ntdll:RtlpNtSetValueKey	4,136	78,256,467
ntdll:RtlDestroyMemoryBlockLookaside	4,136	78,256,467
ntdll:RtlAllocateHeap	4,136	78,256,467
ntdll	4,136	78,256,467
msvcr90d:malloc_base	4,132	78,253,895
msvcr90d:malloc_dbg	4,132	78,253,895
msvcr90d:malloc_dbg	4,132	78,253,895
msvcr90d:malloc_dbg	4,132	78,253,895
msvcr90d:malloc	4,131	78,253,747
msvcr90d:operator new	4,131	78,253,747
xerces-c_3_1d:xercesc_3_1::MemoryManagerImpl::allocate	4,126	78,253,539
xerces-c_3_1d:xercesc_3_1::ValueHashTableOf<bool,xercesc_3_1::StringHasher>::put	791	37,968
xerces-c_3_1d:xercesc_3_1::XMemory::operator new	746	75,364
xerces-c_3_1d:xercesc_3_1::DOMDocumentImpl::allocate	372	142,664
xerces-c_3_1d:xercesc_3_1::XMemory::operator new	332	21,596
xerces-c_3_1d:xercesc_3_1::RangeToken::doCreateMap	291	19,788
xerces-c_3_1d:xercesc_3_1::RangeToken::addRange	223	22,300
xerces-c_3_1d:xercesc_3_1::XMLString::replicate	202	11,906
xerces-c_3_1d:xercesc_3_1::DOMDocumentImpl::allocate	152	77,583,712
xerces-c_3_1d:xercesc_3_1::RefHashTableOf<xercesc_3_1::RangeTokenElemMap,xercesc_3_	146	7,008
xerces-c_3_1d:xercesc_3_1::RefHashTableOf<xercesc_3_1::CPMapEntry,xercesc_3_1::StringH	144	6,912
xerces-c_3_1d:xercesc_3_1::RangeToken::expand	64	74,052

- Quickly rate performance of activities that interest you
- Nothing influences a discussion more than evidence
- Reward your efforts by working in those areas that will make a difference
- Surprise yourself by delivering optimizations you might not have known existed

Usage

The Memory Profile can be used to reveal how activities perform in regard to memory consumption. Using this mode, a user would be interested in questioning the frequency of demands made for memory during a task. They would be less interested in the actual amount consumed. A well managed activity might make relatively few calls to allocate resources but allocate enough memory to do its job efficiently. Other activities might make many thousands of requests, and that typically makes them less efficient. This mode is useful for detecting those scenarios.

Operation

The Memory Profile works by hooking the process in question, so that program has to be launched using the tool in Enterprise Architect. Unlike the Call Graph option, you cannot attach to an existing process. When the program is started, hooking mechanisms track the allocation of memory; this information is collected and collated in Enterprise Architect. You can easily monitor the number of allocations being made. Also, the process is controlled; that is, the memory hooks can be turned on and off on demand. If you might have mistimed some action, you can pause capture, discard the data and resume capture again easily.

Results

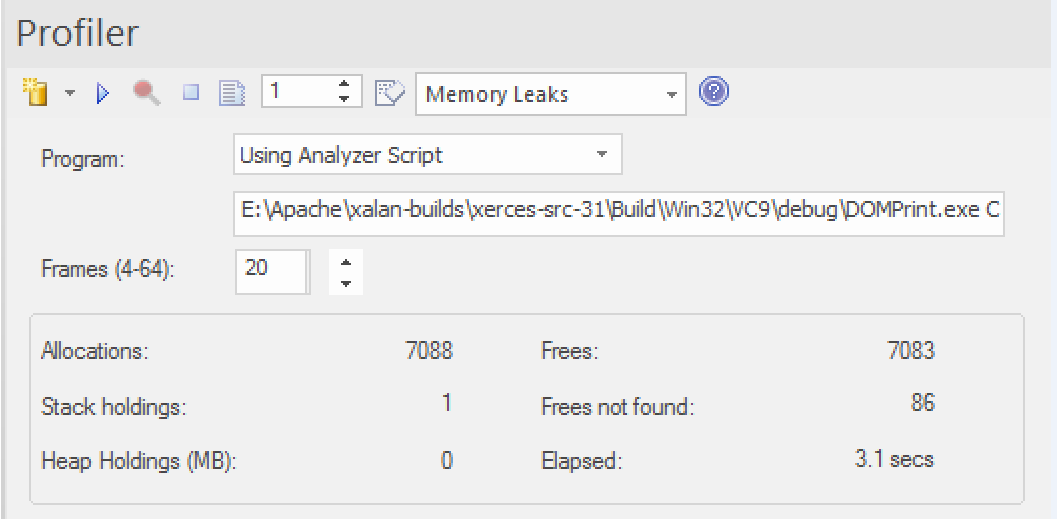
Results can be produced at any time during the session; however, capture must be disabled in order for the Report button to become active. It is your decision how long you let the Profiler run. You enable the Report button by either pausing capture or stopping the Profiler altogether.

Results are displayed in a Report view. The report initially opens with two tabs visible; a single weighted Call Graph and a Function Summary. The Call Graph depicts all the Call Stacks that led to memory allocations, which are aggregated and weighted according to the frequency of the pattern.

Requirements

For best results, the image and its modules should be built with debug information included, and without optimizations. Any module with the Frame Pointer Omission (FPO) optimization is likely to produce misleading results.

Memory Leaks



Profiler

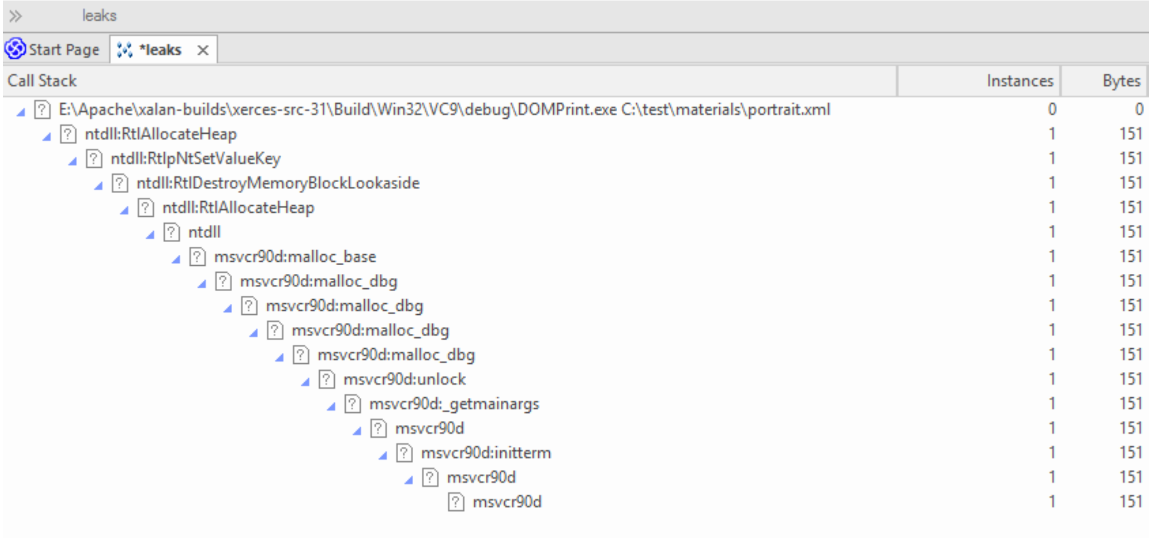
Program: Using Analyzer Script

E:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\DOMPrint.exe C

Frames (4-64): 20

Allocations:	7088	Frees:	7083
Stack holdings:	1	Frees not found:	86
Heap Holdings (MB):	0	Elapsed:	3.1 secs

The Profiler control, showing the count of memory allocations and the count of operations that are memory free.



```
Call Stack                                                                                          Instances   Bytes
E:\Apache\xalan-builds\xerces-src-31\Build\Win32\VC9\debug\DOMPrint.exe C:\test\materials\portrait.xml      0       0
  ntdll:RtlAllocateHeap                                                                                     1     151
    ntdll:RtlpNtSetValueKey                                                                                 1     151
      ntdll:RtlDestroyMemoryBlockLookaside                                                                  1     151
        ntdll:RtlAllocateHeap                                                                               1     151
          ntdll                                                                                             1     151
            msvcr90d:malloc_base                                                                            1     151
              msvcr90d:malloc_dbg                                                                           1     151
                msvcr90d:malloc_dbg                                                                         1     151
                  msvcr90d:malloc_dbg                                                                       1     151
                    msvcr90d:malloc_dbg                                                                     1     151
                      msvcr90d:unlock                                                                       1     151
                        msvcr90d:_getmainargs                                                               1     151
                          msvcr90d                                                                          1     151
                            msvcr90d:initterm                                                               1     151
                              msvcr90d                                                                      1     151
                                msvcr90d                                                                    1     151
```

A well behaved program.

Memory leak detection is a road well traveled. Although many other good options are available, we believe our approach has major benefits, such as:

- No changes at all to existing project build
- No header files required by the project code
- No runtime dependencies to worry about
- No system configuration to think about

Usage

A person would use this mode to track memory leaks in an application or in an activity within the application. A memory leak from the Profiler's point of view is a successful call made to a memory allocation function that returns a memory address for which no matching call is made to free that address.

Operation

The Memory Leak detection works through hooking. The memory routines of the process are hooked to track when memory is both allocated and freed. Call Stacks are captured at the point of the allocation and this information is collated in Enterprise Architect to produce a report in the form of a Call Graph. Capture is controlled; that is, the hooking mechanisms can be enabled or disabled on demand.

Depending on the type of program and its memory consumption, you could employ an appropriate strategy. For small programs, you might track the program from start to finish. For larger windowed programs, you would probably do better by toggling capturing before and after a specific task to avoid tracking too much data.

Results

Results can be produced at any time during the session; however, capture must be disabled in order for the Report button to become active. It is your decision how long you let the Profiler run. You enable the Report button by either pausing capture or stopping the Profiler altogether.

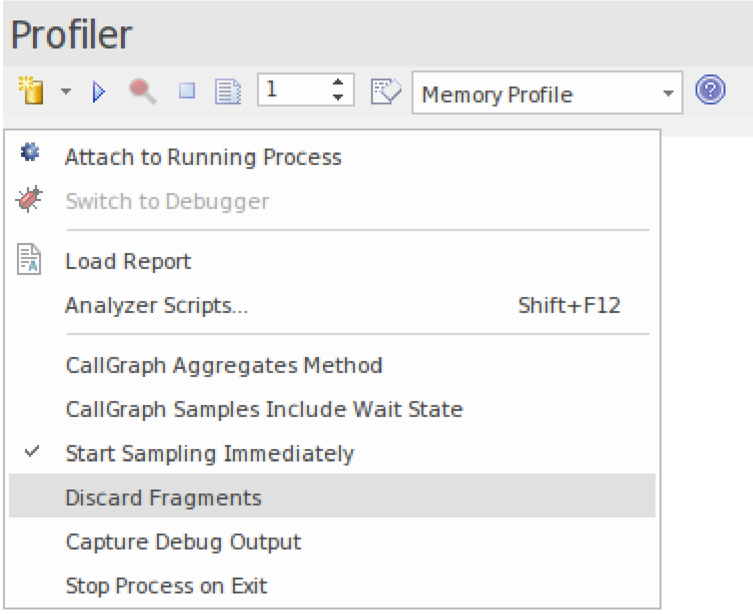
Results are displayed in a Report view. The report initially opens with two tabs visible; a single weighted Call Graph and a Function Summary. The Call Graph depicts all the Call Stacks that led to memory allocations, and are aggregated and weighted according to the frequency of the pattern.

Reports can contain a variable amount of 'noise'. To focus on an area you have specific concerns for, locate a function known to you in the summary report and use that to navigate directly into the line in the graph where it is featured.

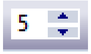
Requirements

For best results, the image and its modules should be built with debug information included, and without optimizations. Any module with the Frame Pointer Omission (FPO) optimization is likely to produce misleading results.

Setting Options



Call Graph Options

Option	Description
Interval	 Clicking on the up/down arrows, set the interval, in milliseconds, at which samples are taken of the target process; the range of possible values is 1 - 250 ms.
Attach to Running Process	Click on this option to select a running process.
Switch to Debugger	Click on this option to switch from Profiling to Debugging. The Debugger has an equivalent drop-down menu option that you can use to switch from Debugging to Profiling.
CallGraph Aggregates Method	When this option is selected, instances of the identical stack sequences are aggregated by method. That is to say, line numbers / instructions within a method are ignored, so two stacks will be counted as one where they differ only by line number in their final frame.
CallGraph Samples Include Wait State	When this option is selected, the Profiler will sample all threads, including those in Wait states. When unselected, the Profiler only samples threads that have accumulated CPU time since the last interval expired.
Discard Fragments	Sometimes, it just happens that the results of a stack walk operation can not be reconciled; that is, they do not appear to lead back to the entry point for the thread reporting the stack. We refer to these partial stack traces as 'fragments' and you can decide to display them or select this option to ignore them.
	Applies to Process Sampling. When selected, output normally visible during

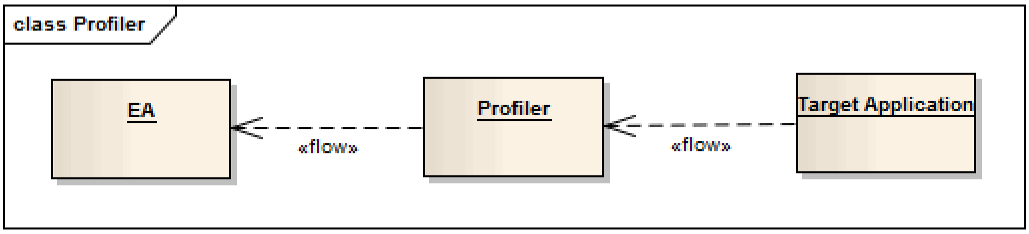
Capture Debug Output	debugging is captured and displayed in the Debugger window. Note that only debug builds will typically emit debug output.

General Options

Option	Description
Load Report	Select this option to load a previously saved report from the file system.
Analyzer Scripts	Select this option to open the Analyzer Script window, which is the model repository for configuring builds, debugging, and all other Visual Execution Analyzer options.
Start Sampling Immediately	Select this option to trigger Data Collection immediately on launch. You would typically use this option to profile a process during startup.
Stop Process on Exit	This option determines termination behavior for the Profiler. When the option is selected, the target process will terminate when the Profiler is stopped.

Start & Stop the Profiler

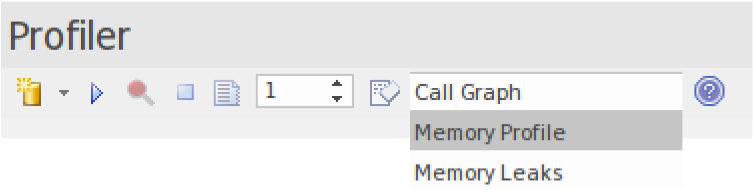
Profiling is a two stage process of data collection and reporting. In Enterprise Architect the data collection has the advantage of being a background task - so you are free to do other things while it runs. The information sent back to Enterprise Architect is stored until you generate a report. To view a report, the capture must be turned off. After the report is produced you can resume capture with the click of a button. If, for some reason, you decide to scrap your data and start again, you can do so easily and without having to stop and start the program again.



Access

Ribbon	Execute > Analyze > Profiler > Open Profiler
Other	Execution Analyzer toolbar : Analyzer Windows | Profiler

Actions

Action	Detail
Toolbar	 Profiler 1 Call Graph Memory Profile Memory Leaks
Strategy Selection	Select the Profiling strategy from the available options on the Toolbar.
Start the Profiler	Click the Run button on the Profiler window
Stop the Profiler	The process exits if: • You click on the Stop button • The target application terminates, or • You close the current model If you stop the Profiler and the process is still running, you can quickly attach to it again.
Pause and Resume Capture	You can pause and resume capture at any time during a session. When capture is turned on, samples are collected from the target. When paused, the Profiler enters and remains in a wait state until either capture is enabled, the

	Profiler is stopped or the application finishes.
Generate Reports	The Report button is disabled during capture but is available when capture is turned off.
Clear Data Collection	You can clear any data samples collected and resume at any time. First suspend capture by clicking on the Pause button. The Discard button, as for the Report button, is enabled whenever capture is turned off. In clicking on the Discard button you will be asked to confirm the operation. This action cannot be undone.

Function Line Reports

After you have run the Profiler on an executing application and generated a Sampler report, you can further analyze the activity of a specific function listed in the report by generating a Function Line report from that item. A Function Line report shows the number of times each line of the function was executed. You produce one Function Line report at a time, on any method in the Sampler report that has a valid source file. The Function Line report is particularly useful for functions that perform loops containing conditional branching; the coverage can provide a picture of the most frequently and least frequently executed portions of code within a single method.

The line report you generate is saved when you save the Sampler report. The body of the function is also saved with the Function Line report to preserve the function state at that time.

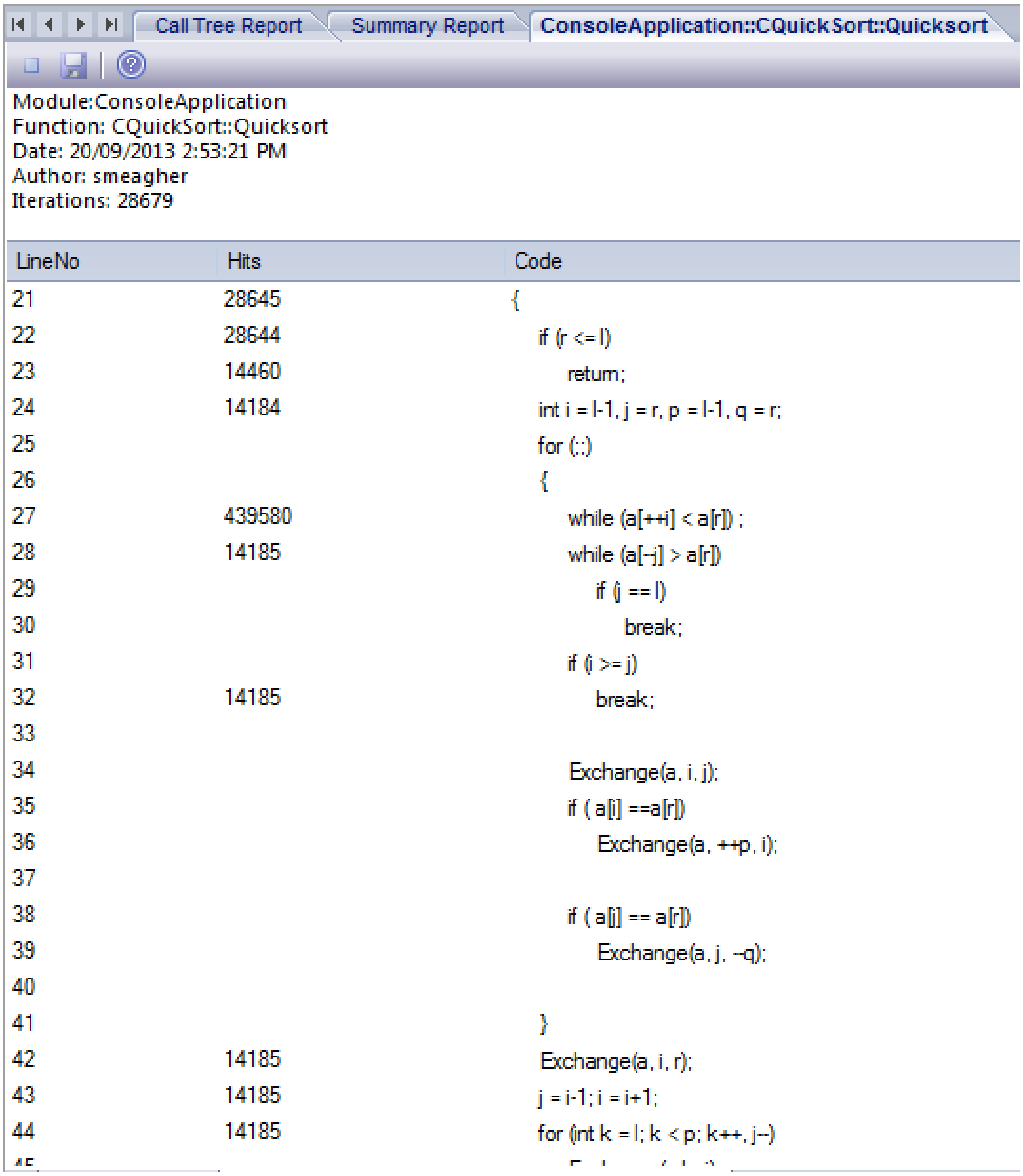
Platforms supported

Java, Microsoft .NET and Microsoft native code

Create a Line Report

In the Sampler report, right-click on the name of the function to analyze, and select the 'Create Line report for function' option.

Once the Profiler binds the method, the Function Line report is opened on the Sampler Report window. The report shows the body of the function, including line numbers and text. As each line is executed a hit value will accumulate against that line. A timer will update the report approximately once every second.



Module:ConsoleApplication
Function: CQuickSort::Quicksort
Date: 20/09/2013 2:53:21 PM
Author: smeagher
Iterations: 28679

```
LineNo      Hits        Code
21          28645       {
22          28644           if (r <= l)
23          14460               return;
24          14184           int i = l-1, j = r, p = l-1, q = r;
25                          for (;;)
26                          {
27          439580              while (a[++i] < a[r]) ;
28          14185               while (a[-j] > a[r])
29                                  if (j == l)
30                                      break;
31                              if (i >= j)
32          14185                   break;
33
34                              Exchange(a, i, j);
35                              if ( a[i] ==a[r])
36                                  Exchange(a, ++p, i);
37
38                              if ( a[j] == a[r])
39                                  Exchange(a, j, -q);
40
41                          }
42          14185           Exchange(a, i, r);
43          14185           j = i-1; i = i+1;
44          14185           for (int k = l; k < p; k++, j-)
```

End Line Report Capture

Once enough information is captured, or the function has ended, click on the Profiler toolbar Stop button to stop recording the capture.

Save Reports

Use the Save button on the Call Stack toolbar to save the Sampler report and any Function Line reports to a file.

Delete Line Reports

Closing the 'Line Report' tab will close that report but the report data will only be deleted when the report is saved.

