Model Driven Development and Visual Execution Analysis

All material © Sparx Systems 2012

http://www.sparxsystems.com
Table of Contents

INTRODUCTION.............................................................................................................................................. 4

OVERVIEW ..................................................................................................................................................... 4
Section 1: Code, Build and Debug Cycles ...................................................................................................... 4
Section 2: The Record, Analyze & Refine Cycles .......................................................................................... 4
Section 3: Testing Cycle .................................................................................................................................. 5

SECTION 1 - WRITE, BUILD AND DEBUG CODE ......................................................................................... 6

CHAPTER 1 - THE GENERAL PROCESS ....................................................................................................... 6
Write, Build and Debug Process .................................................................................................................. 6
Enterprise Architect Links to the Compiler .................................................................................................... 7
Prerequisites for Coding Building and Debugging ............................................................................................ 7

CHAPTER 2 – WORKING WITH THE CODE .................................................................................................. 9
Establish your code ........................................................................................................................................ 9
Code Editing .................................................................................................................................................. 9
Customize Workspace Layouts for Debugging ................................................................................................. 11

CHAPTER 3 – CONFIGURING BUILD AND DEBUG IN ENTERPRISE ARCHITECT ...................................... 13
Setting up an Analyzer Script ....................................................................................................................... 13
Analyzer Scripts ........................................................................................................................................... 13
Setting up Linking for Builds (compilations) .................................................................................................. 14
Locating Compile Errors in Code .................................................................................................................. 16
Setup for Debugging .................................................................................................................................... 16

CHAPTER 4 – DEBUGGING FROM THE MODEL ........................................................................................... 18
Compile for Debug .......................................................................................................................................... 18
Setting Breakpoints ...................................................................................................................................... 18
Stepping through Code – the Debug view ....................................................................................................... 20

CHAPTER 5 – USING THE DEBUG VIEWS ...................................................................................................... 21
Locals View .................................................................................................................................................... 21
Call Stack View ............................................................................................................................................ 23
Watches View ................................................................................................................................................ 23
Search ........................................................................................................................................................... 25
Modules View .............................................................................................................................................. 26
Memory View ............................................................................................................................................... 26
Output View .................................................................................................................................................. 26
Debug View .................................................................................................................................................. 27
Tips on using Views ....................................................................................................................................... 27

SECTION 2 – VISUAL EXECUTION ANALYSIS ............................................................................................ 30

CHAPTER 6 - SEQUENCE DIAGRAM ANALYSIS .......................................................................................... 32
Efficiency Example ....................................................................................................................................... 32
Flow Example ............................................................................................................................................... 34
Setup for Generating Flow Sequence diagrams ............................................................................................. 35

CHAPTER 7 – GENERATED SEQUENCE DIAGRAMS SHOWING STATE TRANSITIONS .......................... 38
Setup for Generating Call Stack Sequence Diagrams .................................................................................... 40

CHAPTER 8 – PROFILING AND WORKBENCH ......................................................................................... 41
Profiling Native Applications .......................................................................................................................... 41
Workbench .................................................................................................................................................... 42

SECTION 3 - TESTING .................................................................................................................................. 44

CHAPTER 9 - TESTPOINTS ............................................................................................................................ 45
Operation Constraints (Contract Conditions) .................................................................................................. 46
Setting up a set of Testpoints .......................................................................................................................... 48
Execution of Testpoints .................................................................................................................................. 49
Combining Test Sets ..................................................................................................................................... 50
Output Options ............................................................................................................................................... 51
# CHAPTER 10 - UNIT TESTING

1. **Unit Test Setup** ........................................................................................................... 52
2. **Setup the 3rd party xUnit package** .............................................................................. 53
3. **Generate skeletal code for the xUnit testing** ............................................................... 53
4. **Insert xUnit Code** ....................................................................................................... 54
5. **Run the Test scripts** .................................................................................................... 54
6. **View test results** ........................................................................................................... 55

# CONCLUSION .................................................................................................................. 55

# APPENDIX ....................................................................................................................... 57

# REFERENCES: .................................................................................................................. 57
Introduction

When developing software using a traditional Integrated Development Environment (IDE) your scope is limited to a single coding tool and typically a single phase of the software development life cycle, which disconnects the code from the design specifications. Rather than needing separate tools for design and development, Enterprise Architect provides all the facilities to design models, develop code from these models and then test and analyze these systems within a single development environment.

Compared to the broader life-cycle of model-based application development, debugging and dynamic analysis of program behavior are short cycles. However, these are intensive and critical aspects of the model driven development life-cycle, and can be of high cost in terms of human resources where the tools are not sufficient for the job.

Enterprise Architect allows you to integrate your model with each and every phase of the software development cycle supporting true Model Driven Development (MDD).

The purpose of this paper is to examine the development of software systems, looking specifically at Enterprise Architect’s features for the code, build, debug and testing cycles, along with features for analyzing and refining the code developed in these cycles.

Overview

The processes for building a software intensive system are tightly connected. For clarity however, we will cover these processes in three separate sections:

Section 1: Code, Build and Debug Cycles

This development cycle has short intense cycles involving coding, compiling and debugging. Enterprise Architect provides comprehensive facilities for running through the standard debug cycle that is usually done within the IDE. This section outlines the benefits of running the processes in this cycle from within the model and covers:

- Code editing:
  This includes standard code editing like ‘Syntax highlighting’ and ‘Intellisense’, auto-indenting, as well as, Go To Declaration and Go To definition for most of the key code languages.

- Connectivity to the Compiler:
  This outlines direct linking for compiling and catching compilation errors.

- The Debug facilities:
  This covers setting break points and checking execution data via views.

Section 2: The Record, Analyze & Refine Cycles

The Record, Analyze and Refine cycles are for refining the correctness and the efficiency of the code to meet both its functional and performance requirements. This involves monitoring the sequence flows, the execution timing and the resource usage. Enterprise Architect provides tools for these analysis activities within its debugging capability. This provides direct means to refine the code in the modeling environment. The analysis tools provide:

- A qualitative assessment of how the code is executed
- A quantitative assessment of the performance of the code
For the **qualitative** assessment there are the **Visual Execution** tools for creating Sequence Diagrams. These help identify errors in logic, explain unexpected system behavior and identify data flow inconsistencies.