Generate, Save and Load Profile Reports

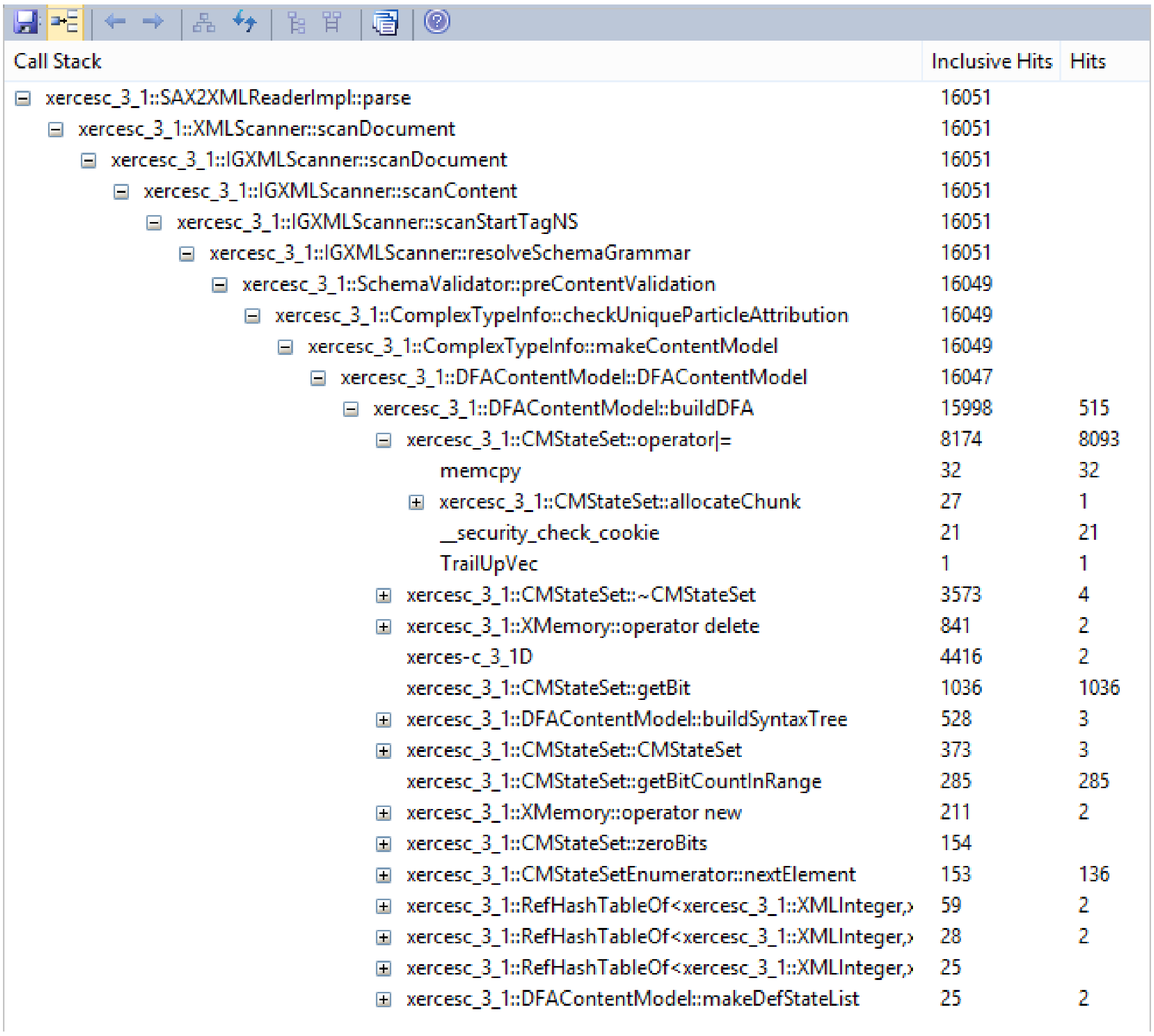
Reports can be produced at any time during a session, or naturally when a program ends. To enable the Report button while the program is running, however, you need to suspend Profiling by toggling the Pause/Resume button, or by terminating the Profiler with the Stop button. You have some options for reviewing and sharing the results:

- View the report
- Save the report to File
- Distribute the report as a Team Library resource
- Attach the report as a document to an Artifact element
- Synchronize the model by reverse engineering the source code that participated in the profile

Access

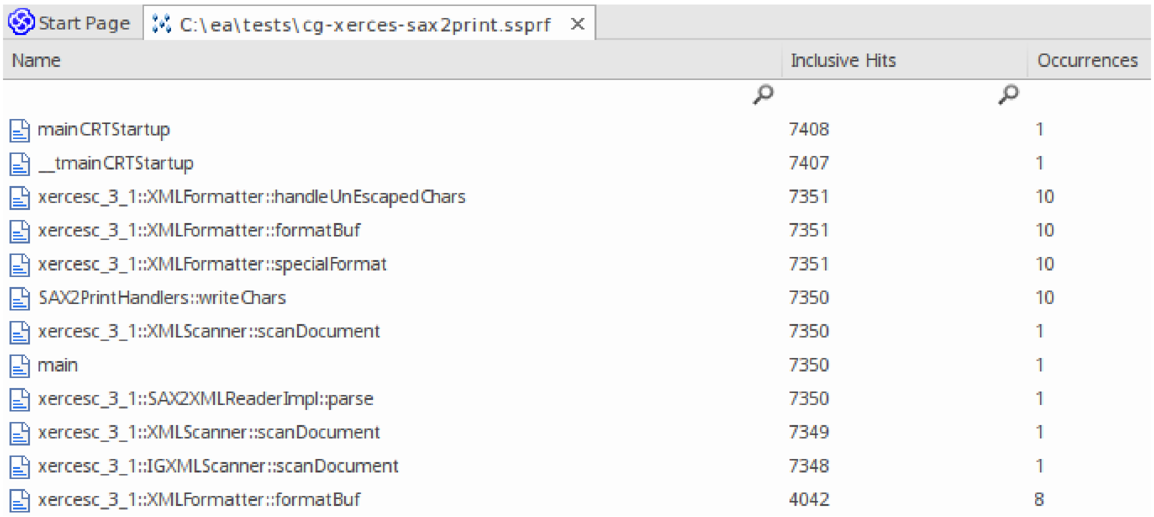
Ribbon	Execute > Analyze > Profiler > Create Report from Current Data
Other	From the Profiler window, click on the 🗎 icon in the toolbar.

CallFrequency Report



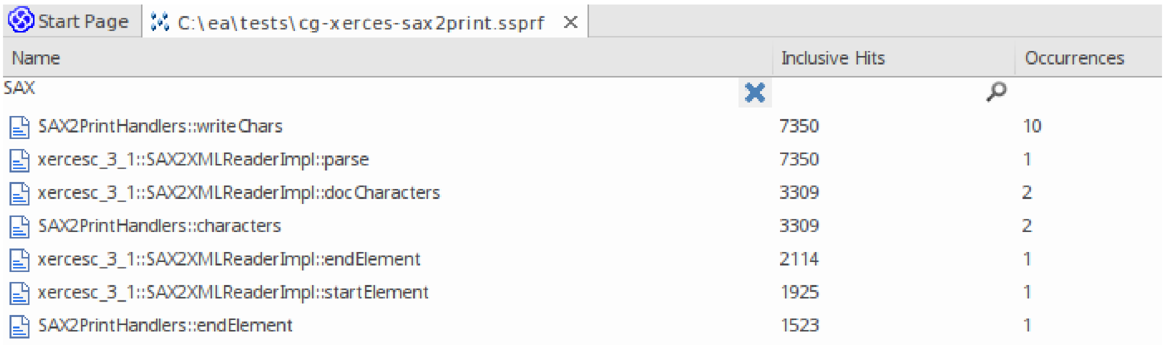
Call Stack	Inclusive Hits	Hits
⊟ xercesc_3_1::SAX2XMLReaderImpl::parse	16051	
⊟ xercesc_3_1::XMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanDocument	16051	
⊟ xercesc_3_1::IGXMLScanner::scanContent	16051	
⊟ xercesc_3_1::IGXMLScanner::scanStartTagNS	16051	
⊟ xercesc_3_1::IGXMLScanner::resolveSchemaGrammar	16051	
⊟ xercesc_3_1::SchemaValidator::preContentValidation	16049	
⊟ xercesc_3_1::ComplexTypeInfo::checkUniqueParticleAttribution	16049	
⊟ xercesc_3_1::ComplexTypeInfo::makeContentModel	16049	
⊟ xercesc_3_1::DFAContentModel::DFAContentModel	16047	
⊟ xercesc_3_1::DFAContentModel::buildDFA	15998	515
⊟ xercesc_3_1::CMStateSet::operator|=	8174	8093
memcpy	32	32
⊞ xercesc_3_1::CMStateSet::allocateChunk	27	1
__security_check_cookie	21	21
TrailUpVec	1	1
⊞ xercesc_3_1::CMStateSet::~CMStateSet	3573	4
⊞ xercesc_3_1::XMemory::operator delete	841	2
xerces-c_3_1D	4416	2
xercesc_3_1::CMStateSet::getBit	1036	1036
⊞ xercesc_3_1::DFAContentModel::buildSyntaxTree	528	3
⊞ xercesc_3_1::CMStateSet::CMStateSet	373	3
xercesc_3_1::CMStateSet::getBitCountInRange	285	285
⊞ xercesc_3_1::XMemory::operator new	211	2
⊞ xercesc_3_1::CMStateSet::zeroBits	154	
⊞ xercesc_3_1::CMStateSetEnumerator::nextElement	153	136
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	59	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	28	2
⊞ xercesc_3_1::RefHashTableOf<xercesc_3_1::XMLInteger,›	25	
⊞ xercesc_3_1::DFAContentModel::makeDefStateList	25	2

Function Summary



Name	Inclusive Hits	Occurrences
mainCRTStartup	7408	1
__tmainCRTStartup	7407	1
xercesc_3_1::XMLFormatter::handleUnEscapedChars	7351	10
xercesc_3_1::XMLFormatter::formatBuf	7351	10
xercesc_3_1::XMLFormatter::specialFormat	7351	10
SAX2PrintHandlers::writeChars	7350	10
xercesc_3_1::XMLScanner::scanDocument	7350	1
main	7350	1
xercesc_3_1::SAX2XMLReaderImpl::parse	7350	1
xercesc_3_1::XMLScanner::scanDocument	7349	1
xercesc_3_1::IGXMLScanner::scanDocument	7348	1
xercesc_3_1::XMLFormatter::formatBuf	4042	8

Unfiltered Summary Report listing all participating functions in order of inclusive hits.

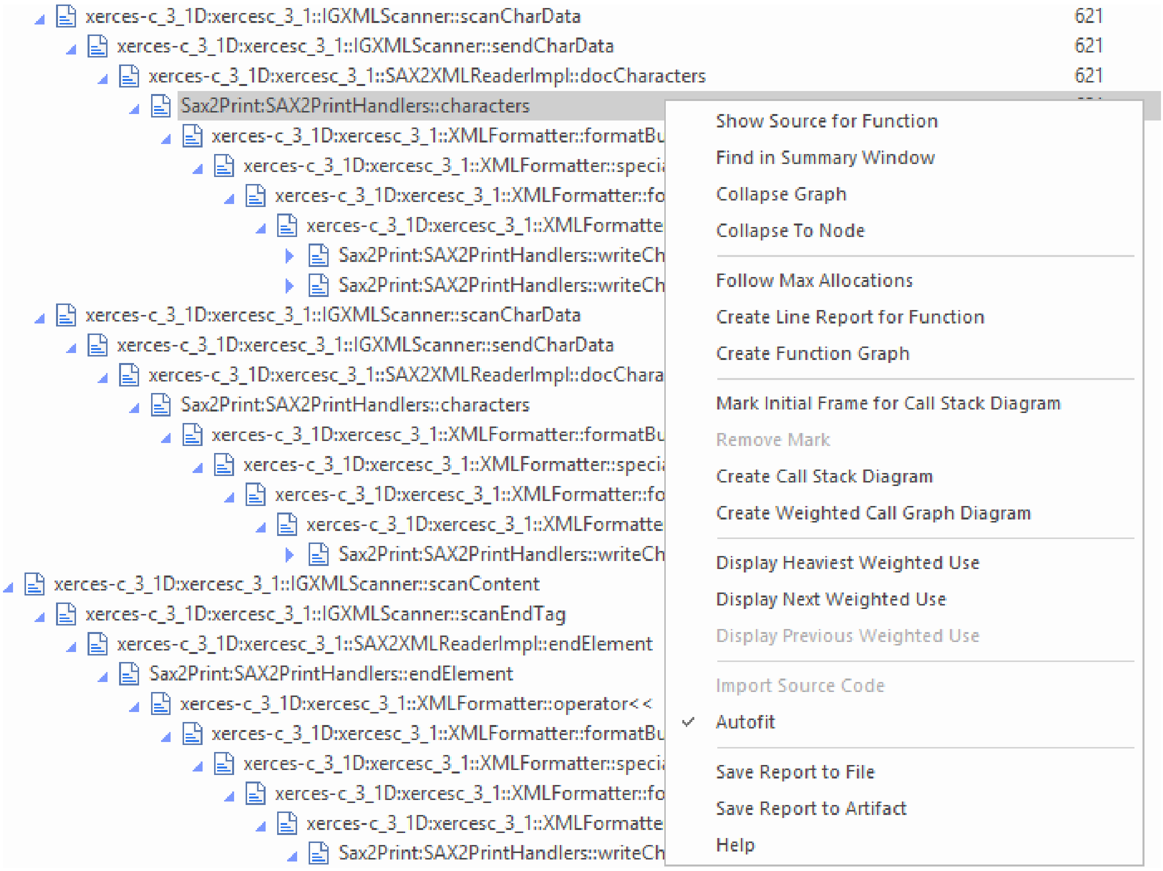


Name	Inclusive Hits	Occurrences
SAX		
SAX2PrintHandlers::writeChars	7350	10
xercesc_3_1::SAX2XMLReaderImpl::parse	7350	1
xercesc_3_1::SAX2XMLReaderImpl::docCharacters	3309	2
SAX2PrintHandlers::characters	3309	2
xercesc_3_1::SAX2XMLReaderImpl::endElement	2114	1
xercesc_3_1::SAX2XMLReaderImpl::startElement	1925	1
SAX2PrintHandlers::endElement	1523	1

You can filter and reorganize the information in the report, in the same way as you do for the results of a Model Search.

Report Options

Right-click on the report to display the context menu.



```
  xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanCharData                      621
    xerces-c_3_1D:xercesc_3_1::IGXMLScanner::sendCharData                    621
      xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::docCharacters            621
        Sax2Print:SAX2PrintHandlers::characters
          xerces-c_3_1D:xercesc_3_1::XMLFormatter::formatBu
            xerces-c_3_1D:xercesc_3_1::XMLFormatter::speci
              xerces-c_3_1D:xercesc_3_1::XMLFormatter::fo
                xerces-c_3_1D:xercesc_3_1::XMLFormatte
                  Sax2Print:SAX2PrintHandlers::writeCh
                  Sax2Print:SAX2PrintHandlers::writeCh
  xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanCharData
    xerces-c_3_1D:xercesc_3_1::IGXMLScanner::sendCharData
      xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::docChara
        Sax2Print:SAX2PrintHandlers::characters
          xerces-c_3_1D:xercesc_3_1::XMLFormatter::formatBu
            xerces-c_3_1D:xercesc_3_1::XMLFormatter::speci
              xerces-c_3_1D:xercesc_3_1::XMLFormatter::fo
                xerces-c_3_1D:xercesc_3_1::XMLFormatte
                  Sax2Print:SAX2PrintHandlers::writeCh
xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanContent
  xerces-c_3_1D:xercesc_3_1::IGXMLScanner::scanEndTag
    xerces-c_3_1D:xercesc_3_1::SAX2XMLReaderImpl::endElement
      Sax2Print:SAX2PrintHandlers::endElement
        xerces-c_3_1D:xercesc_3_1::XMLFormatter::operator<<
          xerces-c_3_1D:xercesc_3_1::XMLFormatter::formatBu
            xerces-c_3_1D:xercesc_3_1::XMLFormatter::speci
              xerces-c_3_1D:xercesc_3_1::XMLFormatter::fo
                xerces-c_3_1D:xercesc_3_1::XMLFormatte
                  Sax2Print:SAX2PrintHandlers::writeCh
```

Show Source for Function
Find in Summary Window
Collapse Graph
Collapse To Node
Follow Max Allocations
Create Line Report for Function
Create Function Graph
Mark Initial Frame for Call Stack Diagram
Remove Mark
Create Call Stack Diagram
Create Weighted Call Graph Diagram
Display Heaviest Weighted Use
Display Next Weighted Use
Display Previous Weighted Use
Import Source Code
✓ Autofit
Save Report to File
Save Report to Artifact
Help

Action	Detail
Show Source for Function	For the selected frame, select this option to display the corresponding line of code in a code editor. Frames that have source available are identifiable by their icon.
Find in Summary Window	Select this option to locate the function in the 'Summary' tab.
Collapse Graph	Select this option to collapse the entire graph including child nodes, visible or not.

Collapse to Node	Select this option to collapse the entire graph, then expand and set the focus to the selected node.
Follow Max Allocations	Select this option to expand an entire line in the graph.
Create Line Report for Function	Select this option to launch the Profiler (if it is not already running), immediately bind the selected function and ready it for recording. Once bound, an extra tab is opened in the current Report View. This report will update instantaneously, showing the number of times each line executed. Of course, the report will continue to record activity in the function even if is not visible. Note: In windowed programs, it is usually necessary to take some action in the application to cause the function to be invoked.
Create Function Graph	Select this option to create an additional tab, which shows the selected function in isolation. For a Call Frequency Profile, this produces a graph showing all the lines that led to this function being called (that is, the callers). For a Memory Profile, this produces a graph showing all lines that emanate from this function (that is, the callees).
Mark Initial Frame for Call Stack Diagram	Use prior to creating a Call Stack sequence diagram to limit the stack length. When this option is selected, the frame is marked and its text is highlighted. Frames above this one will then be excluded from any Sequence diagram produced.
Remove Mark	Removes the mark from a frame that was previously marked as 'Initial'.
Create Call Stack Diagram	Generates a sequence diagram for a single stack in the graph. The selected frame is depicted as the terminal frame in the stack. The initial frame of the stack defaults to the root node if no 'Initial' frame has been marked.
Create Weighted Call Graph Diagram	Generates a sequence diagram that presents a sequence for each visible stack branching from the selected frame. By expanding and collapsing the nodes of interest, a user can tailor the sequence diagram content to their liking.
Display the Heaviest Weighted Use	Select this option to display the line in the graph with the highest weight in which this function appears.
Display the Next Weighted Use	Select this option to navigate to the next line in the graph where the function appears. You can use the shortcut key combination Ctrl+Down Arrow.
Display the Previous Weighted Use	Select this option to navigate to the previous line in the graph where this function appears. You can also use the shortcut key combination Ctrl+Up Arrow
Save Report to File	Select this option to display the 'Save As' dialog, allowing you to choose where to store the report.
Save Report to Artifact	Note: Before selecting this option, go to the Project Browser and select the Package or element under which to create the Artifact element. You are prompted to provide a name for the report (and element); type this in and click on the OK button. The Artifact element is created in the Project Browser, underneath the selected Package or element. If you add the Artifact to a diagram as a simple link, when you double-click on the

	element the report is re-opened.

Team Library Options

Option	Description
Make Report a Team Library Resource	You can save any current report as a resource for a Category, Topic or Post in the Team Library to share and review at any time, as it is saved with the model. The report can also be compared with future runs. To begin this process, on the 'Team Library context menu select the menu option 'Share Resource | Add Active Profiler Report'.

Notes

- If you add the Profiler report to an Artifact element and also attach a Linked Document, the Profiler report takes precedence and is displayed when you double-click on the element; you can display the Linked Document using the 'Edit Linked Document' context menu option

Save Report in Team Library

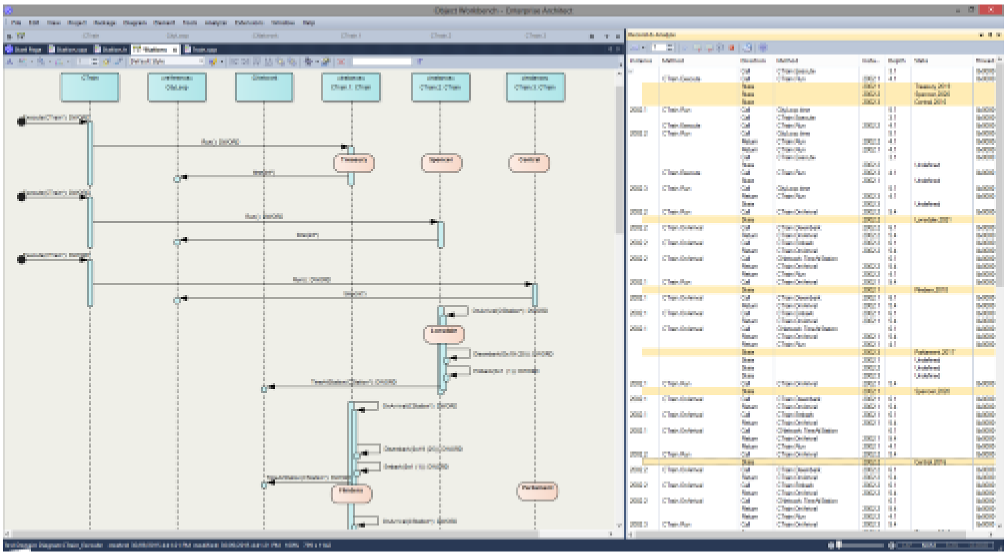
You can save any current report as a resource for a Category, Topic or Document in the Team Library. The report can then be shared and reviewed at any time as it is saved with the model. This helps you to:

- Preserve a Profiler report to compare against future runs
- Allow other people to investigate the profile

Access

Context Menu	Right-click in Team Library window | Share Resource | Active Profiler Report

Recording



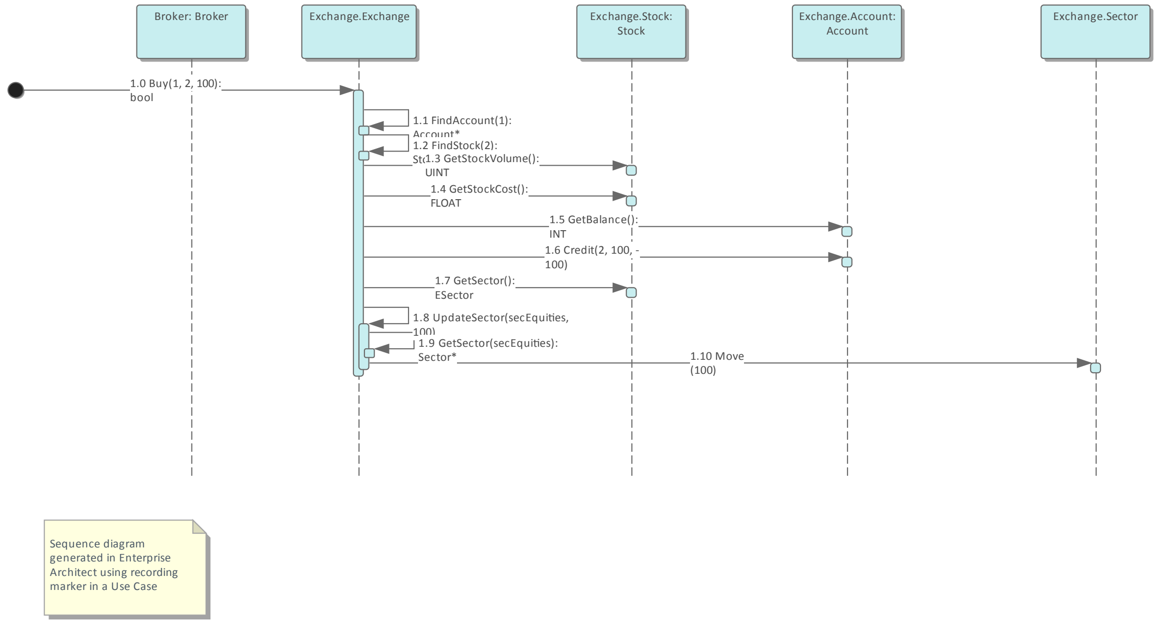
Sequence diagrams are a superb aid to understanding behavior. Class Collaboration diagrams also can be helpful. In addition to these, sometimes a Call Graph is just what we need. Then again, if you have this information available, you could use it to document a Use Case, and why not build a Test domain while you are at it? The Enterprise Architect Analyzer can generate all of these for you and from a single recording. It does this by recording a running program, and it works on all of the most popular platforms.

Access

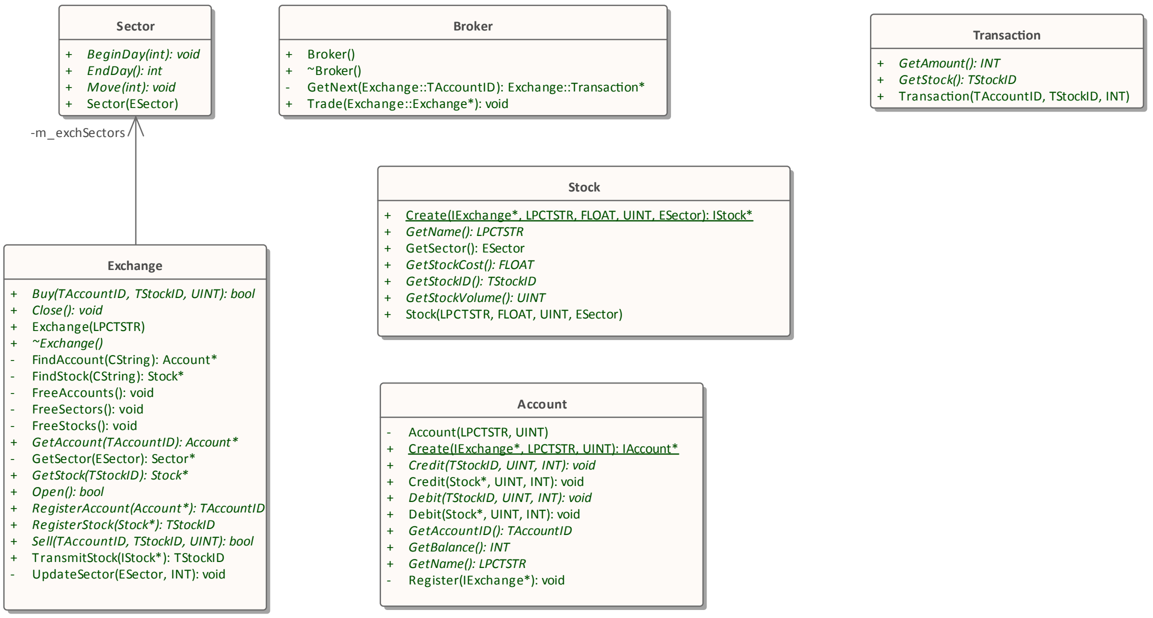
Ribbon	Execute > Analyze > Recorder > Open Recorder

Overview

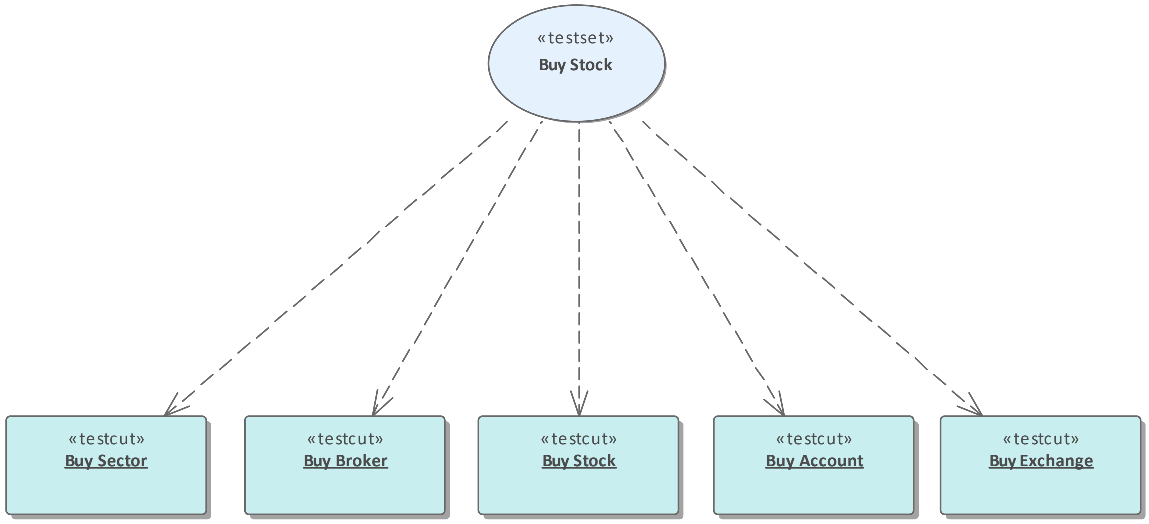
At its simplest, a Sequence diagram can be produced in very few steps, using even a brand new model. You do not even have to configure an Analyzer Script. Open the Enterprise Architect code editor (Ctrl+Shift+O), place a recording marker in a function of your choice, and then attach the Enterprise Architect debugger to a program running that code. Any time that function is called, its behavior will be captured to form a recording history. From this history these diagrams can be easily created.


Broker: Broker
Exchange.Exchange
Exchange.Stock: Stock
Exchange.Account: Account
Exchange.Sector
1.0 Buy(1, 2, 100): bool
1.1 FindAccount(1): Account*
1.2 FindStock(2): St
1.3 GetStockVolume(): UINT
1.4 GetStockCost(): FLOAT
1.5 GetBalance(): INT
1.6 Credit(2, 100, -100)
1.7 GetSector(): ESector
1.8 UpdateSector(secEquities, 100)
1.9 GetSector(secEquities): Sector*
1.10 Move (100)
Sequence diagram generated in Enterprise Architect using recording marker in a Use Case

The Sequence diagram from the Example Model recording.


Sector
+ BeginDay(int): void
+ EndDay(): int
+ Move(int): void
+ Sector(ESector)
-m_exchSectors
Broker
+ Broker()
+ ~Broker()
- GetNext(Exchange::TAccountID): Exchange::Transaction*
+ Trade(Exchange::Exchange*): void
Transaction
+ GetAmount(): INT
+ GetStock(): TStockID
+ Transaction(TAccountID, TStockID, INT)
Stock
+ Create(IExchange*, LPCTSTR, FLOAT, UINT, ESector): IStock*
+ GetName(): LPCTSTR
+ GetSector(): ESector
+ GetStockCost(): FLOAT
+ GetStockID(): TStockID
+ GetStockVolume(): UINT
+ Stock(LPCTSTR, FLOAT, UINT, ESector)
Exchange
+ Buy(TAccountID, TStockID, UINT): bool
+ Close(): void
+ Exchange(LPCTSTR)
+ ~Exchange()
- FindAccount(CString): Account*
- FindStock(CString): Stock*
- FreeAccounts(): void
- FreeSectors(): void
- FreeStocks(): void
+ GetAccount(TAccountID): Account*
- GetSector(ESector): Sector*
+ GetStock(TStockID): Stock*
+ Open(): bool
+ RegisterAccount(Account*): TAccountID
+ RegisterStock(Stock*): TStockID
+ Sell(TAccountID, TStockID, UINT): bool
+ TransmitStock(IStock*): TStockID
- UpdateSector(ESector, INT): void
Account
- Account(LPCTSTR, UINT)
+ Create(IExchange*, LPCTSTR, UINT): IAccount*
+ Credit(TStockID, UINT, INT): void
+ Credit(Stock*, UINT, INT): void
+ Debit(TStockID, UINT, INT): void
+ Debit(Stock*, UINT, INT): void
+ GetAccountID(): TAccountID
+ GetBalance(): INT
+ GetName(): LPCTSTR
- Register(IExchange*): void

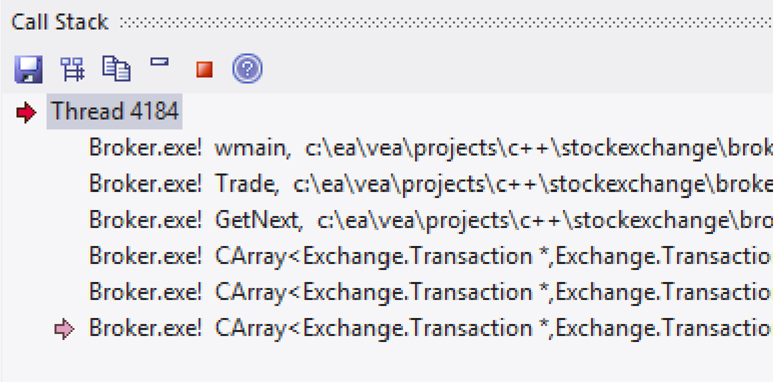
The Class Collaboration diagram from the same recording.



The Test Domain diagram from the same recording.

Of course, an Analyzer Script is still the best idea, and opens up an incredibly rich development environment, but it is worth noting that significant results can be obtained without one. This is also true of the Enterprise Architect Debugger and Profiler tools.

A point of interest: you can view a thread's behavior while it is recorded. Showing the Call Stack during a recording will show updates to a thread's stack in real-time, much like an animation. It is a good feedback tool and in some circumstances it might be all that is required.



```
Thread 4184
    Broker.exe!  wmain,  c:\ea\vea\projects\c++\stockexchange\brok
    Broker.exe!  Trade,  c:\ea\vea\projects\c++\stockexchange\broke
    Broker.exe!  GetNext,  c:\ea\vea\projects\c++\stockexchange\bro
    Broker.exe!  CArray<Exchange.Transaction *,Exchange.Transactio
    Broker.exe!  CArray<Exchange.Transaction *,Exchange.Transactio
    Broker.exe!  CArray<Exchange.Transaction *,Exchange.Transactio
```

Features at a glance

Diagram Generation

- Sequence diagram
- Class Collaboration diagram
- Test Domain diagram
- State Transition capture
- Call Graph

Control

- Support multi-threaded and single-threaded models
- Support stack depth control

- Support filters to restrict capture
- Filter wildcard support

- Real-time stack update

Integration

- Class Model
- Test Domain
- StateMachine
- Executable StateMachines
- Unit Tests

Platforms

- Microsoft .NET
- Microsoft Native
- Java
- PHP
- GDB
- Android

Requirements

- Recording is available to users of all editions of Enterprise Architect

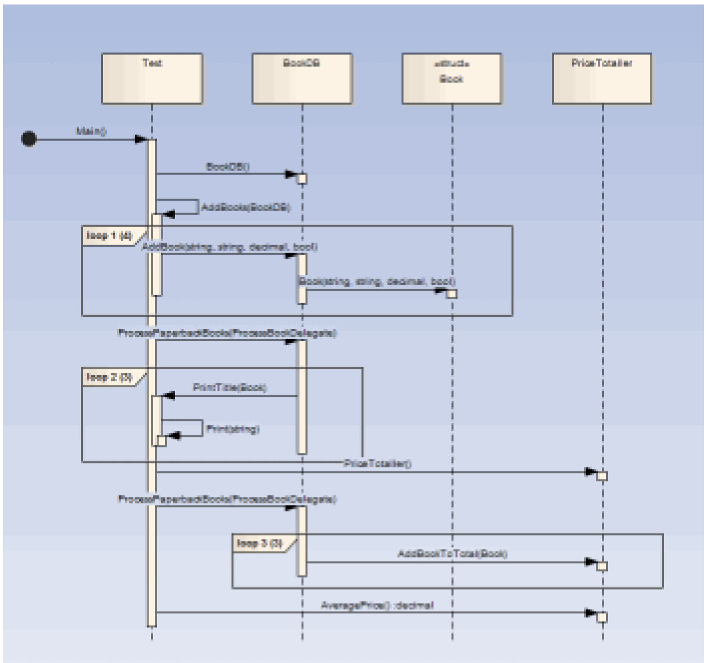
Notes

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

How it Works

This topic explains how the Visual Execution Analyzer generates Sequence diagrams.

Explanation

Points	Detail
Usage	The Visual Execution Analyzer enables you to generate a Sequence diagram from recordings of the live execution of an application. As the application runs, the history of each thread is recorded. This history can be used to generate the Sequence diagram. This is a Sequence diagram generated from a program that calculates the price of books:  Test BookDB «struct» Book PriceTotaller Main() BookDB() AddBooks(BookDB) loop 1 (4) AddBook(string, string, decimal, bool) Book(string, string, decimal, bool) ProcessPaperbackBooks(ProcessBookDelegate) loop 2 (3) PrintTitle(Book) Print(string) PriceTotaller() ProcessPaperbackBooks(ProcessBookDelegate) loop 3 (3) AddBookToTotal(Book) AveragePrice() :decimal How does the recorder know what to record? • The recorder works from recording markers; these are placed by you in the functions of interest Call Stacks in Java can stretch further than the eye can see. How can we restrict the recording to just ten frames? • The recorder is controlled by the depth either set on the recorder toolbar or associated with a Marker Set stored in the model
Its the real thing	In recording, the target application is not modified; no instrumentation of any image or module occurs at all. A recording produced using a 'Release' build of a program is a trustworthy document of what a program did.
Where do you start	We have a very large server application; so where do we start? If you have little or no understanding of the program you intend to record and little or no model to speak off, you might be best starting with the Profiler. Running the Profiler whilst using a program in a specific manner can quickly identify Use Cases from the entry

	points and Call Graphs presented. Having that knowledge can enable you to focus on areas that are uncovered and record those functions. If you have the source code, all you need to do is place a recording marker in a function that interests you. We recommend against placing multiple recording markers in multiple functions at the same time. In practice this has shown to be less helpful. Where do you place a recording marker? For windows UI programs, and in relation to some business use case, you might start by placing one in the event handlers for a message that seems most pertinent. If you are investigating a utility function, just set a method recording marker at or somewhere near the start. For services, daemons and batch processes you might want to profile the program once for each behavior of interest and use the report to explore those areas uncovered.
Tip	It's a good idea to have a quick glance at the Breakpoints and Markers window before debugging, and check that the markers listed here are what you are expecting.
Scenarios	• Microsoft Native C and C++, VB (Windows programs, Window Services, Console programs, COM servers, IIS ISAPI modules, Legacy) • Microsoft .NET (ASP.NET, Windows Presentation Foundation (WPF), Windows Forms, Workflow Services, devices, emulators) • Java (Apps, Applets, Servlets, Beans) • Android (using Android debug bridge for devices and emulators) • PHP (Web site scripts) • GDB (Windows / Linux interopability)

The Recording History

When the execution analysis of an application encounters user-defined recording markers, all information recorded is held in the Record & Analyze window.

Access

Ribbon	Execute > Analyze > Recorder > Open Recorder

Facilities

Facility	Information/Options
Information Display	The columns in the Record & Analyze window are: • Sequence - The unique sequence number • Threads - The operating system thread ID • Delta - The elapsed thread CPU time since the start of the sequence • Method - There are two Method columns: the first shows the caller for a call or for a current frame if a return; the second shows the function called or the function it is returning to • Direction - Stack Frame Movement, can be Call, Return, State, Breakpoint or Escape (Escape is used internally when producing a Sequence diagram, to mark the end of an iteration) • Depth - The stack depth at the time of a call; used in the generation of Sequence diagrams • State - The state between sequences • Source - There are two Source columns: the first shows the source filename and line number of the caller for a call or, if a return, for a current frame; the second shows the source filename and line number of the function called or function returning • Instance - There are two Instance columns, which only have values when the Sequence diagram produced contains State Transitions; the values consist of two items separated by a comma - the first item is a unique number for the instance of the Class that was captured, and the second is the actual instance of the Class For example: supposing a Class 'CName' has an internal value of 4567 and the program created two instances of that Class; the values might be: - 4567,1 - 4567,2 The first entry shows the first instance of the Class and the second entry shows the second instance
Operations on Information	The Record & Analyze window toolbar provides a range of facilities for controlling the recording of the execution of an Analyzer script. You can perform a number of operations on the results of a recording, using the Record & Analyze window context menu, once the recording is complete.

Notes

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

Diagram Features

When you generate a Sequence diagram, it includes these features:

Features

Feature	Detail
References	When the Visual Execution Analyzer cannot match a function call to an operation within the model, it still creates the Sequence but also creates a reference for any Class that it cannot locate. It does this for all languages.
Fragments	Fragments displayed in the Sequence diagram represent loops or iterations of a section(s) of code. The Visual Execution Analyzer attempts to match function scope with method calls to as accurately as possible represent the execution visually.
States	If a StateMachine has been used during the recording process, any transitions in State are presented after the method call that caused the transition to occur. States are calculated on the return of every method to its caller.

Setup for Recording

This section explains how to prepare to record execution of the application.

Steps

Step
Prerequisites - To set up the environment for recording Sequence diagrams you must: • Have completed the basic set up for Build & Debug and created Execution Analysis scripts for the Package • Be able to successfully debug the application
Narrow the focus of a recording by applying filters.
Control the detail of a recording by adjusting the stack depth.

Control Stack Depth

When recording particularly high-level points in an application, the Stack Frame count can result in a lot of information being collected; to achieve a quicker and clearer picture, it is better to limit the stack depth on the toolbar of either:

- The Breakpoint and Markers window or
- The Record & Analyze window

Access

Ribbon	Execute > Analyze > Recorder > Open Recorder

Set the recording stack depth

You set the recording stack depth in the numerical field on the toolbar of the Breakpoints & Markers window or the Record & Analyze window:

3

By default, the stack depth is set to three frames. The maximum depth that can be entered is 30 frames.

The depth is relative to the stack frame where a recording marker is encountered; so, when recording begins, if the stack frame is 6 and the stack depth is set to 3, the Debugger records the frames 6 through 8.

For situations where the stack is very large, it is recommended that you first use a low stack depth of 2 or 3. From there you can gradually increase the stack recording depth and insert additional recording markers to expand the picture until all the necessary information is displayed.

Place Recording Markers

This section explains how to place recording markers, which enable you to silently record code execution between two points. The recording can be used to generate a Sequence diagram.

As this process records the execution of multiple threads, it can be particularly useful in capturing event driven sequences (such as mouse and timer events).

Access

Ribbon	Execute > Windows > Breakpoints

Actions

Action
Different recording markers can be used for recording the execution flow; see the related links for information on the properties and usage of these markers.
Manage breakpoints in the Breakpoint & Markers window.
Activate and deactivate markers.
Working with Marker Sets - when you create a breakpoint or marker, it is automatically added to a marker set, either the Default set or a set that you create for a specific purpose.

Notes

- The *Breakpoint and Marker Management* topic (Software Engineering) describes a different perspective

Set Record Markers

Markers are set in the source code editor. They are placed on a line of code; when that line of code executes, the Execution Analyzer performs the recording action appropriate to the marker.

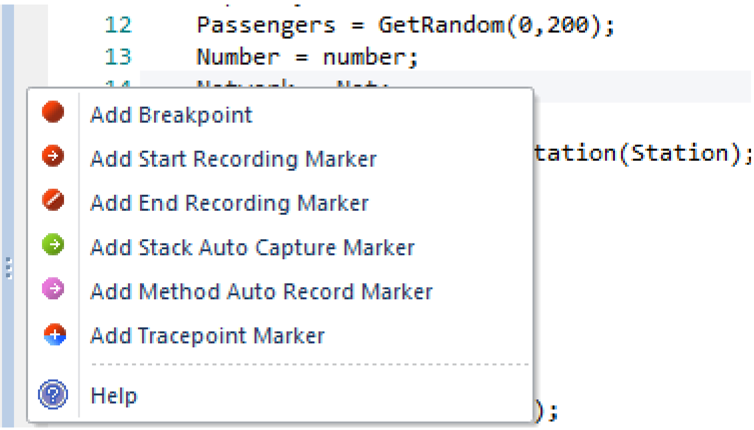
Access

Use one of the methods outlined here, to display the Code Editor window and load the source code associated with the selected Class.

Ribbon	Code > Source Code > Edit > Edit Element Source
Keyboard Shortcuts	F12

Set a recording marker

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

2	Find the appropriate code line and right-click in the left (Breakpoint) margin to bring up the breakpoint/marker context menu; select the required marker type:
	
	`Passengers = GetRandom(0,200);`
	`Number = number;`
	`tation(Station);`
	`);`
	 Add Breakpoint
	Add Start Recording Marker
	Add End Recording Marker
	Add Stack Auto Capture Marker
	Add Method Auto Record Marker
	Add Tracepoint Marker
	Help

3	If a Start Recording Marker has been set, you must also set an End Recording Marker.

Marker Types

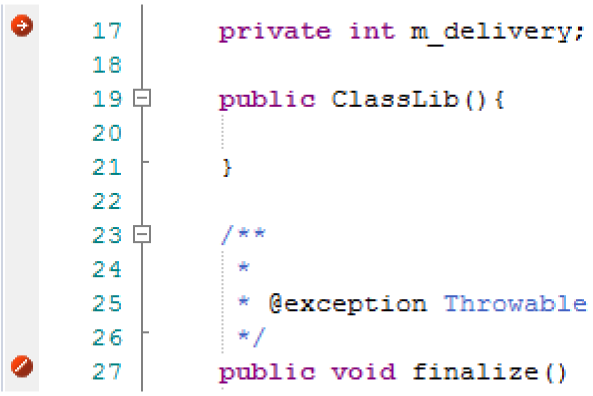
Markers are really fantastic. Unusual by their very light footprint when used with care, their impact on the performance of the programs being recorded can be negligible. Markers come in several flavors (well colors actually) and more are always being added. They are placed and are visible in the left margin of the editor, so you will need to have some source code.

Use to

- Record a single function
- Record parts of a function
- Use Cases spanning multiple functions
- Record call stacks
- Generate Sequence diagrams
- Generate Test Domain diagrams
- Generate Class Collaboration diagrams

Reference

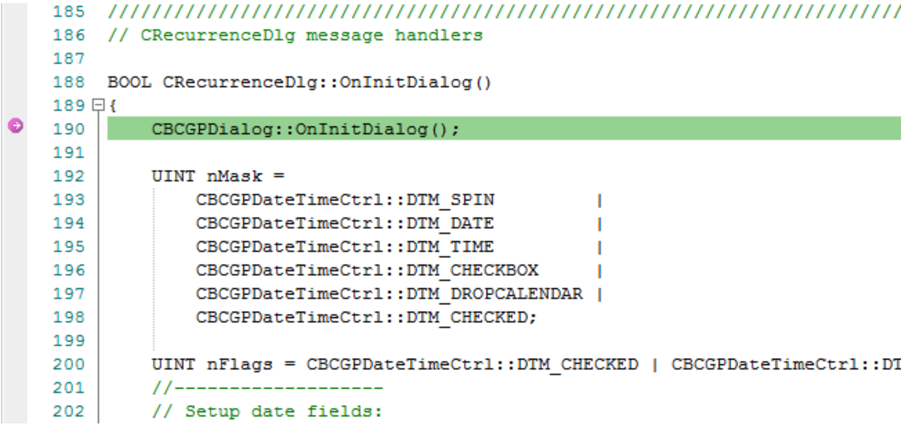
Marker	Detail

Start / End Recording markers	Place the markers at the start and end lines of the code to record. These need not be within the same function.
	
	`private int m_delivery;`
	
	`public ClassLib(){`
	
	`}`
	
	`/**`
	` *`
	` * @exception Throwable`
	` */`
	`public void finalize()`
	When the program encounters a start recording marker, a new recording is initiated (*the camera starts rolling!).* When an end marker is encountered, the current recording ends *(it's a take*). How you use these markers is up to you and your knowledge of the system under your care.
	Advanced Stuff (nested markers):
	If a Start recording marker is encountered while a recording is in progress, but where *capture is inhibited by the Stack depth value in use*, a separate recording will be initiated. Each recording is kept on a stack. When one ends, it is removed. This technique can be used in Enterprise Architect to record and render scenes in very complex systems. It is rather like splicing short scenes from a video to create a trailer. If you only want to record a single function, you should use an Auto record marker.

<table>
<tr>
<td>Method Auto Record marker</td>
<td>

A Method Auto Record marker enables you to record a particular function. The debugger will automatically end the recording when the function completes. This is good because recording is an intensive operation.