The tools available for **quantitative** assessment are the **Profiling** and **Workbench** tools. These are useful for monitoring runtime system usage.

### Section 3: Testing Cycle

Underlying both of the above cycles is the requirement for the developer to define and implement code testing, as well as return test results. Enterprise Architect provides comprehensive support for two types of code based testing. These are as follows:

#### Testpoints

Although similar to xUnit testing, Test Points provide support for “Programming by Contract Principles”. As they are defined as statements that evaluate in the Debug framework, unlike xUnit testing, these can be run without having to write separate code files. This means they can be stored, updated and run directly within the model.

#### xUnit Test Classes

Enterprise Architect supports the creation of xUnit Classes, either manually or generated using MDA transforms. These can be detailed with code, then compiled and run, with the unit-test results being stored against the tested Class.

The purpose of examining these three phases of the software development life cycle is to clarify the tools available in Enterprise Architect and methods for using these features for developing a system from a model through to executable code.
Section 1 - Write, Build and Debug Code

The following section examines Enterprise Architect’s features that support software development, with a specific focus on writing, building and debugging the code in the modeling environment. This section describes how Enterprise Architect links to different compilers in order to offer greater flexibility in the choice of languages and software development options available to you as a developer. In summary this section covers Enterprise Architect’s features for:

1. Developing the code derived from the model or reverse engineered legacy code.
2. Using the context sensitive code editing environment.
3. Debugging from the model by linking to a variety of popular compilers.

Chapter 1- The General Process

This chapter begins by describing the standard processes for writing, compiling and debugging your code with reference to the chapters in this section covering the facilities in Enterprise Architect used in these processes. It also provides an overview of the compiler interface used for debugging within Enterprise Architect and the prerequisites for using these facilities.

Write, Build and Debug Process

When working with any development environment the standard process of coding and debugging involves the following general cycle of steps – the “Debug Cycle” as depicted below:

![Figure 1: The write, build and debug process](image_url)
This depicts the flow of the standard process after generating your code stubs. It involves updating the code, compiling the code, setting up break points and markers, then running through the debug process and tracing any problems encountered back to the source code ready to re-edit the code.

The following core points of this cycle are presented in this section:

- **Chapter 2** - Opening the core views see [Code Editing & Setting Debug Views](#).
- **Chapter 3** - Setting the Package Builds scripts - see [Building / Compiling](#).
- **Chapter 4** - Setup the Debug Views that you require: Debug, Breakpoints views etc.
- **Chapter 5** - Covers the main debug facilities.

**Tip:** Be familiar with the comprehensive code editing functionality in Enterprise Architect (including Syntax Highlighting, Intellisense and Bracket Matching). For details on this see the code editor functions in the User Guide: [Code Editors](#).

**Enterprise Architect Links to the Compiler**

Enterprise Architect allows you to link to the compiler and automate the compile process from within the modeling environment, as well as, run the application using the internal debug facilities. These facilities include standard debug options, such as setting breakpoints, executing and stepping through code, and options to view “context data” such as variables and stack views.

Figure 2 illustrates the flexibility of connecting to external compilers.

![Figure 2: Enterprise Architect connectivity to compiler, code and executables](image)

**Prerequisites for Coding Building and Debugging**

In order to use the Integrated Development options within Enterprise Architect it is assumed the following processes are completed:

1. The Classes are defined within the model.
2. The Classes within Enterprise Architect are related to their source code files (See [Chapter 2](#)).
3. Ensure the external compilers are set up correctly (e.g. Visual Studio .Net or Java). For more information see Prerequisites (details are covered in Chapter 3). Check also:
   o The Linkage to the Compilers is available (see help: Add Build Scripts)
   o Any Operating System environment variables (e.g. CLASSPATH for Java), need to be set. See Help: Integrated Development Setup.

This chapter has covered how Enterprise Architect links to the compilers and what prerequisites need to be set up before coding and debugging. The following chapter discusses using the internal editing facilities.
Chapter 2 – Working with the Code

When working through the build and debug process Enterprise Architect provides access to edit the code, build the application, and then debug the code while setting break-points in the code editor. This chapter covers Enterprise Architect’s facilities for setting up custom views for interacting with the edit, build and debug facilities.

Establish your code

There are two ways of linking code in a model; modeling the class structure and then generating the skeletal code from the model, or reverse engineering an existing code base.

- **Model Driven Approach**
  
  When working from a model driven perspective, the Classes might have been defined, but might still be Platform Independent - and not yet Platform Specific (e.g. C#).

  If you are working from Platform Independent Models – you can do an [MDA Transform](#) to a target platform and select the Option to [Generate Code on Result](#). This generates skeletal code for the Platform Specific Model.

  If you have a Platform Specific Model then simply [Generate](#) the skeletal code.

- **Reverse Engineering**
  
  Where the code has been developed using an external application (say an IDE) and you must import the code see [Import Source Code](#).

  **Note:** Enterprise Architect also supports options for integration with Visual Studio and Eclipse. Details on using these for code development are outside the scope of this document. For more information on these integrations see: [MDG Integration](#).

Your code should now be linked to model elements, ready for editing. To verify the linkage: right-click on a Class in the Project Browser and then select: [View Source Code](#) or press F12.

Code Editing

The benefits of the code editing being integrated to the modeling and the debugging facilities are that you can quickly check if your code conforms to model guidelines as well as debug your application directly in the model. These model checks can be carried out from the code editor along with facility to set breakpoints and view variables.

The code editor supports key code editing functions such as: Syntax highlighting, Intellisense, Auto-indenting, “Go to Function Definition” and “Go to Declaration”.
Figure 3: Selecting an operation from the Structure Tree.

```c
72 DWORD CTrain::OnArrival(CStation* S)  
73 {  
74   Departing = S;  
75   Location = 0;  
76   Delay = (Disembark() + Embark());  
77   DWORD ScheduleTime = Network->TimeAtStation(Departing);  
78   if(Delay > ScheduleTime)  
79     return Delay;  
80   return ScheduleTime;  
81 }  
82 DWORD CTrain::Run()  
83 {  
84   if(Network->Stations.size() < 2)  
85     return 1;  
```

Figure 4: Context references in the code editor for functions and variables.

Using the context menu ‘Search’ option you can do an immediate search of the model information, allowing you to ensure design correctness is reflected directly in your code.