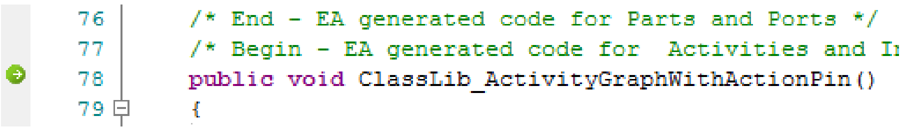
The function marker combines a Start Recording marker and an End Recording marker in one, so recording is executed after the marker point, and always stops when this function exits.



```
//////////////////////////////////////////////////////////////////////////
// CRecurrenceDlg message handlers

BOOL CRecurrenceDlg::OnInitDialog()
{
    CBCGPDialog::OnInitDialog();

    UINT nMask =
        CBCGPDateTimeCtrl::DTM_SPIN         |
        CBCGPDateTimeCtrl::DTM_DATE         |
        CBCGPDateTimeCtrl::DTM_TIME         |
        CBCGPDateTimeCtrl::DTM_CHECKBOX     |
        CBCGPDateTimeCtrl::DTM_DROPCALENDAR |
        CBCGPDateTimeCtrl::DTM_CHECKED;

    UINT nFlags = CBCGPDateTimeCtrl::DTM_CHECKED | CBCGPDateTimeCtrl::DI
    //-------------------
    // Setup date fields:
```

Recording markers can be nested. When a new Method Auto Record marker is hit while recording, the stack depth to record to will be extended to include the current method and the required depth from that function.

</td>
</tr>
<tr>
<td>Stack Auto-Capture marker</td>
<td>



```
    /* End - EA generated code for Parts and Ports */
    /* Begin - EA generated code for  Activities and I
    public void ClassLib_ActivityGraphWithActionPin()
    {
```

Stack markers enable you to capture any unique stack traces that occur at a point in an application; they provide a quick and useful picture of where a point in an application is being called from.

To insert a marker at the required point in code, right-click on the line and select the 'Add Stack Auto Capture Marker' option.

Each time the debugger encounters the marker it performs a stack trace; if the stack trace is not in the recording history, it is copied, and the application continues running.

</td>
</tr>
<tr>
<td>Limiting the recording depth</td>
<td>You can limit the depth of frames in any recording using the stack depth control on either the recorder and breakpoints toolbars.</td>
</tr>
</table>

The Breakpoints & Markers Window

Using the Breakpoints & Markers window, you can apply control to Visual Execution Analysis when recording execution to generate Sequence diagrams; for example, you can:

- Enable, disable and delete markers
- Manage markers as sets
- Organize how markers are displayed, either in list view or grouped by file or Class

Access

Ribbon	Execute > Windows > Breakpoints

Working with Marker Sets

Marker sets enable you to create markers as a named group, which you can reapply to a code file for specific purposes.

You can perform certain operations from the Breakpoints & Markers window alone, but to understand and use markers and marker sets you should also display the appropriate code file in the 'Source Code Viewer' (click on the Class element and press F12).

Access

Ribbon	Execute > Windows > Breakpoints : ▤ ▾ toolbar icon

Using Marker Sets

Action	Details
Example of Use	You might create a set of Method Auto Record markers to record the action of various functions in the code, and a set of Stack Capture markers to record the sequence of calls that cause those functions to be called. You could then create Sequence diagrams from the recordings under each set.
Create a Marker Set	To create a marker set from the Breakpoints & Markers window, click on the drop-down arrow on the ▤ ▾ icon and select the 'New Set' option. The 'New Breakpoint Marker Set' dialog displays; in the 'Enter New Set Name' field, type a name for the set, and click on the Save button. The set name displays in the text field to the left of the 'Set Options' icon. Alternatively, you can either: • Create a Class Activity marker set or • Select the 'Save as Set' option from the 'Set Options' drop-down, to make an exact copy of the currently-selected set, which you can then edit
Accessing Sets	To access a marker set, click on the drop-down arrow on the text field to the left of the 'Set Options' icon, and select the required set from the list. The markers in the set are listed in the Breakpoints & Markers window. You would normally load a marker set prior to the point at which an action is to be captured. For example, to record a sequence involving a particular dialog, when you begin debugging you would load the set prior to invoking the dialog; once you bring up the dialog in the application, the operations you have marked are recorded.
Add Markers to Set	To add markers to a marker set, add each required marker to the appropriate line of code in the 'Source Code Viewer'. The marker is immediately added to whichever set is currently listed in the Breakpoints & Markers window. Each marker listed on the dialog has a checkbox in the 'Enabled' column;

	newly-added markers are automatically enabled, but you can disable and re-enable the markers quickly as you check the code.
Storage of Sets	When you create a marker set it is immediately saved within the model; any user using the model has access to that set. However, the Default set, which always exists for a model, is a personal workspace, is not shared and is stored external to the model.
Delete a Marker from a Set	Right-click on the marker and select the 'Delete Breakpoint' option.
Delete a Set	If you no longer require a marker set, access it on the Breakpoints & Markers window and select the 'Delete Selected Set' option from the 'Set Options' drop-down list. You can also clear all user-defined marker sets by selecting the 'Delete all sets' option; a prompt displays to confirm the deletion.

Notes

- Marker Sets are very simple and flexible but, as they are available for use by any user of the model, they can be easily corrupted; consider these guidelines:
  - When naming a set, use your initials in the name and try to indicate its use, so that other model users can recognize its owner and purpose
  - When using a set other than Default, avoid excessive experimentation so that you don't add lots of ad-hoc markers to the set
  - Make sure you are aware of which marker set is exposed in the Breakpoints & Markers window as you can easily inadvertently add markers to the set that are not relevant to the code file the set was created for
  - In any set, if you have added markers that don't have to be kept, delete them to maintain the purpose of the set; this is especially true of the Default set, which can quickly accumulate redundant ad-hoc markers

Control the Recording Session

The Record & Analyze window enables you to control a recording session. The control has a toolbar, and a history window that displays the recording history as it is captured. Each entry in this window represents a call sequence made up of one or more function calls.

Access

Open the Record & Analyze window using one of the methods outlined here.

You must also open the Execution Analyzer window ('Execute > Analyze | Analyzer Scripts'), which lists all the scripts in the model; you must select and activate the appropriate script for the recording.

Ribbon	Execute > Windows > Recorder > Open Recorder

Recorder Toolbar

You can access facilities for starting, stopping and moderating an execution analysis recording session through the Record & Analyze window toolbar.

Access

Ribbon	Execute > Windows > Recorder > Open Recorder > Toolbar

Buttons

Button	Description
▾	Display a menu of options for defining what the recording session operates on. • Attach to Process - enabled even if no Analyzer Script exists, this option displays a dialog through which you select a process to record and a debugging platform to use; you can also optionally select a record marker set and/or a StateMachine to use during the recording • Generate Sequence Diagram from Recording - generate a Sequence/State diagram from the Execution Analyzer trace • Generate Testpoint Diagram from History - generate a Test Domain diagram from the Execution Analyzer trace, that can be used with the Testpoint facility • Generate Class Diagram from History - generate a Collaboration Class diagram from the Execution Analyzer trace, depicting only those Classes and operations involved in the recorded action (Use Case) • Generate Call Graph from History - generate a dynamic Call Graph from the recording history, as you might see in the VEA Profile workspace execution analysis layout; this can be more useful than the Sequence diagram in identifying the unique call stacks involved • Generate All - generate the Sequence, Testpoint and Collaboration Class diagrams together from the Execution Analyzer trace • Save as Artifact - create an Artifact element that contains the current recording history, under the currently-selected Package in the Project Browser; if you subsequently drag this Artifact element onto a Class diagram and double-click on it, the history recorded in the Artifact is copied back into the Record & Analyze window • Load Sequence History from file - select an XML file from which to restore a previously-saved recording history • Save Sequence History to file - save the recording history to an XML file
7 ⬍	Select the recording stack depth for the marker set; that is, the number of frames from the point at which recording began.
▷	Launch and record the application described in the script; you can optionally select a record marker set and/or a StateMachine to use during the recording. The icon is enabled when the active Analyzer Script is configured for debugging.

	Perform ad-hoc manual recording of the current thread during a debug session. Use this function with the 'step' buttons of the debugger; each function that is called due to a step command is logged to the history window. The icon is enabled if no recording is taking place and you are currently at a breakpoint (that is, debugging).
	Perform ad-hoc auto-recording during a debug session. When you click on 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. This icon is enabled if no recording is taking place and you are currently at a breakpoint (that is, debugging).
	Step into a function, record the function call in the History window, and step back out. Enabled for manual recording only.
■	Stop recording the execution trace.
	Display the 'Synchronize Model' dialog through which you can synchronize the model with the code files generated during a Record Profile operation.

Working With Recording History

You can perform a number of operations on or from the results of a recording session, using the Record & Analyze window context menu.

Options

Option	Action
Show Source for Caller	Display the source code, in the Source Code Viewer, for the method calling the sequence.
Show Source for Callee	Display the source code, in the Source Code Viewer, for the method being called by the sequence.
Generate Diagram for Selected Sequence	Generate a Sequence diagram for a single sequence selected in the recording history.
Generate Sequence Diagram	Generate a Sequence diagram including all sequences in the recording history.
Clear	Clear the recording history currently displayed in the Record & Analyze window.
Save Recording History to File	Save the recording history to an XML file. A browser window displays, on which you specify the file path and name for the XML file.
Load Recording History From File	Load a previously saved recording history from an XML file. A browser window displays, on which you specify the file path and name for the XML file to load.
Disable All Calls	Disable every call listed in the Record & Analyze window.
Disable This Call	Disable the selected call.
Disable This Method	Disable the selected method.
Disable This Class	Disable the selected Class.
Disable All Calls Outside This Call	Disable every call listed in the Record & Analyze window except for the selected call.
Enable All Calls	Enable every call listed in the Record & Analyze window.
Enable This Call	Enable the selected call.
Enable This Method	Enable the selected method.
Enable This Class	Enable the selected Class.

Help	Display the Help topic for the Record & Analyze window.

Start Recording

When you are recording execution flow as a Sequence diagram, you start the recording by selecting the 'Recording' icon on the Record & Analyze window toolbar. The 'Record' dialog displays with the recording options set to the defaults; that is, the current Breakpoint and Markers Set, the filters defined in the current Analyzer Script and the recording mode as basic.

Access

Ribbon	Execute > Windows > Recorder > Open Recorder : ▶

Record Dialog Options

Field/Button	Detail
Recording Set	Recording markers determine what is recorded. If you have a recording set to use, click on the drop-down arrow and select it.
Additional Filters	Filters are used by the debugger to exclude matching function calls from the recording history. Recording filters are defined in the Analyzer Script. In the 'Additional Filters' field you can add other filters for this specific run. if you specify more than one filter, separate them with a semi-colon.
Basic Recording Mode	In basic mode the debugger records a history of the function calls made by the program whenever it encounters an appropriate recording marker.
Track Instances of Named Classes	In Track Instances mode the debugger also captures the creation of instances of the Classes you specify. It then includes that information in the history. The resulting Sequence diagram can then show lifelines for each instance of that Class with, where appropriate, function calls linked to the lifeline.
Track State Transitions	The recording can also capture changes in State using a specified StateMachine diagram. The StateMachine diagram must exist as a child of a Class. The Execution Analyzer captures instances of that Class and calculates the State of each instance whenever a function in the current recording sequence returns.
OK	Click on this button to start the debugger.

Step Through Function Calls

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

Alternatively, press Shift+F6 or select the 'Execute > Run > Step In' ribbon option.

The 'Step Through' command causes a 'Step Into' command to be executed; if any function is detected, then that function call is recorded in the History window.

The debugger then steps out, and the process can be repeated.

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

Nested Recording Markers

When a recording marker is first encountered, recording starts at the current stack frame and continues until the frame pops, recording additional frames up to the depth defined on the Recording toolbar. Consider this call sequence:

A -> B -> C -> D -> E -> F -> G -> H -> I -> J -> K -> L -> M -> N -> O -> P -> Q -> R -> S -> T -> U -> V -> W -> X -> Y -> Z

If you set a recording marker at K and set the recording depth to 3, this would record the call sequence:

K -> L -> M

If you also wanted to record the calls X, Y and Z as part of the Sequence diagram, you would place another recording marker at X and the analyzer would record:

K -> L -> M -> X -> Y -> Z

However, when recording ends for the X-Y-Z component (frame X is popped), recording will resume when frame M of the K-L-M sequence is re-entered. Using this technique can help where information from the recorded diagram would be excluded due to the stack depth, and it lets you focus on the particular areas to be captured.

Generating Sequence Diagrams

This topic describes what you might do with the recording of an execution analysis session.

Access

Ribbon	Execute > Analyze > Recorder > Open Recorder

Reference

Action	Detail
Generate a diagram	Select the appropriate Package in the Project Browser, in which to store the Sequence diagram. To create the diagram from all recorded sequences, either: • Click on the 'Recorder Menu' icon () in the Record & Analyze window toolbar, and select the 'Generate Sequence Diagram from Recording' option, or • Right-click on the body of the window and select the 'Generate Sequence Diagram' option To create the diagram from a single sequence, either: • Click on the 'Recorder Menu' icon () in the Record & Analyze window toolbar, and select the 'Generate Sequence Diagram from Recording' option, or • Right-click on the sequence and select the 'Generate Diagram from Selected Sequence' option
Save a recorded sequence to an XML file	Click on the sequence, click on the 'Recorder Menu' icon () in the Record & Analyze window toolbar, and select the 'Save Sequence History to File' option.
Access an existing sequence XML file	Either: • Click on the in the Record & Analyze window toolbar, and select the 'Load Sequence History from File' option, or • Right-click on a blank area of the screen and click on the 'Load Sequence From File' option The 'Windows Open' dialog displays, from which you select the file to open.

Use to

- Generate a Sequence diagram from a recorded execution analysis session, for:
- all recorded sequences or

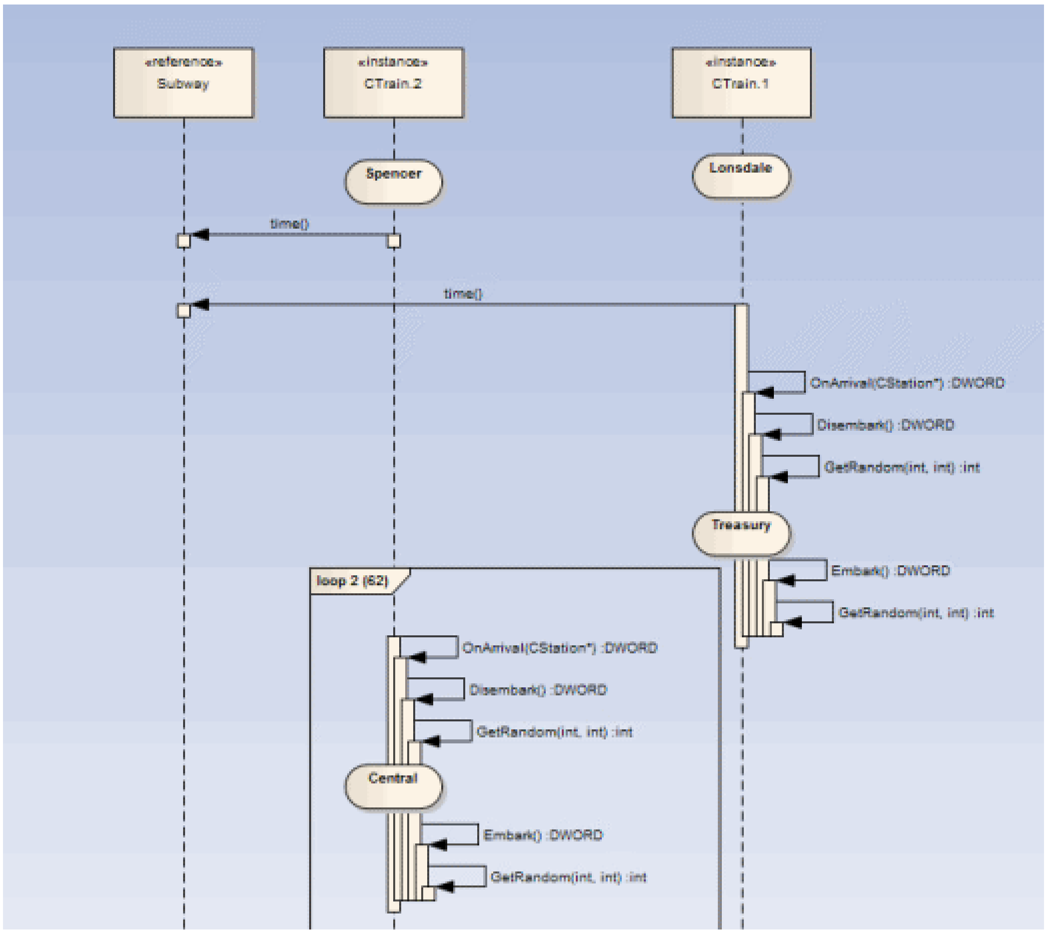
- a single sequence in the session
- Save the recorded sequence to file
- Retrieve the saved recording and load it into the Record & Analyze window

Reporting State Transitions

This section describes how you can generate Sequence diagrams that show transitions in state as a program executes.

Use to

- Generate Sequence diagrams that report user-defined transitions in state as a program executes (as shown in the example generated diagram)



Topic
Create a StateMachine under the Class to be reported.
Set the constraints against each State to define the change in state to be reported.

Reporting a StateMachine

The Execution Analyzer can record a Sequence diagram, we know that. What you might not know is that it can use a StateMachine at the same time to detect State transitions that might occur along the way. These States are represented at the point in time on the lifeline of the object. The transitions also are apparent from the lifelines. Any invalid or illegal transition will be highlighted with a red border. Have a look.

Process

Firstly you model a StateMachine for the appropriate Class element.

You then compose the expressions that define each State using the 'Constraints' tab of each State.

These simple expressions are formed using attribute names from Class model and actual code base. They are not OCL statements. Each expression should appear on a separate line.

```
m_strColor == "Blue"
```

You then use the Recorder window to launch the debugger.

The Recorder window Run button is different from the button on other debugger toolbars.

The Recorder window will allow you to browse for a StateMachine if you do not know the StateMachine name. The 'State Transition' dialog presents a list of StateMachines for the entire model, in which you locate and select the appropriate diagram (see the example).

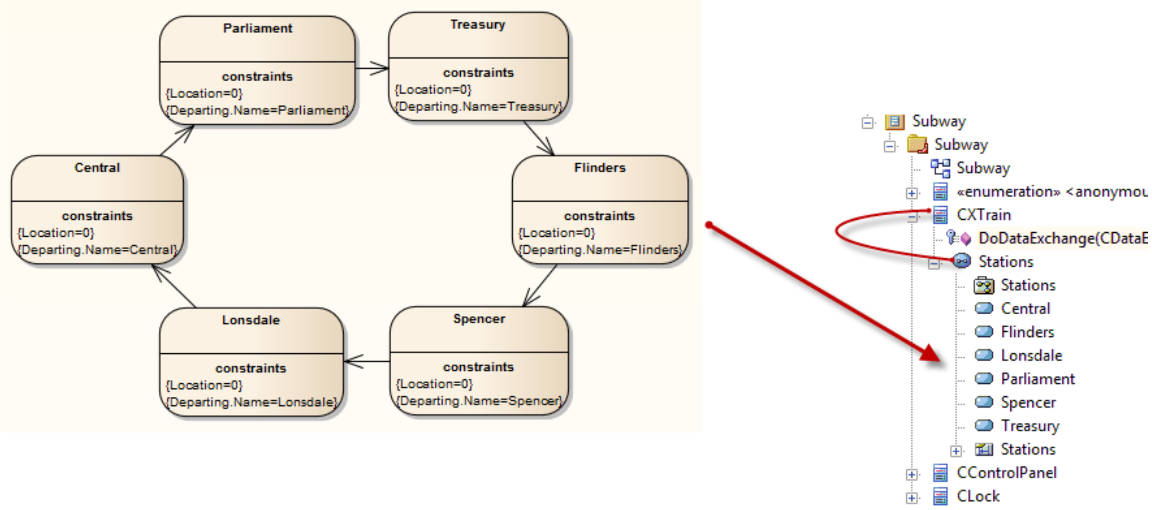
When you generate the Sequence diagram, it depicts not only the sequence but changes in State at the various points in the sequence; each Class instance participating in the detection process is displayed with its own lifeline.

Example

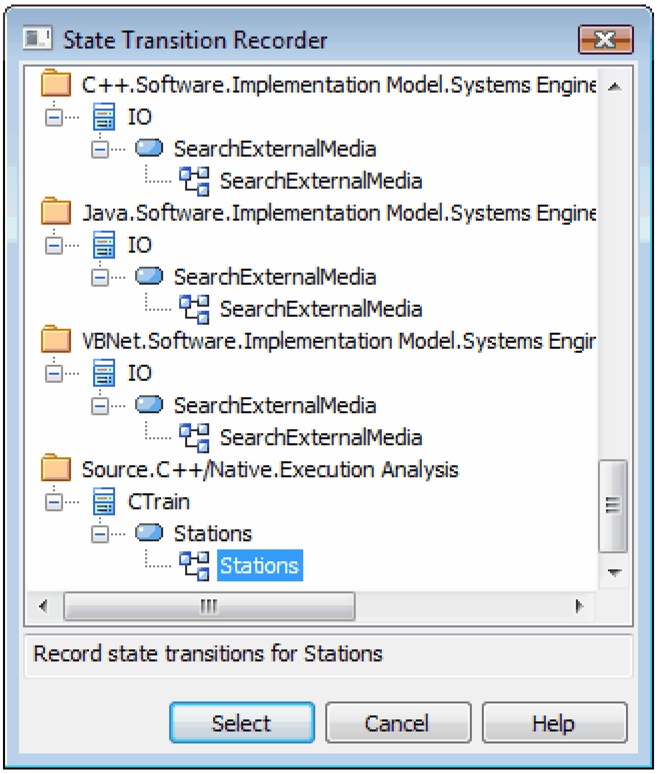
The Stations StateMachine shows the different States within the Melbourne Underground Loop subway system.

A train traveling on the subway network can be stopped at any of the stations represented on the StateMachine.

The Stations StateMachine is a child of the CTrain Class.



When you browse for the diagram in the 'State Transition Recorder' dialog, the hierarchy shows only the root Package, parent Class and child SubMachine and diagram; no other model components are listed.



State Transition Recorder
C++.Software.Implementation Model.Systems Engine
IO
SearchExternalMedia
SearchExternalMedia
Java.Software.Implementation Model.Systems Engine
IO
SearchExternalMedia
SearchExternalMedia
VBNet.Software.Implementation Model.Systems Engir
IO
SearchExternalMedia
SearchExternalMedia
Source.C++/Native.Execution Analysis
CTrain
Stations
Stations
Select
Cancel
Help

Record state transitions for Stations

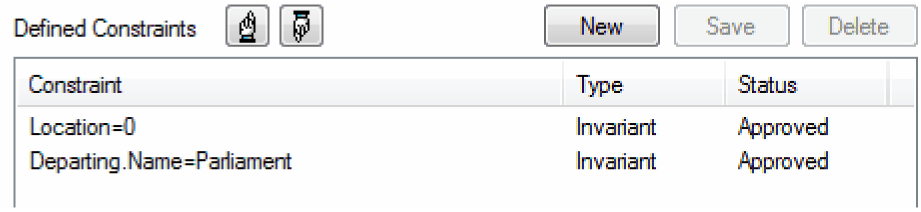
Recording and Mapping State Changes

This topic discusses how to set constraints against each State in the StateMachine under a Class, to define the change in state to be recorded.

Example

This example of a 'State Properties' dialog is for the State called Parliament; the 'Constraints' tab is open to show how the State is linked to the Class CXTrain.

A State can be defined by a single constraint or by many; in the example, the State Parliament has two constraints:



Constraint	Type	Status
Location=0	Invariant	Approved
Departing.Name=Parliament	Invariant	Approved

The values of constraints can only be compared for elemental, enum and string types

The CXTrain Class has a member called Location of type int, and a member called Departing.Name of type CString; what this constraint means is that this State is evaluated to True when:

- an instance of the CXTrain Class exists and
- its member variable Location has the value 0 and

the member variable Departing.Name has the value Parliament

Operators in Constraints

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

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

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

Location=0 OR

Location=1 AND

Departing.Name!=Central

Here are some examples of using the equivalence operators:

Departing.Name!=Central AND

Location!=1

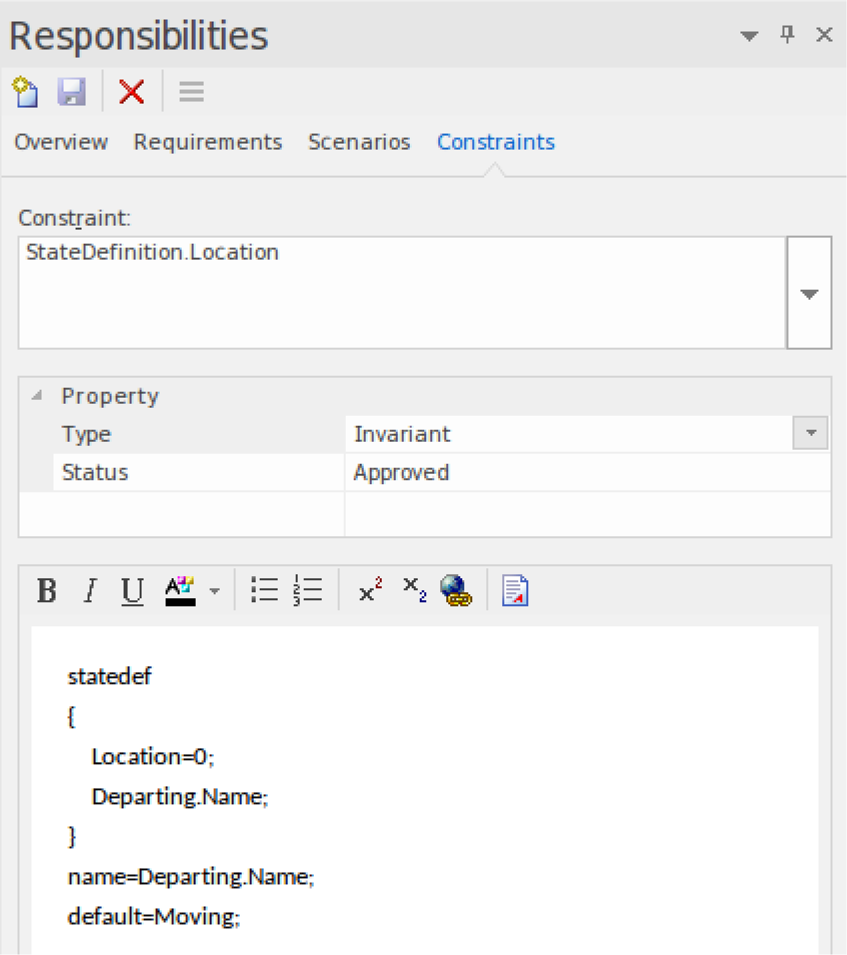
Notes

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

State Analyzer

The State Analyzer is a feature that can analyze, detect and record states for instances of a Class. The feature works by combining a state definition, defined on a Class as a constraint, and markers called State points. It is available on any languages supported by the Execution Analyzer, including Microsoft. NET, Mono, Java and native C++.

We begin by selecting a Class and composing our state definition.

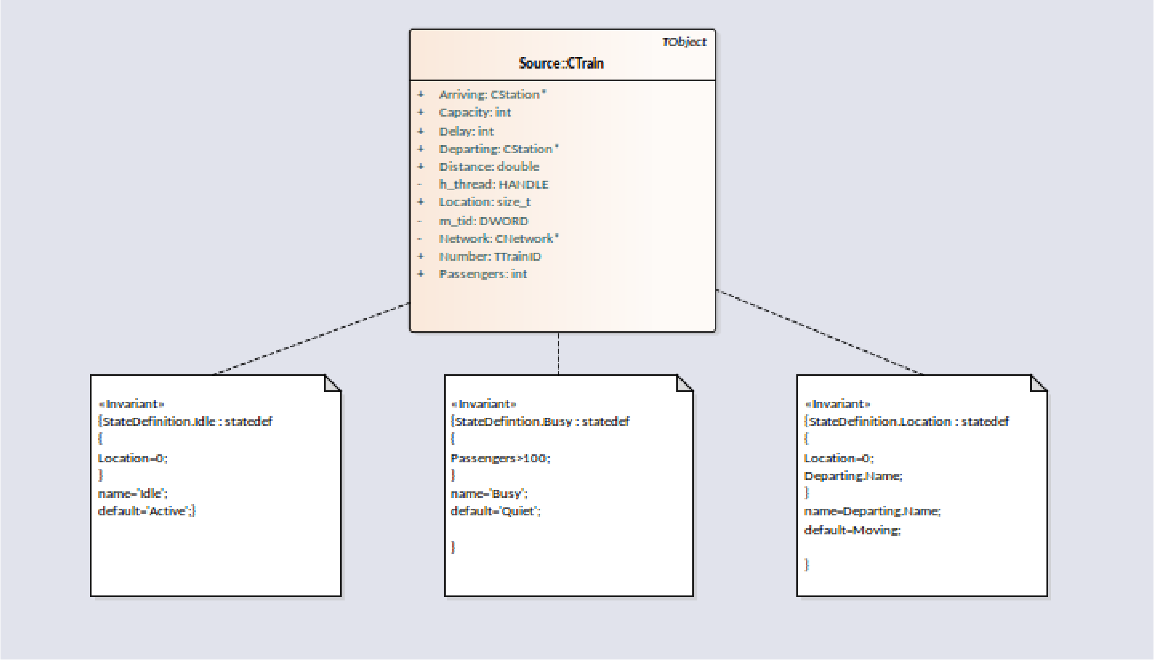


StateDefinition.Location

Property	
Type	Invariant
Status	Approved

```
statedef
{
   Location=0;
   Departing.Name;
}
name=Departing.Name;
default=Moving;
```

We can get a picture of all the state definitions we've defined by placing the Class on a diagram and linking notes to the Class that themselves link to a particular state definition constraint. We explain how to do that in a later section.

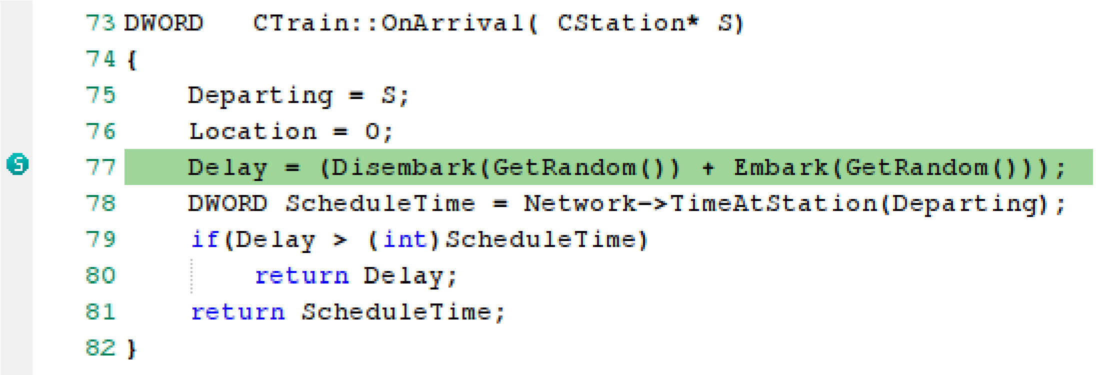


```
«Invariant»
{StateDefinition.Idle : statedef
{
Location=0;
}
name='Idle';
default='Active';}
```

```
«Invariant»
{StateDefintion.Busy : statedef
{
Passengers>100;
}
name='Busy';
default='Quiet';

}
```

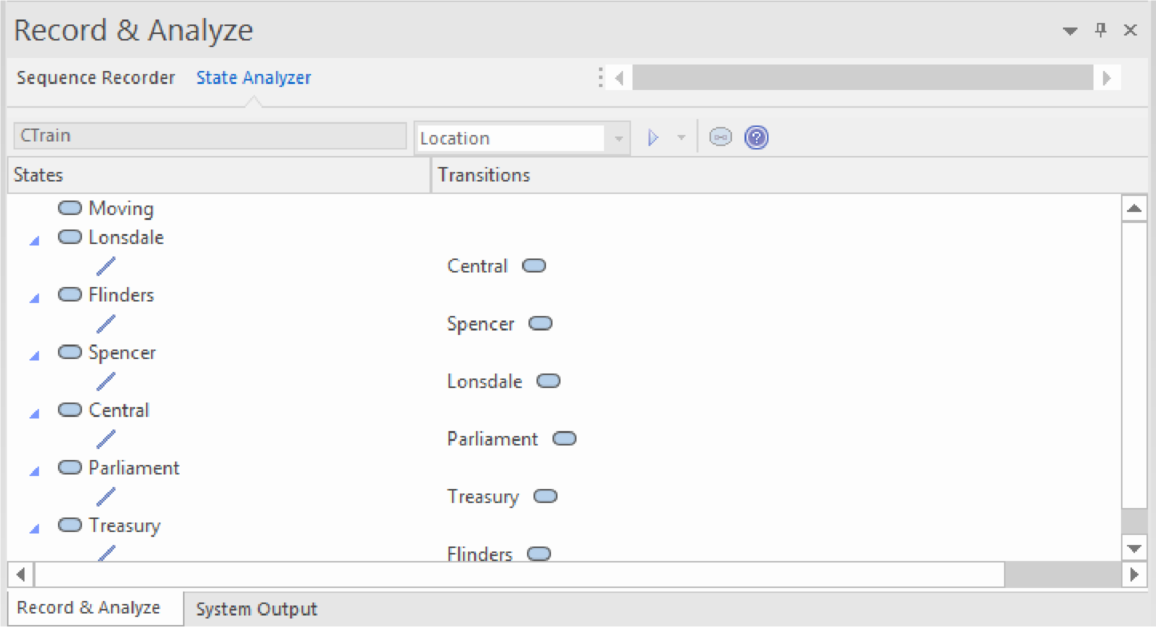
```
«Invariant»
{StateDefinition.Location : statedef
{
Location=0;
Departing.Name;
}
name=Departing.Name;
default=Moving;

}
```

State points are set by placing one or more markers in relevant source code.



```
DWORD   CTrain::OnArrival( CStation* S)
{
    Departing = S;
    Location = 0;
    Delay = (Disembark(GetRandom()) + Embark(GetRandom()));
    DWORD ScheduleTime = Network->TimeAtStation(Departing);
    if(Delay > (int)ScheduleTime)
        return Delay;
    return ScheduleTime;
}
```

The program to be analyzed is run using the State Analyzer control. When the Execution Analyzer encounters any State point, the current instance of the Class is analyzed. Where the value domain of the instance matches the state definition, a state is recorded. Each time the instance varies, new states are thus detected. The control lists each state as it is discovered. Under each state the control lists the discrete set of transitions to other states made by instances of the class.



Record & Analyze

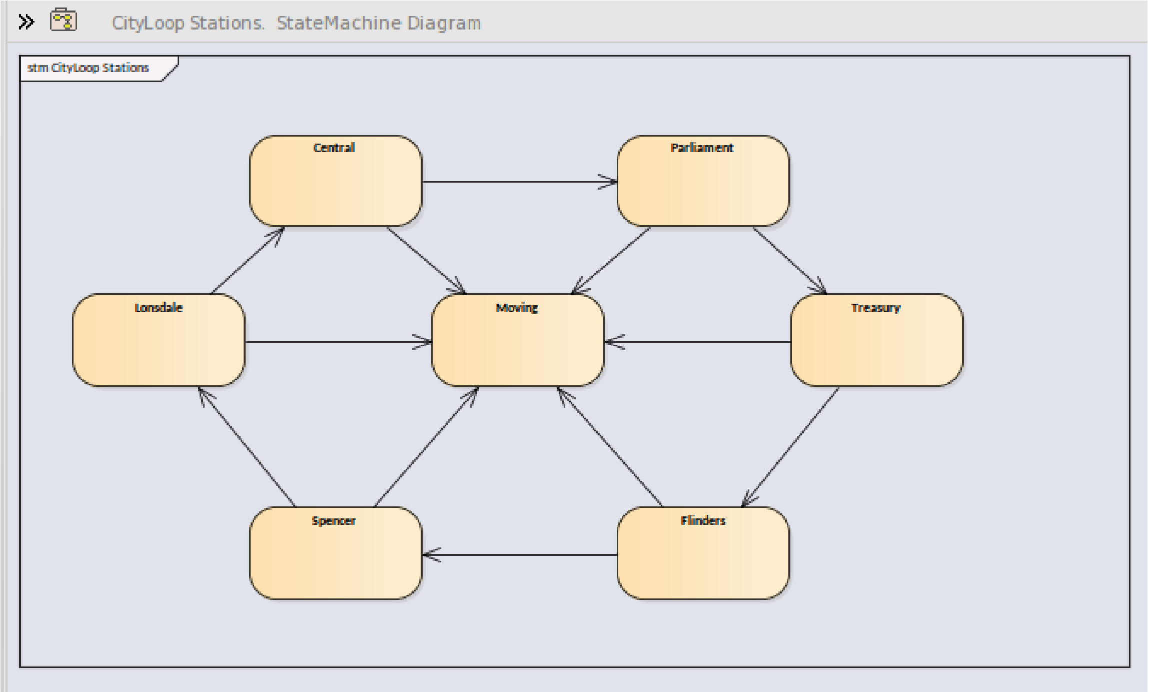
Sequence Recorder State Analyzer

CTrain Location

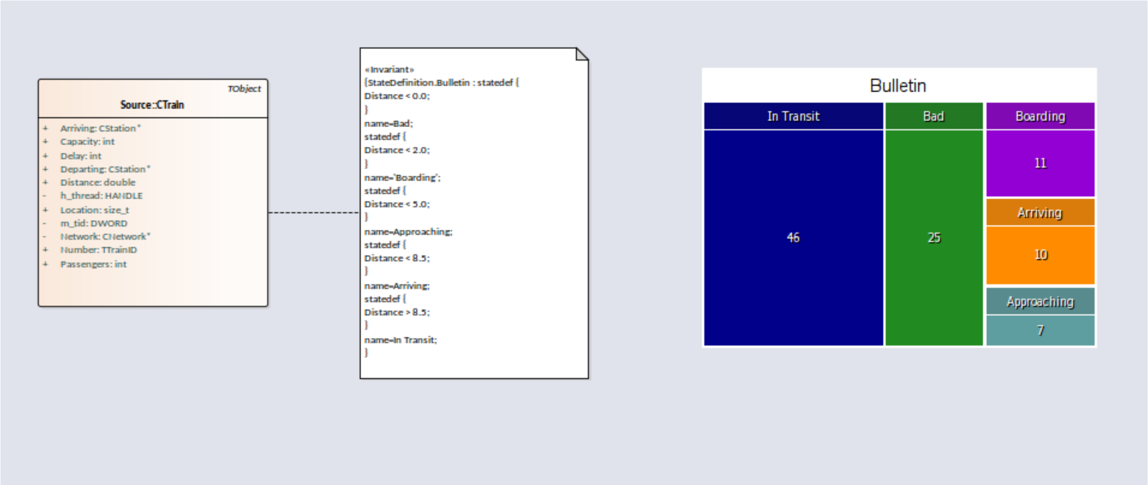
States	Transitions
Moving	
Lonsdale	
	Central
Flinders	
	Spencer
Spencer	
	Lonsdale
Central	
	Parliament
Parliament	
	Treasury
Treasury	
	Flinders

Record & Analyze System Output

The information can be used to create a State Machine.

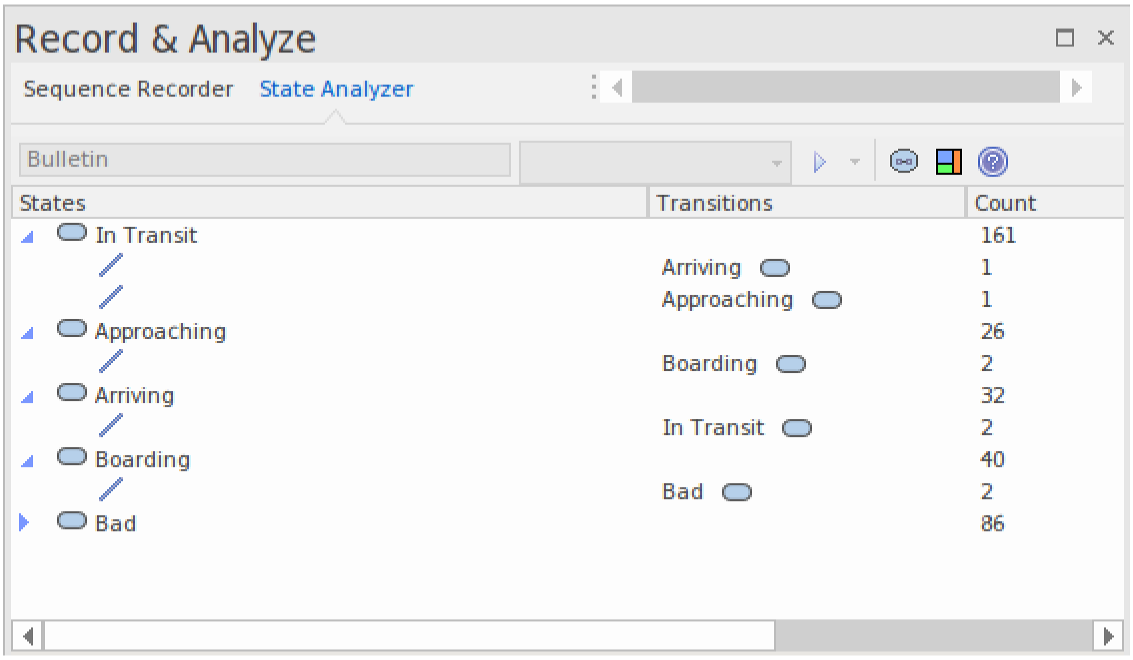


Using the same information we can easily produce a Heat Map. This example shows a 'Train' Class, its 'Bulletin' State Definition (as a linked note), and the Heat Map it produced. The Figures in the map are percentages. From the map we can observe that trains were in the 'In Transit' state 46% of the time.



```
«Invariant»
{StateDefinition.Bulletin : statedef {
Distance < 0.0;
}
name=Bad;
statedef {
Distance < 2.0;
}
name='Boarding';
statedef {
Distance < 5.0;
}
name=Approaching;
statedef {
Distance < 8.5;
}
name=Arriving;
statedef {
Distance > 8.5;
}
name=In Transit;
}
```

This is the analysis for the 'Bulletin' State Definition that produced our Heat Map.



States	Transitions	Count
In Transit		161
	Arriving	1
	Approaching	1
Approaching		26
	Boarding	2
Arriving		32
	In Transit	2
Boarding		40
	Bad	2
Bad		86

Access

Ribbon	Execute > Analyze > Recorder > Open Recorder > State Analyzer Start > Properties > Constraints

State Definitions

State Definitions are composed in the Constraints properties of a Class element. The constraint type should be named *StateDefinition.name*, where 'name' is your choice of title for the definition. These titles are listed in the combo box of the State Analyzer whenever a Class is selected. You select a single definition from this combo box prior to running the program. The State Definition in our example is named 'StateDefinition.Location'. It defines states based on the location of instances of the CTrain Class.

State Definitions are composed of one or more specifications. Each state specification begins with the keyword 'statedef' which is then followed by one or more statements. Statements define the constraints that describe the state, and optionally a variable whose value can be used to name the state. Statements are enclosed in curly brackets and are terminated with a semi colon as shown:

```
statedef {
    Location=0;
    Departing.Name;
}
```

Naming states using variables

In this example, 'Location' is a constant and 'Departing.name' is a variable. An additional statement follows the constraints and instructs the name of the State to be assigned from the variable value. Here is the definition with the naming directive.

```
statedef {
    Location=0;
    Departing.Name;
}
name=Departing.Name;
```

Naming states using literals

In this example the State Definition only contains constants and the state is named using a literal.

```
statedef {
  Location=100;
}
name='Central';
```

A single State Definition defining multiple State specifications.

```
statedef {
  Passengers > 100;
}
name=Busy;
statedef {
  Passengers >= 50;
}
name=Quiet;
statedef {
  Passengers < 50;
}
name=Very Quiet;
statedef {
  Passengers = 0;
}
name=Idle;
```

Default State

A State definition can specify a default 'catch all' state that will describe the state of an instance when no other state holds true. You define a default state for the definition with a statement resembling this:

```
statedef {
    Location=0;
    Departing.Name;
}
name=Departing.Name;
default=Moving;
```

In this example, while execution is in progress any instance detected having a non-zero 'Location' attribute will be recorded as being in the 'Moving' state.

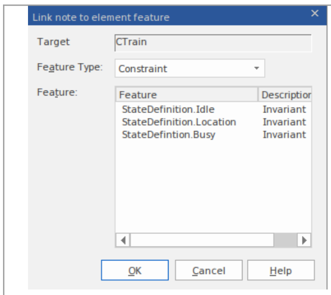
You can choose to exclude the recording of the default state by disabling the 'Include default state' option on the drop down menu of the State Analyzer toolbar. This would exclude transitions to any 'default' state being recorded.

Creating Notes on a Class element that display State Definitions

This section describes how to create the Class diagram that shows all the State Definitions defined for the Class.

Actions

Display a Class diagram	Open an existing Class diagram or create a new one.
Create a link to the Class element	Drag the Class of interest on to the diagram as a link.
Create a note element	Create a note element on the diagram and link it to the class.
Link the note to the State Definition	Select the link between the Note and the Class and, using its context menu, select the 'Link Note to Element Feature' option.
Choose the definition to display on the Note	From the element dialog, choose 'Constraints' from the drop combo. Any defined State Definitions will be listed for you to choose from.

 Link note to element feature Target: CTrain Feature Type: Constraint Feature: Feature / Descriptior StateDefinition.Idle / Invariant StateDefinition.Location / Invariant StateDefintion.Busy / Invariant 	
Repeat	Repeat the procedure for any other State Definitions on the class.

Synchronization

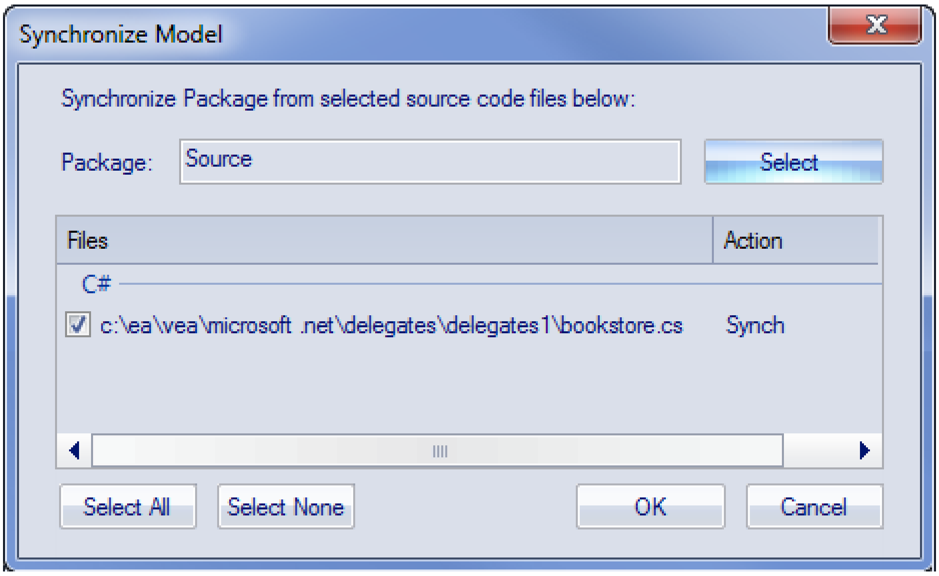
The recording produces a number of assets, the recording history being the main one. Recording also identifies a set of source code files. This set can be used to produce Class and Test Domain diagrams, but can also be used to synchronize your model.