For more information on the functionality provided in the Code Editor facilities see the help sections: Code Editors and Code Editors Context menu.
Customize Workspace Layouts for Debugging

Enterprise Architect provides a variety of workspace layouts including default layouts for code editing and debugging. They can be quickly setup using the Workspace Layouts Toolbar – see help: Workspace Layouts.

Two standard layouts that cover debug related views are: “Execution Analysis: Visual”, “Execution Analysis: Debug”. These views can be used as the basis for your own custom workspace layouts for debugging.

The following screen shows a custom Layout with the key views used for debugging:

![Figure 7: Code Editor and Debug Views](image)

Enterprise Architect’s code editor supports two panes; the Structure pane (left upper pane - Figure 7), and the main Edit pane (middle pane Figure 7). The Edit pane supports many options for general editing, as well as, interaction in the code while debugging. For more details on the code editor options see: Code Editors and Code Editors Context menu.
In this chapter we covered the Enterprise Architect options for setting up custom views for interacting with the edit, build and debug facilities.
Chapter 3 – Configuring Build and Debug in Enterprise Architect

By linking Enterprise Architect to the Compiler/Interpreter, you can compile and run the application from within the model. The key advantage being that you keep the code, build and debug processes within the one application, maintaining consistency between the model and the code facilitating improved traceability.

This chapter gives an overview of configuring Enterprise Architect for automating the build process.

Setting up an Analyzer Script

Using an Analyzer Script in Enterprise Architect, you can automate compilations using .Net and Java compilers, as well as, linking directly to a PHP interpreter. This chapter covers the setup of these connections as outlined in the diagram below.

What is covered below is an overview of creating a build script for a Visual Studio C++ application available in the EAEexample.eap model supplied with Enterprise Architect. For more details on the set-ups specific to .Net, Java compilers or PHP interpreter, see the help topics under Integrated Development along with the Learning Center topics: Build and Debug | Build and Run

Analyzer Scripts

In any model there can be a number of different applications under development. A separate build script can be defined for each application; these can be created and accessed from the Execution Analyzer view, which is accessed from the main menu: Analyzer | Execution Analyzer (Shift+F12).
Figure 9: Selecting an Analyzer Script Containing the Build script

Double-click on an Analyzer script to open the dialog containing the script details:

Figure 10: Analyzer script details

These scripts include settings for building (compiling), running, debugging, running xUnit and Testpoints tests, and generating Sequence diagrams. In this section we focus on the Build and Debug options.

Setting up Linking for Builds (compilations)

To add a new Build script:

1. In the Execution Analyzer view select the New script icon.
2. This displays the Package Selection view for you to select the package-tree that contains the Classes with the code.
3. Select a package and Press OK.
4. In the Execution Analyzer dialog define the Build compiler location and the Default directory for the Code. See Figure 10 above.
5. Click on OK to save the script and return to the Execution Analyzer view.
For more details on setting up the build scripts specific for the .Net and Java compilers see the help topic on Add Commands and the Learning Center topics under “Execution Analysis”.

You can now do a Build using the icon. This instigates the compiler and any logs generated by it are listed in Enterprise Architect’s Output window. Figure 10 is an example of the log for a compilation using Visual Studio.

![Output view showing the compiler output](image)

**Figure 11: Output view showing the compiler output.**

Ensure that there are no compilation errors. In the above example this is indicated with: **Exit Code 0**.

**Tip:**
When setting up multiple Build scripts, a method to simplify the process is to set a Local Path for the Compiler directory. To set a Local Path, select the main menu option: Settings | Local Paths. See help page: Local Paths Dialog.

Where compilers for different languages are used – set an ID and path for each.

Then in the Build script use the Local path to define the path to the Compiler:

![Using the Local Path to set the Build path for the compiler](image)

**Figure 12: Using the Local Path to set the Build path for the compiler**

This Local path can now be used on multiple build scripts which will simplify:

- Creating new Build scripts involving the same compiler
- Upgrades on the Compiler version (if installed on another directory). This sets a single point that the path needs to be updated.
Locating Compile Errors in Code

After a build, if the compiler returns errors in the code, these are listed in the Output view as links back to the code. You can simply double-click on a logged error to open the code at the associated line, providing you with a quick means to update and correct syntax errors in your code.

![Code Snippet]

Figure 13: Accessing compiler errors in code.

Setup for Debugging

Having setup the build script you now configure settings for debugging the code. Enterprise Architect supports debugging of:

- Microsoft Native Code applications (C++, Native C & VB)
- Microsoft .NET applications
- Java applications
- PHP

Each of these technologies requires different configurations for debugging. For more information on setting the debug script for each technology, see the “See also” in Setup for Debugging, then select the platform to be used.

Figure 14 shows the configuration for a C++ application using the Microsoft Native configuration.
Figure 14: Configuring the debug options in the Execution Analyser.

Figure 15 below gives a general overview of the process of code editing, compiling and debugging.
Chapter 4 – Debugging from the Model

Chapter 3 outlined how to connect the modeling environment to the compiler for building applications and debugging. In Chapters 4 and 5 we will discuss the options for debugging and updating the code, including options to set breakpoints and view variables, examine call-stacks and memory, as well as related options for searching code and tracking function definitions in code.

These all provide a comprehensive facility for the debug cycle that was historically done within the IDE.

Tip: Set your own viewing environment or set the workspace layout to “Execution Analysis: Debug” – for options see Setting up Debug Views as outlined above.

Compile for Debug

Critical to the debug cycle is the Compile process (Build). After there are updates to code, the code must recompile for debugging. To run a build from the Execution Analyzer use the icon or (Ctrl+ Shift + F12).

Setting Breakpoints

Breakpoints can be set in two ways; by marking lines in the code, or they can be set to occur on the change of the value of a variable.

Standard Breakpoints

Figure 16 below shows an example of the code editor when in debug mode, with break (line 61). The code has been stepped through to line 63 (☞) and a mouse-over on the variable “Passengers” shows this its current value (Passengers=18):

Figure 16: Code stepping past a break point; mouse-over variables to display their values.
Figure 17: Breakpoints view - shows the break on line 61 as defined in Figure 16.

These Break Points can be set On/Off in the Breakpoints and Markers View (see Figure 17).

**Breakpoint Conditions**
When setting a Breakpoint you can set a constraint as to when it is applied. For instance, when starting to debug, you might have multiple breaks on a point until a variable is set. To avoid having to make repeated clicks to continue the run, you can set a Condition to not break where the variable has not been initialized: e.g. `str_Name <> ""`. For more information see the help topic “Breakpoint Properties”.

For more information on breakpoints see the help topic: Breakpoint and Marker Management. Note: other markers associated with this view are covered in the Visual Analysis section below.

**Value Change Breakpoints**
Code line breakpoints provide a simple starting method for identifying a problem; however there can be times where an obscure event causes a change in state of a variable. Setting a breakpoint based on a ‘Value Change’ can give clarity with far less user interaction (less code stepping). These breakpoints can be set after pausing for a breakpoint. They are set in the Locals View against a specific variable, as shown in the following diagram:

![Figure 18: Setting a breakpoint based on a variables change of Value](image-url)
For information on setting Value Change breakpoints see: Setting Value Change Breakpoints.

Enterprise Architect also supports setting markers for collating data used in Analysis – this will be covered in more detail in Section 2.

**Stepping through Code – the Debug view**

As with Integrated Development Environments, Enterprise Architect provides comprehensive functionality for controlling the flow of steps when operating in Debug mode.

The Debug view is a critical tool for stepping through your code and jumping to breakpoints. The Debug view also displays a log of: the processes performed on the start of a session, runtime exceptions and errors, along with diagnostic output; such as C++ Trace statements, and Session.Output statements.

The debug functions can also be accessed from the Debug Toolbar.

For more details on these options see the following help file entries: Start Stop & Step the Debugger See also using the Debug toolbar: Debug Toolbar.

In this chapter we have covered setting breakpoints for debugging and stepping through the code, allowing you to locate the errors and bugs by either manually stepping through the code or examining the behavior of the code in real-time. As the code is traceable to the design, the behavior of the code can be viewed while debugging and verified against the model to ensure it meets the design criteria.
Chapter 5 – Using the Debug Views

When stepping through code using the debugging tools it is critical that information behind the execution is visible. Enterprise Architect supports numerous debugging views such as Local Variables, Call Stack, Watches (out of scope variables) and Memory. It also supports viewing variable values against the code in the editor (see Figure 21).

The following sections cover these debug views in detail:

Locals View

When stepping through the code, the Locals View shows the values, variable type and address of the local variables for the current line of code.

![Figure 19: Local Variables view](image)

The Locals view supports a number of useful options including defining Breakpoints based on a change of value of a variable, viewing long strings and comparing variable snapshots.

![Figure 20: Locals view options - includes setting variable change breakpoints](image)
The Local Variables can also be viewed in the code editor by moving the cursor over a variable. Figure 21 shows the value-set for the cursor selected variable ‘Departing’:

![Code snippet showing variable 'Departing']

Figure 21: Viewing variable values from the code

Variable values can be viewed by placing the cursor on a variable and from the context menu selecting the Display Variable option:

![Context menu showing Display Variable option selected for variable 'Departing']

This allows you to view any complex data structure more clearly in the Locals View:

![Locals view showing selected variables]

Figure 22: Local view detailing the selected variable.

**Note:** the option - Display Variable - when used on global variable returns these in the Watches view (see Watches below).
Call Stack View

The Call Stack lists the threads on the stack:

![Call Stack View](image)

**Figure 23: Call stack view**

During debug this window can be used to view the stack at the current code-line being stepped through.

If there are multiple threads, these can be inspected. On selecting a thread you can use the stack view to check the code context that represents the current instruction pointer or its calling code.

On double-clicking any compiled code thread in the Stack view, the Code Editor view and Locals view are updated to reflect the line of call and the variables associated with the selected thread, giving a powerful tool for tracing the history of execution when debugging.

**Tip**
The Call Stack is used more extensively when changes are recorded using the Record History and Sequence diagram generation (see Section 2 - Execution Analysis). This gives a clearer diagrammatic overview of Call Stack history.

Watches View

The Watches view supports viewing persistent variables; that is, global variables that overlay the scope of the Local Variables view. These can be viewed by selecting a variable in the code or by typing the variable name into the Watches tab. The following code view shows the process for displaying the cursor selected variable:

Tips:
- When you want to see where in the code a Variable is being changed - set **Breaks on Change of Value** in the Locals View.
- If small changes to complex variables are difficult to view – save an initial Snapshot. Then use the **Compare Variable** Snapshot option to compare values at different points in the execution.
- When dealing with long text strings – use the **View in Editor** option.
- To check for memory conflicts you can view the Memory directly at the address given (see the details of Memory view options below).
‘theNetwork’ being a global variable returns the details in the Watches view:

![Figure 24: Displaying in Watches the cursor selected variable](image1)

Note: the same process (Figure 24) when applied to a Local variable opens the Locals view to show the Local variable value.

The Watches view can also be used to display specific variables in other scopes using the syntax: Module!Class::VarName. For example see Ctrain::Passengers in Figure 26:

![Figure 25: Watches view with the expanded ‘theNetwork’ variable](image2)

![Figure 26: Watches view](image3)
Search

When debugging code linked across multiple files, the Search view is a critical component. It returns references to all occurrences of an item searched for.

The following images give an example of accessing related code in other files using the code editor context menu. A right-click on the variable ‘Passengers’ returns these options:

The ‘Search in Files’ option returns all the results in the current search directory. Each entry shows the location by line number and the file name.

Selecting an entry opens the editor to the specified line:

Figure 27: Using the Search facilities.
For more details on setting the search paths and file-type filtering, see the help entry: Search in Files

Modules View

The Modules view is a list of the modules called during execution of the code. This can be used to check if:

- A specific module is loaded (e.g. User32.dll is loaded).
- Two modules are loaded when only one is required.
- Module A is dependent on Module B – then check that both are loaded (e.g. Rich32 requires Rich20 – check both are loaded).
- Debug is operating correctly (e.g. by verifying that the PDB file was loaded).

![Figure 28: Module view showing the modules loaded when executing the code.](image)

Memory View

The Memory viewer allows you to check the raw memory data. The memory address of a variable can be selected using the Locals view.

![Figure 29: Memory view - showing a selected variable memory reference.](image)

This view allows you to inspect the process memory address to confirm that the data is stored at the correct address and not overwritten by any external process. For more details see the Help page: Inspect Process Memory

Output View

The Output view logs the sets of statements issued by the Compiler when performing a Build.
Where the compiler issues an error on a line of code you can simply double-click on the logged error to open the code at the associated line.