A synchronized model provides quick and accurate navigation between diagram elements and the Class model.

Access

Ribbon	Execute > Analyze > Recorder > Open Recorder > Toolbar button
Context Menu	Right-click on the Record & Analyze window | Synchronize Model with Source Code

Synchronize Model



Synchronize Model

Synchronize Package from selected source code files below:

Package: Source

Files	Action
C#	
☑ c:\ea\vea\microsoft .net\delegates\delegates1\bookstore.cs	Synch

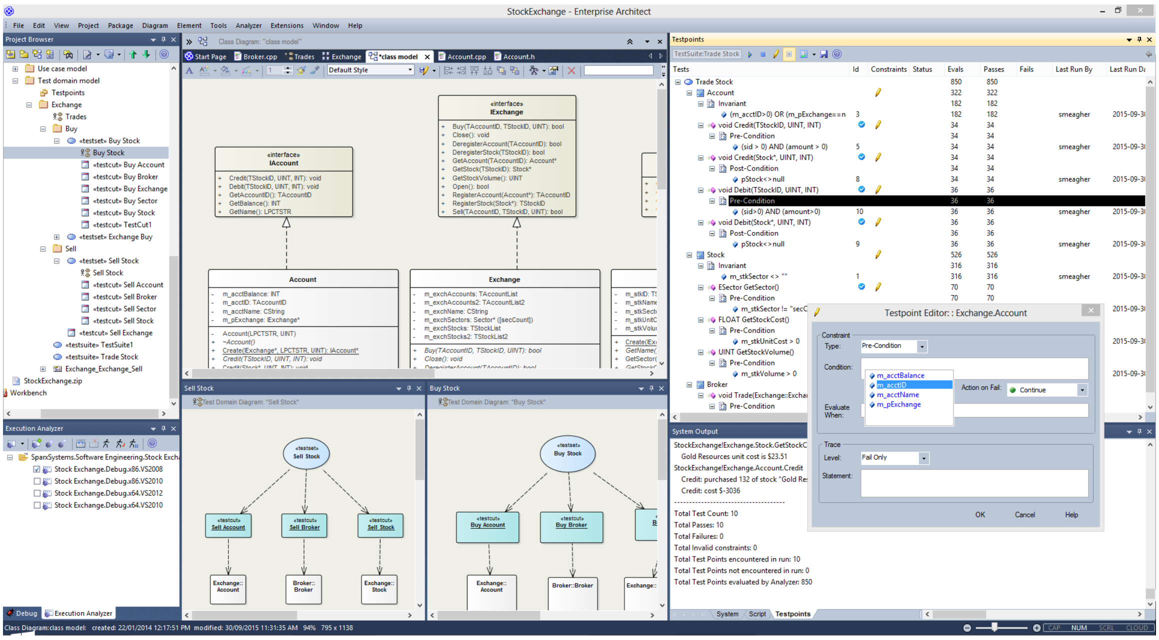
Field/Button	Action
Package	Click on the Select button and select the target Package into which to reverse-engineer the code files.
Files/Action	Lists the files identified during one or more recording(s). The appropriate action is listed next to each file.
Select All	Click on this button to select the checkbox against every file in the 'Files' list.
Select None	Click on this button to clear the checkbox against every file in the 'Files' list.
OK	Click on this button to start the operation. The progress of the synchronization will

	be displayed.
Cancel	Click on this button to abort synchronization and close the dialog.

Testpoints

Testpoints present a scheme by which constraints and rules governing the behavior of objects can be taken from the model and applied to one or more applications. The advantages that schemes such as this offer are tolerance to code changes - adding and subtracting lines from a function has no effect on the constraints that govern it. Another advantage is that changes to the behavioral rules do not require a corresponding change to any source code; *meaning nothing has to be re-compiled!*

Also, the ability to verify multiple applications using a single test domain is a simple rather than onerous matter. The Test Domain is a both a logical and relational model; constraints in the Class model can be partitioned with Test Cuts. These can be aggregated simply into Test Sets and Test Suites using connectors. Due to the decoupling of the Test Domain from the codebase, it is a simple choice of buttons to run a program normally, or run it for a specific Test Domain. This system also delivers practical benefits in that no instrumentation is required at all. Test results are displayed in the report window during the run, in real-time, as the program runs. These results can be retained, and reviewed at any time in the Test Management window Alt+3 or using Enterprise Architect's documentation features.


StockExchange - Enterprise Architect
File Edit View Project Package Diagram Element Tools Analyzer Extensions Window Help
Project Browser
Class Diagram: "class model"
Sell Stock
Buy Stock
Execution Analyzer
Testpoints
TestSuite:Trade Stock
Debug
Execution Analyzer
System
Script
Testpoints

Tests	Id	Constraints	Status	Evals	Passes	Fails	Last Run By	Last Run Da
Trade Stock				850	850			
Account		✎		322	322			
Invariant				182	182			
(m_acctID>0) OR (m_pExchange==n	3			182	182		smeagher	2015-09-3
void Credit(TStockID, UINT, INT)	✓	✎		34	34			
Pre-Condition				34	34			
(sid > 0) AND (amount > 0)	5			34	34		smeagher	2015-09-3
void Credit(Stock*, UINT, INT)	✓	✎		34	34			
Post-Condition				34	34			
pStock<>null	8			34	34		smeagher	2015-09-3
void Debit(TStockID, UINT, INT)	✓	✎		36	36			
Pre-Condition				36	36			
(sid>0) AND (amount>0)	10			36	36		smeagher	2015-09-3
void Debit(Stock*, UINT, INT)	✓	✎		36	36			
Post-Condition				36	36			
pStock<>null	9			36	36		smeagher	2015-09-3
Stock		✎		526	526			
Invariant				316	316			
m_stkSector <> ""	1			316	316		smeagher	2015-09-3
ESector GetSector()	✓	✎		70	70			
Pre-Condition				70	70			
m_stkSector != "secC								2015-09-3
FLOAT GetStockCost()								
Pre-Condition								
m_stkUnitCost > 0								2015-09-3
UINT GetStockVolume()								
Pre-Condition								
m_stkVolume > 0								2015-09-3
Broker								
void Trade(Exchange::Excha								
Pre-Condition								

Testpoint Editor: : Exchange.Account

Constraint
Type: Pre-Condition
Condition:
- m_acctBalance
- m_acctID
- m_acctName
- m_pExchange

Action on Fail: Continue
Evaluate When:

Trace
Level: Fail Only
Statement:

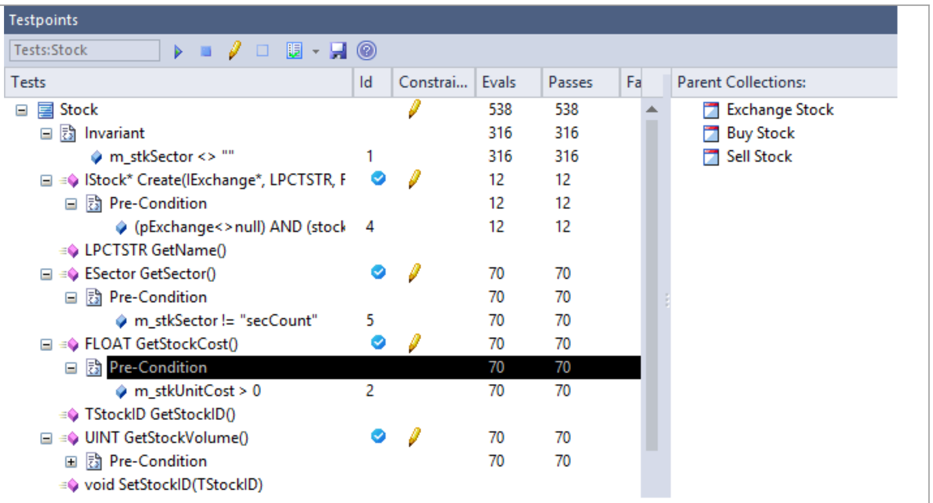
OK Cancel Help

System Output

```
StockExchange!Exchange.Stock.GetStockC
  Gold Resources unit cost is $23.51
StockExchange!Exchange.Account.Credit
  Credit: purchased 132 of stock "Gold Re
  Credit: cost $-3036
-----------------------------------
Total Test Count: 10
Total Passes: 10
Total Failures: 0
Total Invalid constraints: 0
Total Test Points encountered in run: 10
Total Test Points not encountered in run: 0
Total Test Points evaluated by Analyzer: 850
```

Features

Feature	Details
Testpoint Composition	Testpoint composition is performed using the Testpoint Window. The Testpoint Window is context-sensitive and displays the Test Domain for the selected element in either the Project Browser or diagram. Selecting a single Class will display the Class structure. A 'pencil' icon is displayed against Classes and methods that have existing constraints. When you select a Test Cut, Set or Suite Test, the Testpoint window displays the entire Domain structure, including all the Classes that make up the domain. Note: You can navigate the domain hierarchy using the 'Navigation' pane on the right. Testpoints are composed as expressions, using the variable names of the Class members. The Intelli-sense shortcut Ctrl+Space is available within the editor to help you find these. Expressions that evaluate to True are taken to mean a pass. Returning False is taken to mean a fail.

<table>
<tr>
<td></td>
<td>

Testpoints
Tests:Stock
<table>
<tr><th>Tests</th><th>Id</th><th>Constrai...</th><th>Evals</th><th>Passes</th><th>Fa</th><th>Parent Collections:</th></tr>
<tr><td>Stock</td><td></td><td>✎</td><td>538</td><td>538</td><td></td><td>Exchange Stock</td></tr>
<tr><td>Invariant</td><td></td><td></td><td>316</td><td>316</td><td></td><td>Buy Stock</td></tr>
<tr><td>m_stkSector &lt;&gt; ""</td><td>1</td><td></td><td>316</td><td>316</td><td></td><td>Sell Stock</td></tr>
<tr><td>IStock* Create(IExchange*, LPCTSTR, F</td><td>✓</td><td>✎</td><td>12</td><td>12</td><td></td><td></td></tr>
<tr><td>Pre-Condition</td><td></td><td></td><td>12</td><td>12</td><td></td><td></td></tr>
<tr><td>(pExchange&lt;&gt;null) AND (stock</td><td>4</td><td></td><td>12</td><td>12</td><td></td><td></td></tr>
<tr><td>LPCTSTR GetName()</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>ESector GetSector()</td><td>✓</td><td>✎</td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>Pre-Condition</td><td></td><td></td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>m_stkSector != "secCount"</td><td>5</td><td></td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>FLOAT GetStockCost()</td><td>✓</td><td>✎</td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>Pre-Condition</td><td></td><td></td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>m_stkUnitCost &gt; 0</td><td>2</td><td></td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>TStockID GetStockID()</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>UINT GetStockVolume()</td><td>✓</td><td>✎</td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>Pre-Condition</td><td></td><td></td><td>70</td><td>70</td><td></td><td></td></tr>
<tr><td>void SetStockID(TStockID)</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

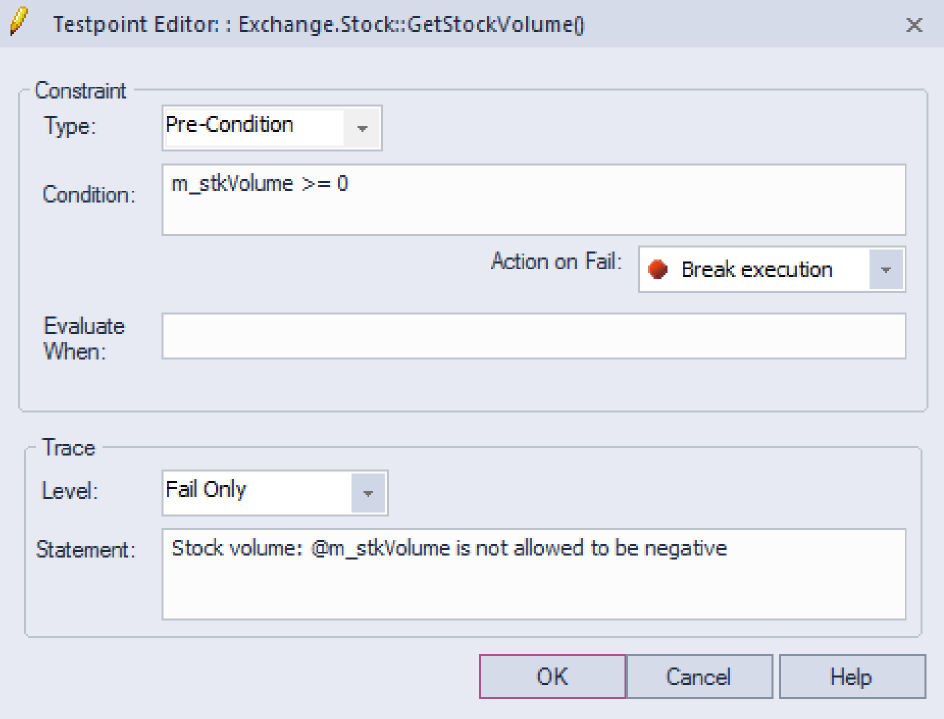
You can add or edit an existing Invariant by double-clicking the Class.

You can add or edit an existing pre- or post-condition similarly by double-clicking the method.

Double-clicking a Testpoint will automatically display the source code if it is available.

Line conditions are best added from within the code editor using its shortcut menus.

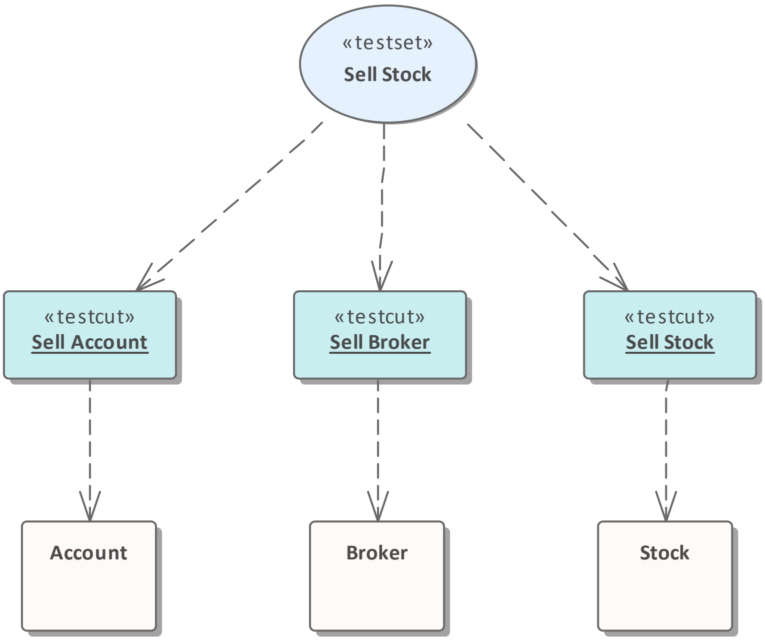
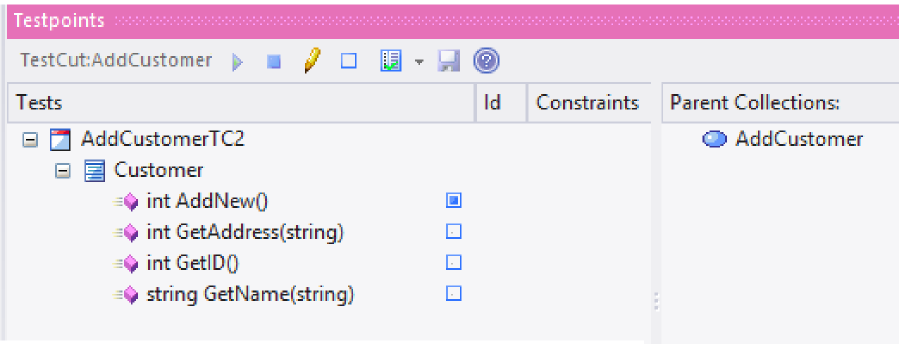
This image is of a pre-condition in the Test domain.


Testpoint Editor: : Exchange.Stock::GetStockVolume()

Constraint
Type: Pre-Condition
Condition: m_stkVolume >= 0
Action on Fail: ● Break execution
Evaluate When:

Trace
Level: Fail Only
Statement: Stock volume: @m_stkVolume is not allowed to be negative

</td>
</tr>
<tr>
<td>Testpoint Trace Statements</td>
<td>Each Testpoint can have its own Trace statement. The Trace statement is a dynamic message that can reference variables in its object or local scope. They are output during the evaluation of a test. They can be configured to be output every time a constraint is evaluated, or more usually when a test has failed. Trace statements can be directed to the 'Testpoints' tab of the System Output Window, or to an external file. You can configure this in any Analyzer Script.</td>
</tr>
<tr>
<td>Test Domain Composition</td>
<td>The Test Domain diagram is a dynamic medium where Testpoints are assembled to test Use Cases. Use Cases in a Test Domain diagram are provided in three different</td>
</tr>
</table>

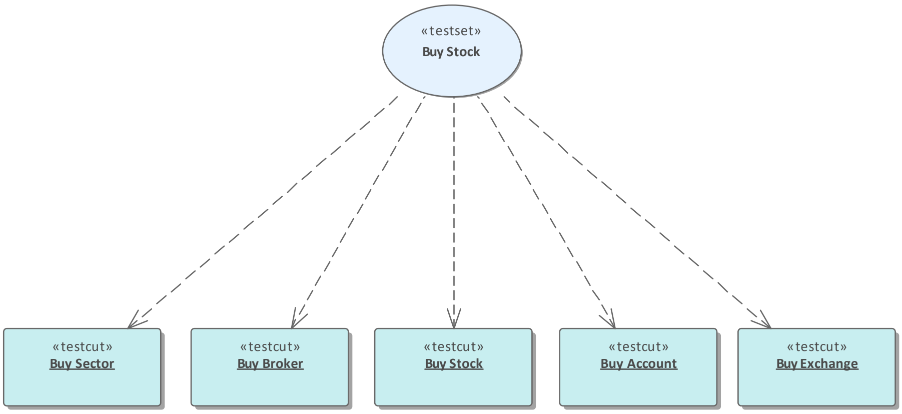
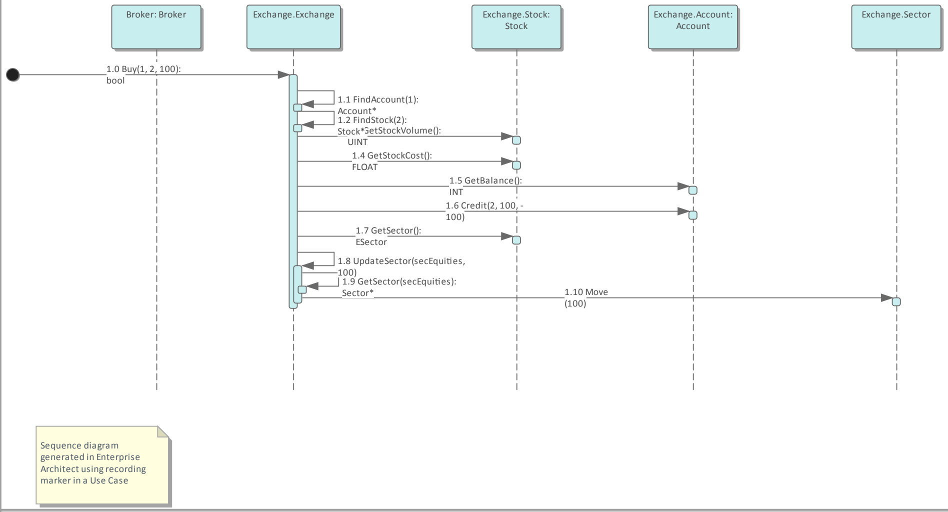
	stereotypes: Test Cut, Test Set and Test Suite. Management of the domain is as easy as modeling on any diagram. The toolbox and shortcut menus provide access to any Test Domain Artifacts. In brief, Testpoints from multiple Classes are aggregated into Test Sets. Test Sets are then linked to form Test Suites. Both Test Cuts and Test Sets are re-usable assets. Linking the same Test Set to the one or more Test Suites is a matter of drawing connectors.  «testset» Sell Stock «testcut» Sell Account «testcut» Sell Broker «testcut» Sell Stock Account Broker Stock
Test Domain and the Class Model	Rarely would a Use Case involve all the methods of a single Class. Most likely it is realized using a variety of methods from collaborating Classes. We call this subset of methods a cut, and the Test Cut Artifact is the tool we use to make these cuts. The Testpoint Window will adapt depending on the context, to be that for a Test Domain or Class element. This image shows the Testpoint window when a Test Cut has been selected. Note the checkboxes, which are only visible for a Test Cut. They denote the methods (Test Cut) which are contributing to a Test Set. In this example the Test domain was generated by the Execution Analyzer, which did the method identification work for us.  Testpoints TestCut:AddCustomer Tests | Id | Constraints | Parent Collections: ⊟ AddCustomerTC2 ⊟ Customer int AddNew() ☑ int GetAddress(string) □ int GetID() □ string GetName(string) □ AddCustomer
Testpoint Evaluation	The Testpoint window is used to evaluate Test domains. The window has a toolbar for starting or attaching to the target application. The domain to test is always reflected by the element that has context, so if you select a Class the window will show only the Class structure and Testpoints of that Class. If you select a Test Suite, the window will display the entire domain hierarchy and all the Testpoints included in it. Clicking on the Run button will load the Testpoint domain in the Execution Analyzer, which will then evaluate, collect and update the report window as Use Cases pass or fail each test. The exact details of each constraint type and the

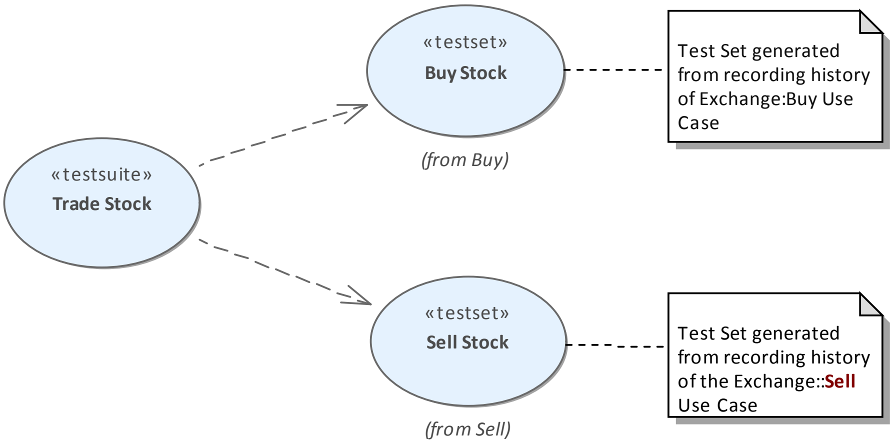
	when and how of that constraint's capture are: • A Class Invariant is evaluated by the Analyzer whenever any method called on an object of this Class type is completed; the invariant serves to test that the state of a complying object is both known and permitted • Pre-conditions are evaluated immediately before an operation is called • Post-conditions are evaluated (at the same time as a Class invariant) when the method is completed • Line-conditions are evaluated if and when their specific line of code comes into scope during program execution

Test Domain Diagram

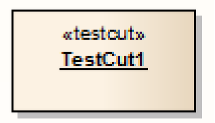
The Test Domain diagram is the medium where you assemble and group test cases for a particular domain. An example of a Test domain might be 'Customer'. The breadth and depth of the domains you assemble is up to you. You might have separate domains for 'Add Customer' and 'Delete Customer', depending entirely on how you consider best to balance the domain hierarchy. The Diagram Toolbox and Shortcut menu provide a number of Artifacts to help model the domain. Because the medium is dynamic, allowing you to revisit and build on relationships between Test domains, the system is a great model for delivering reusable assets to an organization that are low overhead and integrate with both the UML view of the world, and the Software Engineering nuts and bolts of daily life.