For an example of tracking a compiler error back to the code, see the section: Locating Compile Errors in Code.

Debug View

The Debug view, apart from supporting stepping through code, also logs trace statements. These include C++ Trace statements, VB 6 Debug.Print, Session.Output statements.

For more information on:

- The options in the Debug window see help entry: The Debug Window.
- Using the actions available in debugging—see: Debug Actions

Tips on using Views

If you want to view related data from the different views, you can setup separate view groups or keep a set of views on one docked or floating window. Using these options for view placement, you can organize your views to best suit your requirements and then record this layout as a “Workspace Layout” view. Some examples of this are:

1) Where you enable / disable breakpoints keep the Break Points view, Locals and Call Stack views separate:
Figure 30: Viewing the Breakpoints, Locals and Call Stack views as a group.

2) Checking a variable in memory; keep the Locals and Memory viewers both in one view:

Figure 31: Accessing the Locals, Watches and Memory Viewer as an open group.

3) Alternatively you can keep these as tabbed items on one View:

Figure 32: Grouping the Locals Watches and Memory Viewer as a tabbed group

4) Other tips:

The Debug view, Execution Analyzer view and the Profiler view are independent views and can be tabbed to the same window:
Figure 33: Nesting views.

Note: Execution for debug, Testpoints or a Profile can be started from the relevant view e.g. Debug, Profiler, Testpoints, but any output still appears in the Debug view, making it sensible to keep the Debug view accessible.

In Section 1 we have covered the core features for building, debugging and updating code.

- Chapter 4 covered:
  - Setting breakpoints
  - Stepping through the code
- Chapter 5 covered:
  - Using the debug view to examine variables, call stacks, memory and other system data.

In section 2 we will follow up on refining this code using the Execution Analysis Tools.
Section 2 – Visual Execution Analysis

When building a complex system there is often a need to refine the code. This requires analyzing the system and verifying the code conforms to the design, as well as, assessing the performance of that code. When this analysis is performed within the Model Driven Development Environment the analysis results can be checked against the design and alterations can be carried out all within the one application.

Enterprise Architect includes a number of tools used for higher level analysis of the program execution. These Visual Execution Analysis tools provide:

- **Sequence Diagram generation**: For visualizing the flow of the code at high level of abstraction
- **Profiling**: A means of summarizing, by thread and routine, the code’s general efficiency
- **Object Workbench**: For performing Class based run-time analysis using workbench instances

Sequence diagram generation enables you to produce diagrams that reflect the original design. Developers can visually cross-reference the original design diagrams against diagrams generated from the executed code (compare the “As Designed” to the “As Implemented”). These diagrams can also be used to better understand a reverse engineered legacy system by providing an ‘overview’ of the general code flow.

Profiling reports can be used to evaluate the efficiency of the code. It can be used for narrowing down analysis to critical areas that need refinement. They provide a quantitative analysis of CPU usage for each thread and routine call within a thread.

The Object Workbench provides a means to perform runtime analysis by creating ad-hoc instances of a Class in a Model. This can then be used invoke methods and instances, and alter variables to check and debug your code.

Figure 34 provides an overview of the interaction between the Visual Execution Analyzer and the executable code to deliver the analysis as UML Sequence Diagrams and Profiles.
Figure 34: Process-flow for Abstract Analysis of Executable code

**Note:** Sequence Diagram generation can be an excellent tool to familiarise a new programmer with existing code. It can provide a clear overview of the run-time behaviour of the code prior to making any updates.

**What is covered in this section:**
- Sequence diagram generation [Chapter 6]
- State Chart checking [Chapter 7]
- Profiling: viewing/reporting the execution flow [Chapter 8]
- Workbench: runtime analysis

**Options:**
As there are many compilers and different analysis tools, the following table lists the compatibility between analysis tools and compilers:

<table>
<thead>
<tr>
<th></th>
<th>.Net Native</th>
<th>.Net Non Native</th>
<th>Java</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Diagram</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State Chart Checking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Profiling</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Work Bench</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Testpoints</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Execution Analysis tools related to supported Compilers.
Chapter 6 - Sequence Diagram Analysis

Sequence Diagrams generated from runtime code help you to identify errors in logic, explain unexpected system behavior and check that the process flow of the code is conforming to the original specification.

Depending on the start points in the code and the capture options, you can generate different views for analyzing different aspects of the flow of the code. These can then be used to analyze the flow of complex code and determine what changes or refinements are required.

How does it work?
When you execute an application in debug mode using Enterprise Architect, the flow of events can be recorded in the Recording History window. Recorder data includes: arguments to functions, calls to external modules and state transitions based on any given constraint.

Using the Sequence diagram generation, the Record History is simplified and clarified into a diagrammatic form. This history can then be used interactively with the Profile reports generated by the debugger.

![Sequence Diagram generation process](image)

Figure 35: Sequence Diagram generation process.

Why do we use this?
Sequence Diagram generation principally provides a mechanism to view runtime details from an abstract view. This simplifies the analysis and shows patterns of function calls and processes.

Sequence Diagrams help programmers too:
- Visualize the general structure of the function calls
- Clarify errors in logic
- Optimize the code execution
- Identify data flow inconsistencies

Efficiency Example
Below are two examples of a Sequence diagram generated from a recursive call. The diagram on the left shows the initial analysis. The diagram on the right shows the sequence after checking and refining the code to minimize the unnecessary calls.
On viewing the original Sequence diagram it is obvious that the call sequences were overly complex. By adding a simple conditional statement to the code, the call sequence is made more efficient. This increase in efficiency would be more pronounced if the call was to an external module, as it would significantly reduce the execution time by eliminating unnecessary calls to the external module.

When working purely on the code level this type of problem is not necessarily obvious, but it becomes more pronounced in the abstract view of a Sequence diagram.

Figure 36: Sequence Diagram generated from recursive call.
There are numerous views that the Sequence diagrams can produce, depending on the points in the code where they are run and the type of Sequence diagram recording used - Code based diagrams or Stack view diagrams.

Flow Example
To give you some examples of using the Sequence Diagram generation we will run through a code example called “City Loop” that is available in the EAExample model supplied with Enterprise Architect. This is a C++ example application mimicking trains in a metropolitan train subway. For information on accessing and running this see: Learning Center (Alt+F1) | Execution Analysis | C++/Native Examples.

Below is an image of this application during execution. It uses three parallel Threads (3 trains CTrain 1, 2 & 3, going through an underground train station loop).

The stations are: Central, Parliament, Treasury, Spencer and Lonsdale.

![City Loop Recording Sample](image)

Figure 37: Subway program - three trains (numbered 1..3) progressing through a sub-way loop.

The Sequence diagram below shows the Call Sequences on arrival at the stations.
Figure 38: Sequence diagram showing procedure calls in three instances of a class

Generated Sequence diagrams can be enhanced by setting up a set of conditions on state transitions. See setting Generated Sequence Diagrams showing State Transition Changes below.

Setup for Generating Flow Sequence diagrams

The core steps that must be undertaken prior to recording are:
1. Open your Build Script from the Execution Analyzer view (Shift+F12)
2. Configure the Recording Options (see Sequence Diagram recording detail).
3. In your code, set Recording Markers for at least one start point.

Let’s discuss these steps in more detail:
1. A prerequisite for generating Sequence diagrams is to filter out the recording of ‘noise’ from low-level calls, as these produce unwanted information in the diagrams. A typical example is eliminating Foundation Class calls. A simple scenario might be eliminating reference to Standard Templates using “std**”. As the filtering is setup for the Build script, it then applies to all Sequence diagrams generated for that script. For more details see: Sequence Diagram recording detail.
Another useful option for simplifying the diagram is to limit the stack depth that is recorded. This is set in the Record and Analyze view.

2. The next step is to mark the range of code that you want to report on. You do this with Markers set at the start and end points of what you want to report. This can be done by either:
   
   o Manually setting the Markers directly in code. For more details see: Setting Recording Markers

```
56 } return 0;
57
68 DWORD CTrain::OnArrival(CStation* S)
69 {
70     Departing = S;
71     Location = 0;
72     Delay = (Disembark(GetRandom()) + Embark(GetRandom()));
73     DWORD ScheduleTime = Network->TimeAtStation(Departing);
74     if(Delay > (int)ScheduleTime)
75         return Delay;
76     return ScheduleTime;
77 }
78
79 DWORD CTrain::Run()
80 {
81     if(Network->Stations.size() < 2)
```

Figure 39: Code line with a Method Auto marker set at the start of the function being recorded.

   o System generated markers using the Class Markup selection. Note that once the Class has been ‘Marked Up’, there is a new Marker-set created. This marker-set must be selected from the Marker-sets drop-down, as shown below:

   ![Figure 40: List box showing the available Marker Sets.](image)

**Note:** Generating Sequence diagrams requires that at least one Marker (or a break–point) is set before the Sequence Generation is performed.
3. Open the Debug view (Alt+8) and select the Debug Start button.

4. If you have not set a Breakpoint-marker – you must select the Debug Stop button. This prompts you with a dialog for generating the Sequence diagram.

Below is a Sequence diagram generated from the EaExample CTrain Class as per the Marker set in Figure 39:

![Sequence Diagram](image)

**Figure 41:** Sequence Diagram generated from Marker set in Figure 39.

The CTrain Class is accessible in EAExample from:

*Project Models.Execution Analysis.C++/Native.Source.CTrain*
Chapter 7 – Generated Sequence Diagrams Showing State Transitions

Sequence diagram generation also supports a feature for showing where state changes occur during program execution. These are shown in the generated diagram as States. These State Elements depict a change in state of the code variables that match a set of constraints that you define.

Figure 42: State Change as depicted in a generated Sequence Diagram

Note: With State Transition Sequence diagram generation, separate instances of a Class are depicted as Sequence objects. With the Train application (introduced in Chapter 6), there are three instances of CTrain running. Two of these instances are shown above as CTrain.1 and CTrain.2.

You set Constraints by creating a State diagram as a child of the Class that you are analyzing. This diagram has a set of State elements – each defining a Constraint. The Constraint is used to compare changes of state for the Class variables against what you define in your Constraint clauses.

Figure 43: A State Chart used in Execution analysis must be a child of the Class being analyzed, and include Constraints for State Transitions.
As previously stated; the analysis requires that a marker be set on a line of code to define the start of recording:

```cpp
68 DWORD CTrain::OnArrival(CStation* S) 
69 {
  70  Departing = S;
  71  Location = 0;
  72  Delay = (Disembark(GetRandom()) + Embark(GetRandom()));
  73  DWORD ScheduleTime = Network->TimeAtStation(Departing);
  74  if(Delay > (int)ScheduleTime)
  75   return Delay;
  76  return ScheduleTime;
  77 }
```

Figure 44: Code Marker setting for Recording a Sequence Diagram

To record a sequence diagram with State Transitions, use the **Record** button from the **Record & Analyze** view. It will prompt you for options to select the State diagram and debug settings. This will produce a Sequence diagram depicting where the States changes occur:

Figure 45: Sequence diagram that reflects a run time instance of the State Chart (Figure 43).
The States in Figure 45 reflects the Constraints set in Figure 43. Where a variable state change occurs and a Constraint is satisfied – then the State is shown in the generated Sequence diagram.

Checks on constraints apply to changes in the state of Class.Attribute values e.g. CTrain Class has Location :int and Departing :CStation. An example constraint could consist of “Location=0”, “Departing.Name=Lonsdale” (see Figure 43).

For more information on configuring State Transitions see the Help topics: Recording the State Machine.

The example State Chart diagram can be accessed from the EAExample model under: Project Models.Execution Analysis.C++/Native.Source.CTrain

Setup for Generating Call Stack Sequence Diagrams

The Call Stack Sequence diagram generation feature creates a diagrammatic report of the Call Stack. It can be very useful while stepping through code to quickly identify problems and trace these to their source. It is best used to view the current stack state and determine the logical correctness of the code and in cases where crashes occur, to see the end point or state of the code prior to the crash.