Facilities

Facility	Details
Test Domain Generation	If you think the process of composing a Test Domain is complex, it can be, but help is at hand! The Execution Analyzer can produce a Test Domain diagram for you. It cannot write the Tests for you, but it can do some of the leg work. It can identify the Classes and pick out only those methods that participated in a Use Case. And this is not guesswork. The Analyzer Test Domain is obtained from a running program. This image shows the Test Domain generated by the Execution Analyzer from recording an Example Model program.  «testset» Buy Stock «testcut» Buy Sector «testcut» Buy Broker «testcut» Buy Stock «testcut» Buy Account «testcut» Buy Exchange And this is the recording itself (as a Sequence diagram) from which the Test Domain was generated.  Broker: Broker Exchange.Exchange Exchange.Stock: Stock Exchange.Account: Account Exchange.Sector 1.0 Buy(1, 2, 100): bool 1.1 FindAccount(1): Account* 1.2 FindStock(2): Stock* GetStockVolume(): UINT 1.4 GetStockCost(): FLOAT 1.5 GetBalance(): INT 1.6 Credit(2, 100, -100) 1.7 GetSector(): ESector 1.8 UpdateSector(secEquities, 100) 1.9 GetSector(secEquities): Sector* 1.10 Move (100) Sequence diagram generated in Enterprise Architect using recording marker in a Use Case

Test Domain Composition	The first task on a Test Domain diagram is to create the Use Cases (Test Sets). These define this particular domain's responsibility. The Diagram Toolbox and shortcut menu provide Artifacts to help you achieve this. The first of these elements is the Test Cut, which is used in the next step; identifying those methods (from the Class model) that you consider to be participants in the Use Case. The Test Cut Artifact is useful because it allows us to partition a Class, selecting only those methods that are relevant. Test Cuts can be run individually or linked to one or more Test Sets. Test Sets in turn can be linked to one or more Test Suites. In any case, any element of the Test domain tree can be run individually or as a whole.  «testset» Buy Stock (from Buy) Test Set generated from recording history of Exchange:Buy Use Case «testsuite» Trade Stock «testset» Sell Stock (from Sell) Test Set generated from recording history of the Exchange::Sell Use Case

Test Cut



Description

A Test Cut element is a stereotyped Object element, used internal to Enterprise Architect for defining test sets using the Testpoint code testing facilities.

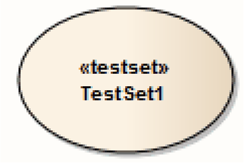
A task, such as 'Print', might involve operations on different Classes. In order to create a 'Print' test, you would want to include only the 'Print' operations of these Classes and exclude any other operations.

A Test Cut enables you to capture only the operations that represent the behavior (in this case, 'Print') defined for a single Class. You might then place the Test Cut from each of several Classes into a single task as a Test Set.

When you drag a Test Cut element onto a Test Domain diagram, you create a Dependency relationship with the required Class element. As a result, when you select the Test Cut element on the Testpoints Window, the operations of the Class are listed in the window, each with a checkbox. You then select the checkbox against each Class operation to include in the Test Cut.

Toolbox icon

Test Set



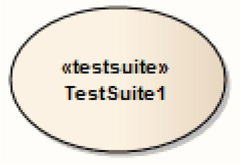
Description

A Test Set element is a stereotyped Use Case element used to aggregate one or more groups of methods (Test Cuts), which perhaps span multiple Classes, into a single task. Test Sets can also be aggregated into Test Suites.

You link the Test Cut elements to the Test Set using Dependency connectors.

Toolbox icon

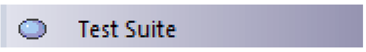
Test Suite



Description

A Test Suite element is a stereotyped Use Case element, used to aggregate one or more groups of tasks (Test Sets). You link the Test Set elements to the Test Suite using Dependency connectors.

Toolbox icon



The Testpoints Window

The Testpoints Window is the hub where Test Domain constraints are composed. It is also the control that lets you verify a particular Test Domain on a program. The program might be already running or it can be launched using the control's Toolbar. Here you will also be able to see the results of your tests, as they happen. This control is context-sensitive, responding to the selection of elements in the Project Browser or on a diagram. Depending on the selection, tests can be carried out on a single class, a Use Case (Test Set) or a collection of Use Cases (A Test Suite).

Access

Ribbon	Execute > Analyze > Testing > Show Testpoints Window

Testpoint Window Columns

Column	Usage
Tests	Displays the name of the selected Testpoint object and the hierarchy of objects beneath it. The selected object can be a: • Class • Operation • Test Cut • Test Set or • Test Suite
Id	For an Operation, this column shows a Testpoint marker icon (✓) when the Analyzer has successfully bound this operation in the target application. If no icon appears in this column during a run, it indicates that the model and code base might not be synchronized; perhaps the signature of the function has changed, or the operation is a new method you are working on that exists in the source code but not yet in your model. For a Testpoint, this column shows a generated id number. This id number is used in trace output to indicate which constraint is being referenced.
Constraints	A pencil icon (✏) in this column indicates that one or more constraints are defined for this Class or Operation.
Status	During a test run, indicates these possible statuses: • (✗) Failed - Constraint has evaluated as false one or more times. • (!) Invalid Statement - Constraint failed to parse due to invalid syntax. • (?) Variable not found - A referenced variable name was not found at the location where the constraint was evaluated. No icon is shown if a constraint has Passed.

Evals	During a test run, indicates the number of times the Execution Analyzer has evaluated this constraint.
Passes	During a test run, indicates the number of times the test passed.
Fails	During a test run, indicates the number of times the test failed.
Last Run By	Displays the username of the last person to run this test. (Values are derived from the Project Author definitions in the 'People' dialog - 'Configure > Reference Data > Project Types > People > Project Authors'.)
Last Run Date	Displays the date and time this test was last evaluated.
Last Run Result	Displays the result of the last test run.
Parent Collections Pane	Lists any parent collections that include the selected object as part of their design. Double-click this collection to make it the selected object in the left pane. The Parent Collections pane can be hidden by clicking the Show / Hide Parent Collections pane button on the Testpoints Window Toolbar.

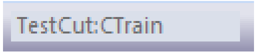
Testpoints Toolbar

The Testpoints Window Toolbar provides options to execute configured tests on the currently selected Testpoint object, stop a test run currently in progress, filter the displayed items, and save the results of a completed test run.

Access

Ribbon	Execute > Analyze > Testing > Show Testpoints Window

Testpoints toolbar options

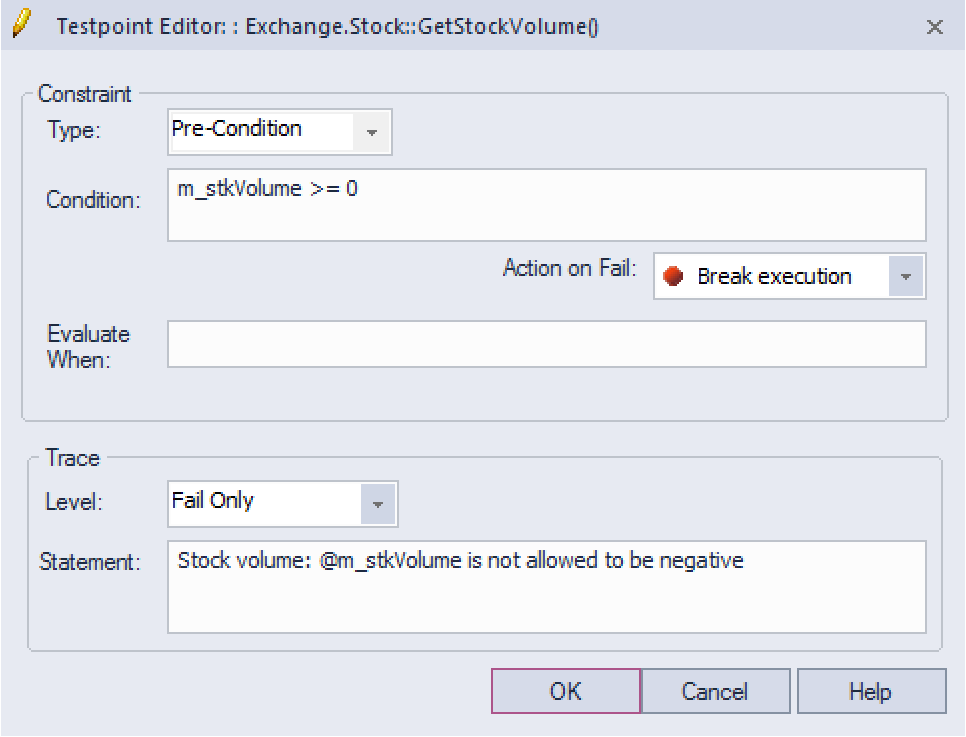
Toolbar Button	Action
 TestCut:CTrain	Field showing the name of the currently selected Testpoint object.
▶	Execute the test run.
■	Stop the test run currently in progress.
✎	Toggle between showing all items and showing only those items that have constraints defined.
▣	Toggle between showing all items and showing only operations that have been marked for inclusion in this Test Cut; this button is only enabled when a Test Cut object is selected. When a Test Cut is selected, each of the operations of its associated Class are displayed with a checkbox; you use this checkbox to mark the operations that apply to this Test Cut.
▤ ▾	Click on the drop arrow next to this icon to display the 'Test Run Options' menu, providing these options: • 'Prefix Trace output With Function Call' - Prefix all trace output lines with the executing function name • 'Enable Standard Breakpoints during Testing' - When not checked, the test run ignores any breakpoints in the current breakpoint set, and any attempts to set breakpoints during the run are ignored • 'View Trace output' - Display the 'Testpoints' tab of the System Output window
💾	Click on this icon after completion of a test run to save the results to Test item on the current object. Saved tests can be viewed using the Testing Workspace. A prompt displays to select the Test Class - Unit, Integration, System, Inspection, Acceptance or Scenario. Select the appropriate Test Class and click on the OK button.

?	Display the Testpoint Management Help topic.
⇨	Show or hide the Parent Collections pane.

Testpoint Editor

The Testpoint Editor is used to compose constraints for Classes and Operations. The types of constraints permitted are dependent on the selected object. For Classes, the type will always be Invariant. For operations, the type can be either Pre-Condition, Post-Condition or Line-Condition.

Invariants are evaluated by the Analyzer when any method called on an object of the selected Class type completes. Pre-conditions are evaluated at the beginning of each call to the specified operation. Post-conditions are evaluated upon completion of each call to the specified operation. Line-conditions are evaluated each time the specified line of code is executed.


Testpoint Editor: : Exchange.Stock::GetStockVolume()

Constraint

Type: Pre-Condition

Condition: m_stkVolume >= 0

Action on Fail: Break execution

Evaluate When:

Trace

Level: Fail Only

Statement: Stock volume: @m_stkVolume is not allowed to be negative

Access

Ribbon	Execute > Analyze > Testing > Show Testpoints Window : Double-click on a Class or Operation in the Testpoints window

Constraint Group fields

Field	Usage
Type	The type of constraint for the selected Class or Operation: • Invariant - Evaluated after any method called on the specified Class has completed • Pre-Condition - Evaluated at the beginning of each call to a specific Operation • Post-Condition - Evaluated after completion of each call to a specific Operation

	• Line-Condition - Evaluated upon execution of a specific line of code within an Operation
Offset	For Line-Conditions only, the Line number within the specified operation upon which to evaluate the constraint. An offset value is automatically set if the Testpoint was created using the Code Editor context menu.
Condition	The constraint to be evaluated when this Testpoint is triggered. A status of pass or fail will be recorded depending upon whether this constraint condition evaluates as true or false.
Action on Fail	Click on the drop-down arrow and select from the three options: • 'Continue' - ignore failure of this constraint and continue execution • 'Break execution' - halt execution and display the Stack trace • 'Disable on fail' - do not execute the constraint again after failing once
Evaluate When	(Optional) An additional constraint which must be met before the main Testpoint Condition is evaluated, providing greater control over test coverage.

Trace Group fields

Option	Action
Level	Specifies when the trace statement (if defined) will be output. Available options are: • 'Fail Only' - Output trace statement only when this Testpoint condition fails • 'Always' - Output trace statement every time this Testpoint is evaluated
Statement	(Optional) A message to be output when this Testpoint is evaluated. Variables currently in scope can be included in a trace statement output by prefixing the variable name with a $ token for string variables, or a @ token for primitive types such as int or long. Output from a Trace Statement can be directed either to the 'Testpoints' tab of the System Output Window, or to an external file, as configured by the Analyzer Script for the parent Package.

Testpoint Constraints

A Constraint is typically composed using local and member variables in expressions, separated by operators to define one or more specific criteria that must be met. A constraint must evaluate as true to be considered as Passed. If a constraint evaluates as false, it is considered as Failed.

Any variables referenced within the constraint must be in scope at the position where the Testpoint or Breakpoint is evaluated.

General/Arithmetic Operators

Operator	Description
+	Add Example: a + b > 0
-	Subtract Example: a - b > 0
/	Divide Example: a / b == 2
*	Multiply Example: a * b == c
%	Modulus Example: a % 2 == 1
()	Parentheses - Used to define precedence in complex expressions. Example: ((a / b) * c) <= 100
[ ]	Square Brackets - Used for accessing Arrays. Example: Names[0].Surname == "Smith"
.	Dot operator - Used to access member variables of a Class. Example: Station.Name == "Flinders"
->	Alternative notation for the Dot operator. Example: Station->Name == "Flinders"

Comparison Operators

Operator	Description
=	Equal To

	Example: a = b
==	Equal To Example: a == b
!=	Not Equal To Example: a != b
<>	Not Equal To Example: a <> b
>	Greater Than Example: a > b
>=	Greater Than or Equal To Example: a >= b
<	Less Than Example: a < b
<=	Less Than or Equal To Example: a <= b

Logical Operators

Operator	Description
AND	Logical AND Example: (a >= 1) AND (a <= 10)
OR	Logical OR Example: (a == 1) OR (b == 1)

Bitwise Operators

Operator	Description
&	Bitwise AND Example: (1 & 1) = 1 (1 & 0) = 0
|	Bitwise OR Example: (1 | 1) = 1

	(1 | 0) = 1
^	Bitwise XOR (exclusive OR) Example: (1 ^ 1) = 0 (1 ^ 0) = 1

Additional Examples

Example	Description
((m_nValue & 0xFFFF0000) == 0)	Use a Bitwise AND operator (&) with a hexadecimal value as the right operand to test that no bits are set in high order bytes of the variable.
((m_nValue & 0x0000FFFF) == 0)	Use a Bitwise AND operator (&) with a hexadecimal value as the right operand to test that no bits are set in low order bytes of the variable.
m_value[0][1] = 2	Accessing a multi-dimensional array
a AND (b OR c)	Combining AND and OR operators, using parentheses to ensure precedence. In this example, variable 'a' must be true, and either 'b' or 'c' must be true.

Notes

- String comparisons are case-sensitive

Object Workbench

The Object Workbench is an Enterprise Architect debugging tool that helps you create objects from your Class model. The Workbench allows multiple instances of any Class to coexist in the same session. Each Object can serve as the target of a method you want to invoke. They can also participate as parameters in methods you invoke. The Object Workbench is supported for the Java and Microsoft .NET platforms.

Workbench Tasks

Task
Provides a guide and the requirements for using the Object Workbench.
Explains what Workbench objects are, and how to create them.
Explains how to execute methods on a Workbench Object and provides information on passing arguments.
Explains stepping through a method's execution using the Debugger.
Explains how to record a method and produce a Sequence diagram.
Explains how to delete a Workbench Object once you are finished with it.
Explains how to shut down the Debugger and close the Workbench once you have finished with it.

Using the Workbench

Using the Object Workbench is straightforward. From your Class model, select the Classes to workbench and drag them individually on to the Workbench window. You might have to choose a constructor if more than one exists, then simply give the variable a name. The Object workbench prepares the required runtime, loads any required modules and instantiates the objects for you. Executing a method is a matter of selecting from a list. Parameters can be entered where required. Workbench objects themselves can be used as parameters either singly or as object arrays.

Access

Ribbon	Execute > Analyze > Testing > Open Object Workbench

Analyzer Script Requirements

An Analyzer Script that has been configured for debugging is required. It should specify this information:

- The debugger to match your project
- For Microsoft .NET, the location of the assembly that will be hosted by the Object Workbench
- For Java, the location of the JDK and additional class paths to use

Checklist

- Select the intended Workbench Class and press F12; the source code should be displayed in a code editor
- Press Shift+F12 to build the project; the output from the build should show successful compilation

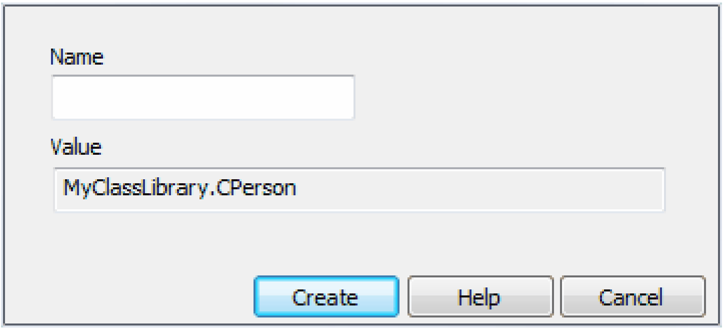
Creating Objects

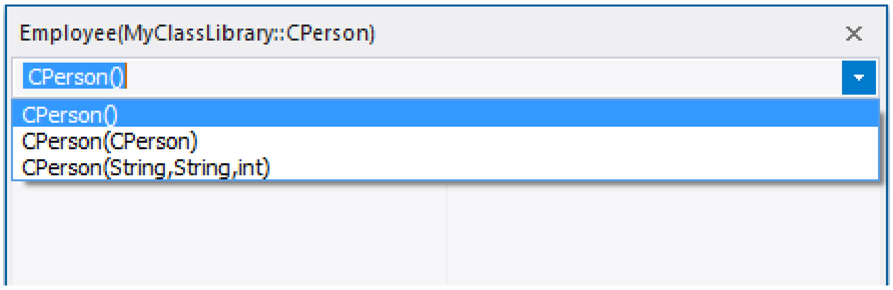
This topic explains how to create a workbench instance for a Class from your model.

Access

Ribbon	Execute > Analyze > Testing > Open Object Workbench
Keyboard Shorcuts	Ctrl+Shift+J
Other	Drag a Class directly from the Project Browser onto the Workbench window

Tasks

Task	Detail
Creating an Object on the Workbench	Select the Class in the Project Browser and drag it on to the Workbench window. The 'Workbench' dialog displays.  Name: Value: MyClassLibrary.CPerson Type in a name for the new instance. The name should be unique for the Workbench. Click on the Create button.

Choosing a Constructor	The 'Constructor' dialog is displayed where a choice of constructor exists.
	
	Employee(MyClassLibrary::CPerson)
	CPerson()
	CPerson()
	CPerson(CPerson)
	CPerson(String,String,int)
	Select the constructor from the drop-down list.

Enter Parameters	Provide values for the selected constructor's parameters:

	• Strings as arguments - Surround values with quotes where appropriate, or where the value would conflict with the name of a Workbench object • Objects as arguments - Enter the name of the Workbench object • String array arguments take text values separated by commas: one,two,three,"a book","a bigger book" • Object arrays as arguments take object names separated by commas; supply the named Workbench objects separated by commas, for example: Tom,Dick,Harry
Invoke Constructor	Click on the Invoke button to create the instance. The object can be recognized by its name in the Workbench window.

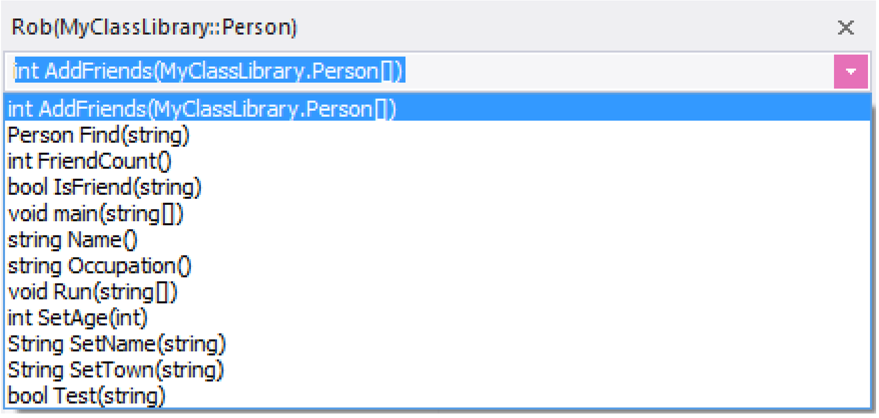
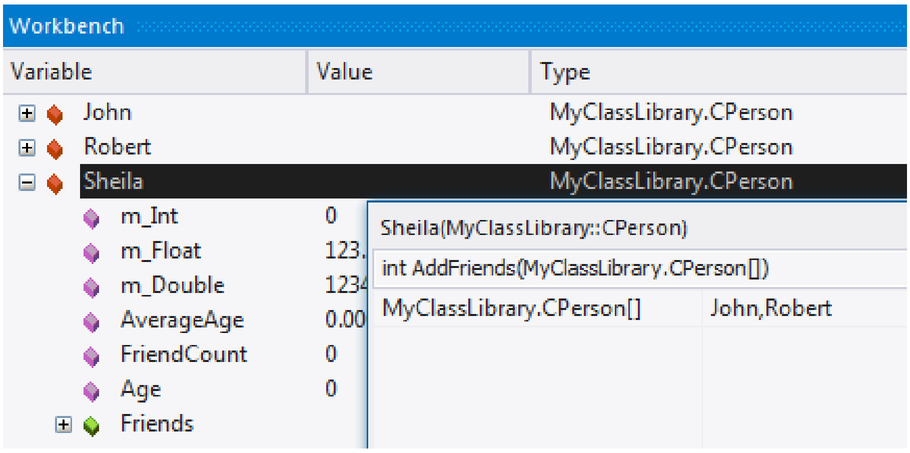
Invoking Methods

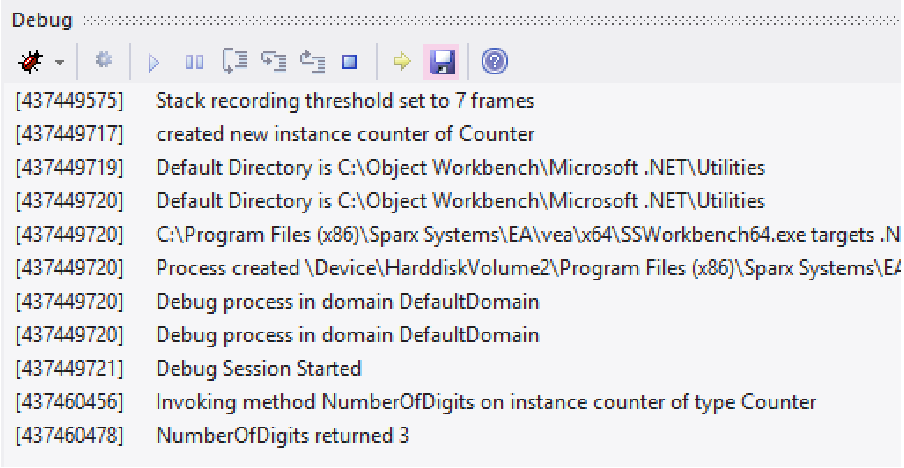
Access

On the Workbench window, right-click on the instance on which to execute a method, and select 'Invoke'.