Below is a simple example of the Call Stack Sequence diagram for an application that is recursively reporting Packages and Elements to a word processor application:

![Call Stack Sequence Diagram](image)

Figure 46: A Sequence diagram generated from the Stack view

The call stack diagram can be generated from the Call Stack view (main menu: Analyzer | Call Stack) using the Generate Sequence Diagram icon.
Chapter 8 – Profiling and Workbench

Profiling Native Applications

Enterprise Architect Profiler is a dynamic code analysis tool that allows you to drill down into any sections of code that may be inhibiting performance. On isolating specific sections of code that are under-performing, you can then start refining the code to achieve optimal response.

The Profiler works by periodically sampling a running application and collecting the stack information for all threads. This is then collated to report the number of hits with the objective of capturing data on the most frequently called operations (highest cost).

This feature is very useful for checking how code is threaded and which functions are called most frequently. It allows for viewing usage against each of the different threads, how routines are invoked within these threads and the most frequently called code sequences.

Note: Profiling can be used only with Native .Net (C++, C) and VB 6 programs.

Below is an example using the Train Station application (covered earlier in this document) using the CTrain Class. This shows the Profiling of three instances of the CTrain Class being executed:

Figure 47: Profiler report on three threads CTrain 1, 2 & 3 in the Train Station application
The following is a Profiler report of the application scanning through an Enterprise Architect repository (see Figure 36 above). The report shows recursive calls within a thread.

![Profiler report image](image)

**Figure 48: Profiler report of calls in the Repository report code.**

**Workbench**

The object workbench is a facility that lets you interact manually with Class instances to establish whether the Class behaviour conforms to its specifications. By manipulating the instance's properties and by invoking its methods, you can produce Interaction diagrams that show how the class behaves when you invoke its methods. The workbench allows you to verify that the implemented software is as designed.

Workbench supports the Java and the .Net platforms excluding ‘native’ C++, C and VB.

It can be very useful when testing that an operation to conforms to specifications. It can also help you become familiar with a new language – being able to invoke a simple section of code to quickly verify the results.

To start a Workbench you need an Execution Analyzer build script. The Execution Analyzer script needs to define a reference to the debug assembly to enable this to be run from the workbench. For an Example see the EAXmple model: **Execution Analyzer script: Project Models.ExecutionAnalysis..Net.Workbench**.

In the Analyzer script, view the **Debug | Workbench** page and set the links to the assembly.
You can then click on a Class, and select **Execution Analyzer | Create Workspace Instance** from the context menu. For details on creating and working with a workspace instance see the help page: on **Creating Workspace Instance** and **Invoke Methods**.

Below is an example of executing an instance of the CPerson Class. This can be accessed in the EAExample model under:

**Project Models.Execution Analysis..NET.Workbench.MyClassLibrary**

![Figure 49: Using Workbench to execute an instance of a Class.](image)

For more information on using the Workbench facilities see the help section: **Object Workbench**
Section 3 - Testing

A core part of Visual Execution Analysis is visual testing of the code. Using Enterprise Architect’s Integrated Development Environment you can test your code interactively in the modeling environment, giving you tight integration between design specifications, test plans, code execution, and the collated test results.

Enterprise Architect supports two code testing features. These are Unit Testing and Testpoint Testing (based on Design by Contract testing). Both of these are similar in their overall purpose; however Testpoints are defined and run at the level of the debugger rather than requiring user-written code and compilation of such code.

Specific details on using both of these testing methods are covered in the following chapters:

- Chapter 9: Test Points or “Design by Contract” covers:
  - Setting Assertions against the code via the debugger.
  - Running the code and checking the Testpoint (Assertion) results.

- Chapter 10: xUnit Testing covers:
  - Using MDA Transforms for creating skeletal script NUnit & JUnit tests Classes.
  - Setting up package Build Scripts for Unit Tests and running these.
  - Recording xUnit Test results direct to Element::Test Cases.

**Note:** Enterprise Architect includes a number of other Test related facilities external to Integrated Development – for more information on these, see the Test Management whitepaper.
**Chapter 9 - Testpoints**

Testpoint analysis is based on the same principles as “Programming by Contract”. Testpoints allow the programmer to define a set of rules on how the class and its operations will behave. Executing the code with Testpoints, tests if the code conforms to this set of rules (Contracts).

These Test Conditions are set by the user in the model and are evaluated using the debug data produced on compilation of the application. This has the following advantages:

- It obviates the need to write and compile xUnit test scripts
- Test definitions can be changed without re-compilation of the test code
- Installation of xUnit applications is not required (NUnit or JUnit).

Along with simplifying the test definition, Testpoints uphold a Model Driven Development principle: Test statements are defined within the model rather than using external code that can be difficult to trace back to the system requirements.

Testpoints can be created in the model at the same time as you write your code to implement each class. Testpoints are equivalent to ‘contract clauses’ in “Programming by Contract” principles. Testpoints can consist of ‘Constraints’ (or ‘Assertions’ in “Programming by Contract” principles). These constraints make assertions about the operation’s outputs and how it should behave.

![Figure 50: Testpoint integration in Enterprise Architect.](image-url)
Constraints can be defined in Classes and Operations.

Classes can have:
- ‘Class Invariant’ Constraints

Operations can have:
- Pre-Condition Constraints
- Post-Condition Constraints

Constraint definitions also support Trace Points. Rather than being assertions, these return the values of variables (similar to a Debug.Print/Session.output statements) to the Output.Testpoints view.

Example

We will now demonstrate the process of creating Constraints in a model. It is assumed that we have a model linked to an application that has been compiled and executed using the Visual Execution Analyzer. For this example we use the Ctrain application that was introduced in Section 2 and is available in Enterprise Architect's example model. We will execute the compiled application and view the logged results of the set of Constraint clauses.

To open the Test points View select: Analyzer | Test | Test Point Manager.

To setup tests against a Class select the Class to be tested in the Project Browser (in this case CTrain). In the Testpoints view this shows the Operations on the Class with any Test-Points that have been set against them:

**Figure 51:** The Disembark Operation showing the Post Condition - Test-point.

Test Points allow a user to create ‘Test Contracts’ on both a Class and any Operation under it. Figure 51 shows an example of an Operation test point.