Ribbon	Execute > Analyze > Testing > Open Object Workbench

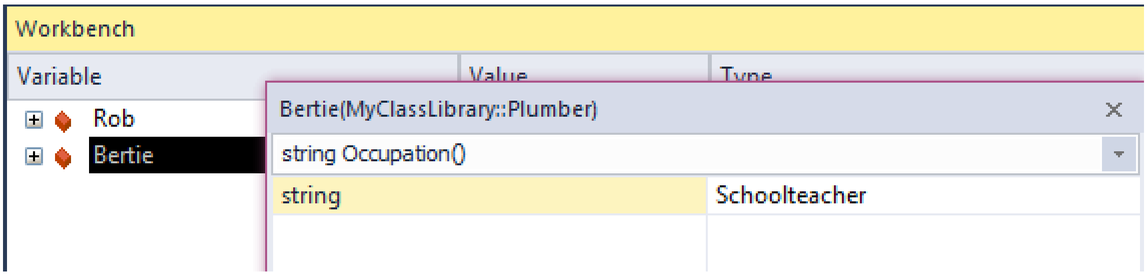
Actions

Action	Details
Choose Method	Select a method from the list and click on the Invoke button. Note that all methods listed are public; private methods are not available.  Rob(MyClassLibrary::Person) int AddFriends(MyClassLibrary.Person[]) int AddFriends(MyClassLibrary.Person[]) Person Find(string) int FriendCount() bool IsFriend(string) void main(string[]) string Name() string Occupation() void Run(string[]) int SetAge(int) String SetName(string) String SetTown(string) bool Test(string)
Provide Arguments	In this image, the method to be invoked takes an array of objects as its only argument. You construct this argument by naming the other instances on your Workbench that you want to pass to the method.  Workbench Variable | Value | Type John | | MyClassLibrary.CPerson Robert | | MyClassLibrary.CPerson Sheila | | MyClassLibrary.CPerson m_Int | 0 m_Float | 123. m_Double | 1234 AverageAge | 0.00 FriendCount | 0 Age | 0 Friends Sheila(MyClassLibrary::CPerson) int AddFriends(MyClassLibrary.CPerson[]) MyClassLibrary.CPerson[] | John,Robert
Argument Types	These are the parameter types supported by the Workbench: • Strings • Numbers

	- Objects - String Arrays - Object Arrays
Argument Syntax	- Strings as arguments - Surround strings with quotes where necessary; for example, to avoid conflict with Workbench object names - String Arrays as arguments - Enter the elements that compose the array, separated by commas; for example: "A maths book","A geography book","A computer book" - Objects as arguments - Type the Workbench object name as the argument; the debugger checks any name entered in an argument against its list of Workbench instances, and will substitute that instance in the actual call to the method - Object Arrays as arguments - Enter the Workbench objects' names to satisfy the argument, separated by commas: Tom,John,Peter
Invoke	Click on the Invoke button to execute the method. Output confirming this action is displayed in the Debug window.  [437449575] Stack recording threshold set to 7 frames [437449717] created new instance counter of Counter [437449719] Default Directory is C:\Object Workbench\Microsoft .NET\Utilities [437449720] Default Directory is C:\Object Workbench\Microsoft .NET\Utilities [437449720] C:\Program Files (x86)\Sparx Systems\EA\vea\x64\SSWorkbench64.exe targets .N [437449720] Process created \Device\HarddiskVolume2\Program Files (x86)\Sparx Systems\E [437449720] Debug process in domain DefaultDomain [437449720] Debug process in domain DefaultDomain [437449721] Debug Session Started [437460456] Invoking method NumberOfDigits on instance counter of type Counter [437460478] NumberOfDigits returned 3

Setting Properties

For languages that support properties, we can set the value of an Object's property in the same manner in which we invoke a method. Select the instance in the Workbench, and use its context menu to select the 'Invoke' option. You will find the properties exposed by the Class listed alphabetically, along with its methods. You will be prompted to provide the new value of the property. Type the value as you would have entered it for the parameter in a method call. This image demonstrates changing the *Occupation* property of a *Person* called *Bertie; Bertie* being a type of *Person.*

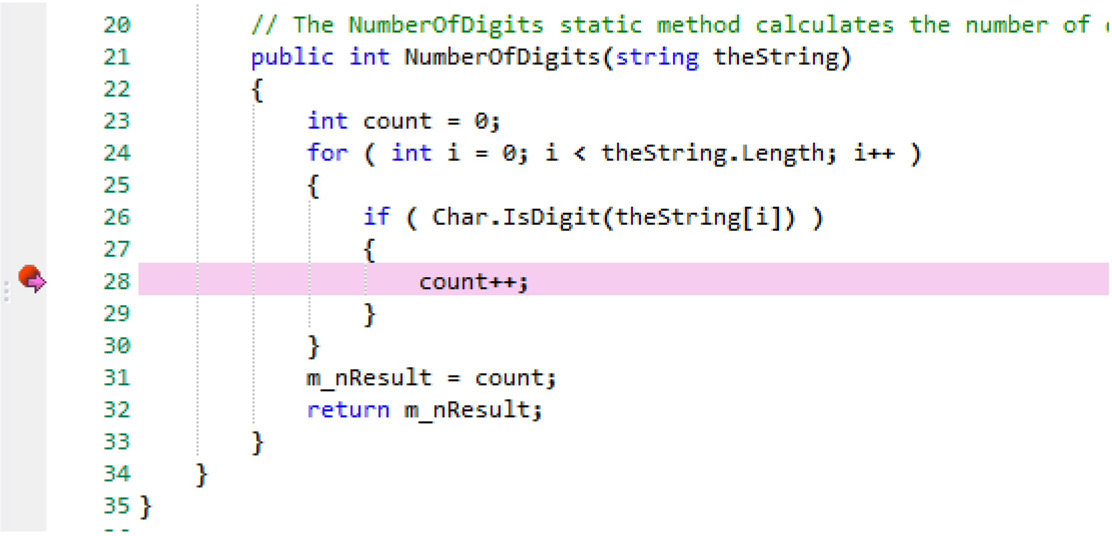


string Occupation()

string	Schoolteacher

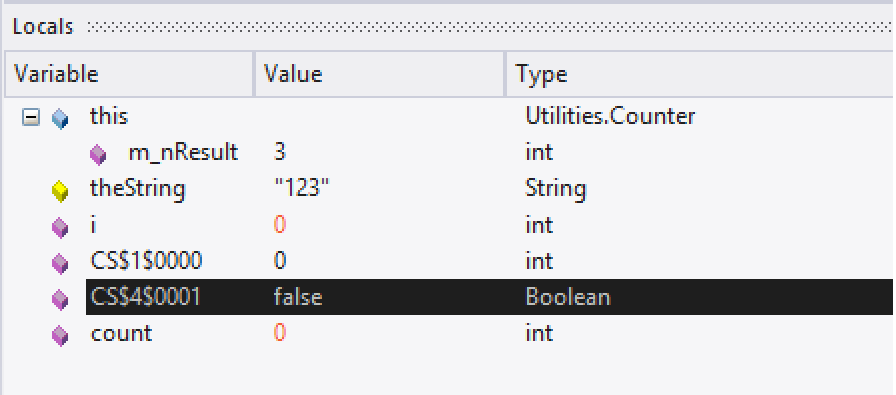
Debugging and the Workbench

While you are working in the Workbench, you might want to debug one or more methods you are developing or investigating. This can easily be accomplished. The same features of Enterprise Architect's Execution Analyzer are available to users of the Object Workbench. Debugging can be performed during object construction and destruction as well as during the execution of a method. To gain access to the debugger, simply place a breakpoint at the points at which to step through the code. You could also set the condition on these breakpoints to only break under certain conditions.



```
        // The NumberOfDigits static method calculates the number of
        public int NumberOfDigits(string theString)
        {
            int count = 0;
            for ( int i = 0; i < theString.Length; i++ )
            {
                if ( Char.IsDigit(theString[i]) )
                {
                    count++;
                }
            }
            m_nResult = count;
            return m_nResult;
        }
    }
}
```

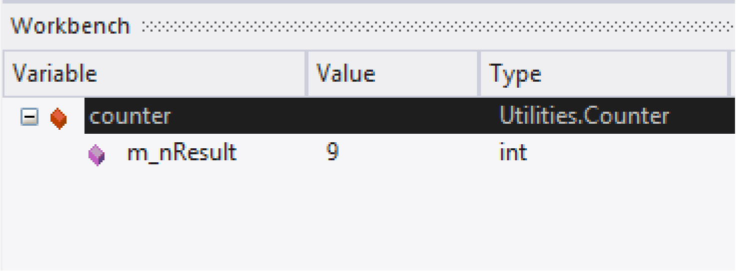
When debugging, the states of objects are inspected using the debugger controls. Here we use the Locals window to examine the state of our object while execution is halted.



Locals

Variable	Value	Type
this		Utilities.Counter
m_nResult	3	int
theString	"123"	String
i	0	int
CS$1$0000	0	int
CS$4$0001	false	Boolean
count	0	int

When the program resumes, the Object on the Workbench will reflect any changes in state.



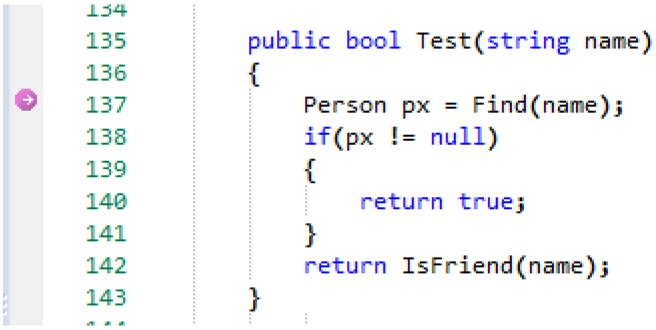
Workbench

Variable	Value	Type
counter		Utilities.Counter
m_nResult	9	int

Recording and the Workbench

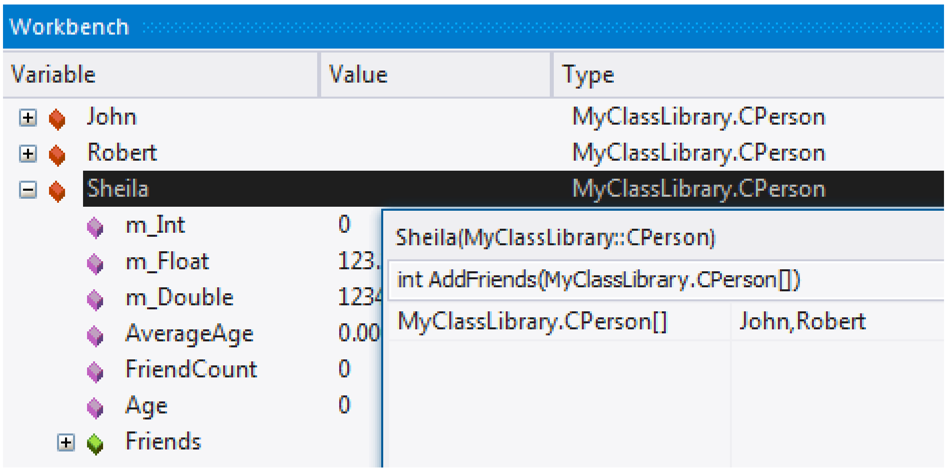
While you are working in the Workbench, you might want to produce a Sequence diagram for one or more methods you are developing or investigating. This can easily be accomplished. The same features of Enterprise Architect's Execution Analyzer are available in the Object Workbench. You might even begin a Workbench session by recording a Sequence diagram first off, as a means of visualizing what you plan to work on.

Set the Recording Marker



```
public bool Test(string name)
{
    Person px = Find(name);
    if(px != null)
    {
        return true;
    }
    return IsFriend(name);
}
```

Invoke the Method



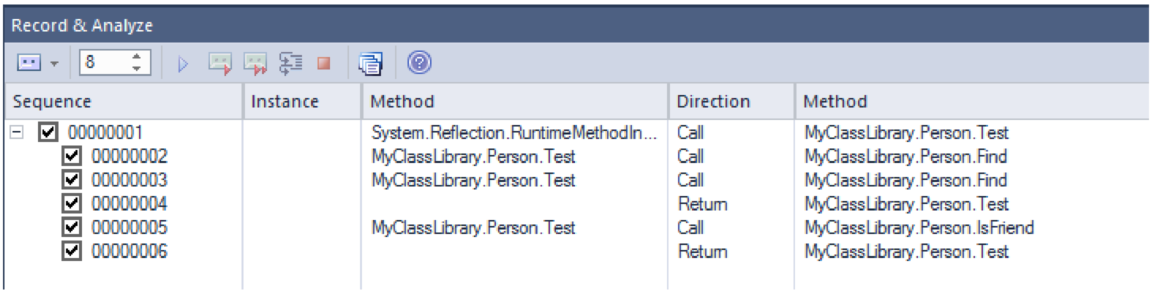
Variable	Value	Type
⊞ John		MyClassLibrary.CPerson
⊞ Robert		MyClassLibrary.CPerson
⊟ Sheila		MyClassLibrary.CPerson
m_Int	0	
m_Float	123.	
m_Double	1234	
AverageAge	0.00	
FriendCount	0	
Age	0	
⊞ Friends		

Sheila(MyClassLibrary::CPerson)

int AddFriends(MyClassLibrary.CPerson[])

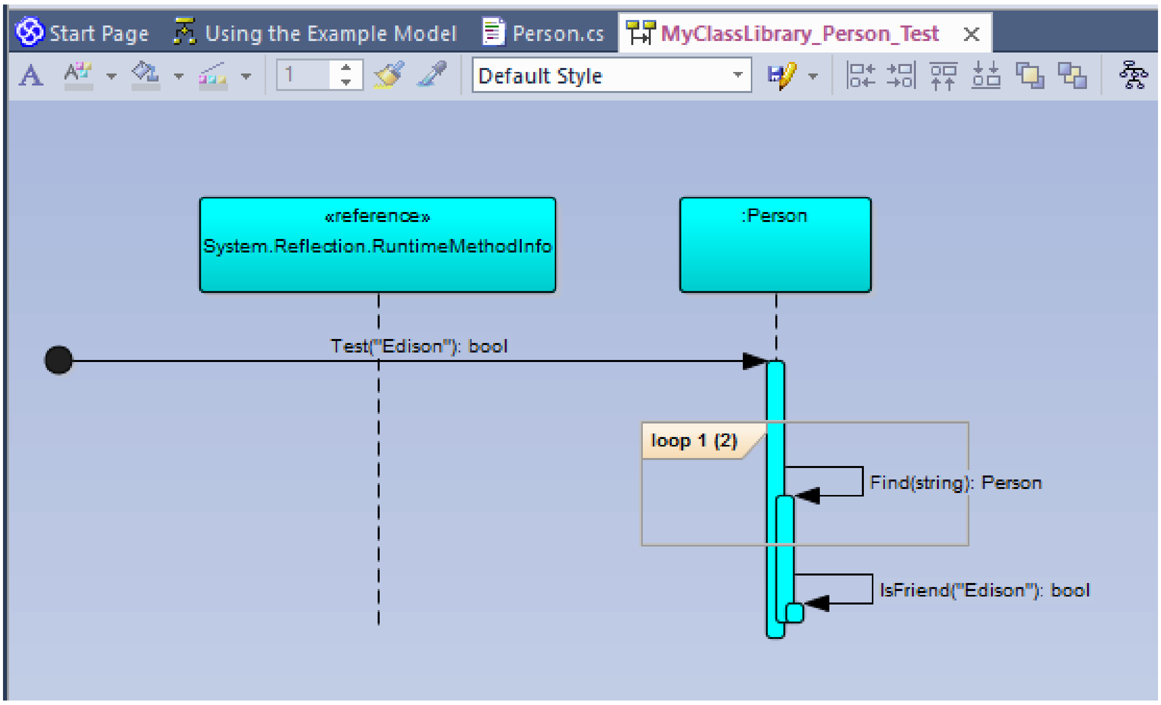
MyClassLibrary.CPerson[]	John,Robert

View the Recording History



Sequence	Instance	Method	Direction	Method
⊟ ☑ 00000001		System.Reflection.RuntimeMethodIn...	Call	MyClassLibrary.Person.Test
☑ 00000002		MyClassLibrary.Person.Test	Call	MyClassLibrary.Person.Find
☑ 00000003		MyClassLibrary.Person.Test	Call	MyClassLibrary.Person.Find
☑ 00000004			Return	MyClassLibrary.Person.Test
☑ 00000005		MyClassLibrary.Person.Test	Call	MyClassLibrary.Person.IsFriend
☑ 00000006			Return	MyClassLibrary.Person.Test

Generate the Sequence Diagram



Deleting Objects

You can easily delete an object by selecting it in the Workbench, right-clicking on it and selecting the 'Delete' option.

Closing the Workbench

To shut down the Workbench perform any of these actions:

- Choose 'Reset' from the Object Workbench context menu
- Press the Stop button on any debugger toolbar
- Delete all objects on the Workbench

Visualize Run State

You can record the state transitions of a single object by taking multiple snapshots of the object's run state at key points in its lifetime. To do this simply drag the local or member variable on to an Object diagram.

Unit Testing

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

In sequence:

- You download and install the NUnit and JUnit applications (JUnit - http://www.junit.org/ NUnit - http://www.nunit.org/index.php?p=home); Enterprise Architect does not include these applications in the installer
- Enterprise Architect helps you to create test Class stubs with the JUnit and NUnit transformations
- You define your test code within the Class stubs
- You set up and run a test script against any Package
- All test results are automatically recorded inside Enterprise Architect

Set Up Unit Testing

This topic explains the actions you should take in setting up Unit Testing, after having downloaded and installed the JUnit and/or NUnit applications.

Actions

Action	Details

Create Unit Test Stubs	By using the JUnit or NUnit transformations and code generation you can create test method stubs for all of the public methods in each of your Classes.
	(TestFixture)
	public class CalculatorTest
	{
	    (Test)
	    public void testAdd(){
	    }
	    (Test)
	    public void testDivide(){
	    }
	    (Test)
	    public void testMultiply(){
	    }
	    (Test)
	    public void testSubtract(){
	    }
	}

Define Test Cases	Write your unit test in the generated code stubs (either in Enterprise Architect or in your preferred IDE).
	This is an NUnit example in C#, although it could also be any other .NET language, or Java and JUnit.
	(TestFixture)
	public class CalculatorTest
	{
	    (Test)
	    public void testAdd(){
	        Assert.AreEqual(1+1,2);
	    }
	    (Test)
	    public void testDivide(){
	        Assert.AreEqual(2/2,1);
	    }

	(Test)
	public void testMultiply(){
	Assert.AreEqual(1*1,1);
	}
	(Test)
	public void testSubtract(){
	Assert.AreEqual(1-1,1);
	}
	}
	
	Alternatively, if you have not performed an xUnit transformation, you can reverse engineer the code into Enterprise Architect so that the system can record all test results against this Class.

Compile Your Code	Check that the source code being tested compiles without errors, so that the test scripts can be run against it.
Set up the Test Scripts	Set up the Test scripts against the required Package, and then run the tests.

Run Unit Tests

On running a test script you generate test results that are stored as Test Cases against the Classes being tested.

Access

Ribbon	Code > Build and Run > Test
Execution Analyzer Window	ToolBar > Run Test Script

Tasks

Task	Details
Run Tests	Select the appropriate Package in the Project Browser. Select the 'Run Test Script' option to run the test script you previously set up for that Package, in the Execution Analyzer.
View Results	The results of xUnit tests are displayed in the System Output window, identifying which tests have run and which of these have failed. The results also show which method failed, and the file and line number the failure occurred at. Double-click on an error message; Enterprise Architect opens the editor to that line of code, enabling you to quickly find and fix the error. Enterprise Architect also records the run status of each test against the Class being tested; these are stored in the element Test Cases. A diagram containing the Class can be set to display these Test Cases, by exposing the test scripts compartment on the diagram elements.

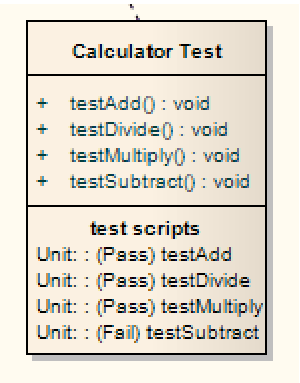
Record Test Results

Enterprise Architect is able to automatically record all results from tests through a testing script in Enterprise Architect.

Process

In order to use this feature, you must reverse engineer the test Class into the Package containing your test script.

Once your model contains your test Class, on the next run of the test script Enterprise Architect adds Test Cases to the Class for each test method found; on this and all subsequent test runs all Test Cases are updated with the current run time and whether they passed or failed, as shown:



The error description for each failed test is added to any existing results for that Test Case, along with the current date and time.

Over time this provides a log of all test runs where each Test Case has failed, which can then be included in generated documentation, resembling this:

Failed at 05-Jul-2006 1:02:08 PM

expected: <0>

but was: <1>

Failed at 28-Jun-2006 8:45:36 AM

expected: <0>

but was: <2>

Samples

Enterprise Architect enables you to easily import complete sample models (Packages), including all necessary model information, code and build scripts. These sample Patterns make it simple to explore and try out the Visual Execution Analyzer. You can generate an example model for:

- Java
- Microsoft.NET
- Microsoft C++
- PHP Apache

Access

Ribbon	Design > Package > Insert > Insert using Model Wizard > VEA Examples
Context Menu	Right-click on Package | Add a Model Using Wizard > VEA Examples
Keyboard Shortcuts	Ctrl+Shift+M > VEA Examples
Other	Project Browser caption bar menu | New Model from Pattern > VEA Examples

Display Samples

Field	Action
Technology	Select the appropriate technology.
Name	Displays the samples available for the selected technology; select the required sample to import.
description field	Displays a description of the selected sample.
Destination folder	Browse for and select the directory in which to load the source code for the sample.
Use Local Path	Enable the selection of an existing local path to place the source code under; changes the 'Destination folder' field to a drop-down selection.
Compiler command	Displays the default compiler command path for the selected technology; you must either: • Confirm that the compiler can be found at this path, or • Edit the path to the compiler location
Edit Local Paths	Many VEA examples specify their compiler using a local path. The first time you use any sample you must click on this button to display the 'Local Paths' dialog, on which you check and - if necessary - correct the local path pointing to the correct compiler location.

Notes

- If required, you can define custom samples by adding files to the AppSamples directory in which Enterprise Architect is installed; top-level directories are listed as Technologies and can contain an icon file to customize the icon displayed for the technology
  Directories below this are defined as groups in the Patterns list; the Patterns are defined by the presence of four files with a matching name: a zip file (.zip), XMI file (.xml), config file (.cfg) and optional icon (.ico)
- The config file supports these fields:
  - [provider], [language], [platform], [url], [description], [version] - all displayed in the 'description' field
  - [xmirootpaths] - the root path of the source code in the exported XMI; this is replaced with the selected destination folder when the user applies the application pattern