**Operation Constraints (Contract Conditions)**

To define Constraints for an Operation you use the Operation Constraint window.
This is accessible from the Testpoints view (and the code Editor). By right-clicking on a Class or an Operation you can select Add Testpoints from the context menu. This will display the dialog:

![Operation Constraints]

Figure 52: Setting Contract Conditions

For a Class there is an option to set a single Condition. For an Operation you can set a Pre-condition, as well as a Post-condition. You define these conditions as follows:

**For Testpoint Constraints:**

The condition statements must return a boolean result and can include the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>%</td>
<td>Integer Divide</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal</td>
</tr>
<tr>
<td>OR</td>
<td>Boolean OR</td>
</tr>
<tr>
<td>AND</td>
<td>Boolean AND</td>
</tr>
<tr>
<td>( )</td>
<td>Brackets</td>
</tr>
<tr>
<td>.</td>
<td>Scope e.g. c.Name</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Scope e.g. c-&gt;Name</td>
</tr>
</tbody>
</table>

The following is an example conditional expression:

\[
( (x \mod 2) > 1) \ \text{AND} \ (y \geq 10) \]

This returns a Boolean value.

Variables can be scoped by either: ‘.’ or ‘->’
For example to access the Name field of class C, use the following:

C.Name

To access an element in an array:

C.Names[0]

To access a property of an element in an array:

C.Names[0].Surname= ”Smith”

For Testpoint Trace:

Testpoints are different to the Constraints outlined above. They are statements that can return the values of variables (similar to a Debug.Print or Session.output statement). These Trace statements, however, are defined in the model, rather than the code.

Variables in Trace statements are identified using a prefix, $ or @.

For example: $C.Name, @C.Account

The $ operator coerces a variable value to print as a string. The @ operator prints values in a format appropriate to the variable’s primitive data type (floating point, signed integer etc). These operators are only compatible with variables of a primitive data type.

Below is an example of the Trace-Point output showing the variable, People, in Ctrain.Disembark:

Trace: Departing=$Departing.name

During execution, such a Trace statement is displayed as shown in Figure 53:

![Figure 53: Test points output for the variable ‘People’ for the Trace set in the operation ‘Disembark’.

Defining a set of Testpoints

Test Points allow a user to create ‘Test Contracts’ on both a Class and any Operation under it. When wanting to create Test Points on more than one specific Class you can group these by using Test Sets. Test Sets can be combined into ‘Test Suites’ to form a collection of tests across related Classes.
For example, when testing print functionality that calls a combination of Classes you would combine the Test Cuts (for these related Classes) into a Test Set. You can also aggregate these Test Sets, using Test Suites.

![Testpoints window](image)

**Figure 54: Viewing the Testpoint tree.**

Test runs can be executed at the Class, Test Cut, Test Set or Test Suite levels. Below is a simple Test Domain diagram that defines the tests in Figure 54.

![Test Domain diagram](image)

**Figure 55: A Test Domain diagram - set to view Testing.**

Note: This diagram is set to view the Element.Testing in the diagram Properties. It shows a set of test results for a number of Testpoint runs (saved in the Class as Unit and Integration tests).

**Execution of Testpoints**

To execute Testpoints, you must define the standard Build scripts that can be executed with the latest compiled application code using the Run icon on the Testpoints window.

When running tests for a single Class, that Class must be selected in the Project Browser. The following illustration shows a Testpoint run from a Test Cut “CTrain”:
Combining Test Sets

As stated above, Tests often include calls in to multiple Classes. To combine sets of tests from multiple Classes, you create groups by relating TestCuts, TestSets and Testgroups.

After defining Testpoints in relevant Classes, a simple method to generate a Combined Test Set in a diagram, is to run your application from the Record and Analyze view, and then create a Testpoint diagram using the Testpoints icon.

Figure 57: A Combined Test Set diagram

To run a Test Set or a Test Suite, the elements must first be selected in a Test Domain diagram or in the Project Browser.

Below is an example of tests run by selecting the element “Test Domain” in Figure 57 above. Notice that it includes both Ctrain and CNetwork as defined in the Testpoint diagram above.
Figure 58: A Testpoint run using the Combined Test Set above

Saving Test Set Results to Test Cases

The results from Test-point Tests and Test-point Traces are by default shown in the Output View (Ctrl+Shift+8). These results can also be output to file. To set this option, open the Execution Analyzer (Shift+F12), then select a script.

Double-click on the Execution script and select Test | Testpoints.
Chapter 10 - Unit Testing

When debugging and making frequent updates to your code, xUnit testing helps to ensure you can re-test these updates quickly and simply. Enterprise Architects supports NUnit and JUnit testing (for .Net and Java respectively). This chapter gives an overview on setting up and using the Unit Testing facilities.

The general principle of Unit testing is similar to what is described for Testpoints. A core difference however, is that Unit testing requires that the tests be written in code and compiled to create executables. These executables are then used to run a set of tests. The test results are logged in the model, as tests within Class elements.

Figure 59 below, shows a diagram that displays NUnit test results logged against the ‘MoneyTest’ Class\(^1\). Enterprise Architect’s Testing view, shown beneath the diagram, provides an editable list of Unit tests and results.

![Diagram view of a Unit Test Class showing tests (above) – and the Test results View (below).](image)

Unit Test Setup

Setting up and running xUnit Test scripts involves the following steps:

1. Install the 3rd party xUnit test platform
2. Setup Unit Test Classes in Enterprise Architect (using an MDA Transform)
3. Insert Unit Test code into the Classes
4. Compile and Run the Unit Test scripts
5. View the Test results logged against the Unit Test Classes

\(^1\) You show the Test Cases on diagram elements by setting the diagram properties option: F5 | Element tab | [ ] Testing.
Each of these steps is covered in more detail below.

**Setup the 3rd party xUnit package**

The software to support xUnit is available in a variety of different forms. Enterprise Architect directly supports the NUnit and JUnit software testers. These must be downloaded and installed prior to using Unit testing (see the Appendix for download links).

**Generate skeletal code for the xUnit testing**

Unit Test stub Classes can be created manually or be auto-generated using Enterprise Architects MDA transforms. Once the code skeleton is created it is simply a matter of adding the appropriate NUnit or JUnit calls to perform the unit tests.

For more information on setting up manual build scripts, see the Help topic: **Set Up Unit Testing**.

Below is an example of applying an MDA transform on the application Classes to produce NUnit Classes:

![Diagram showing MDA nUnit Transform]

**Figure 60: MDA Transform for NUnit Classes.**

For details of performing the MDA transform see NUnit and JUnit options in the help topic **Model Transformations**.

Below is a view of the ‘RentTest’ NUnit code generated for the ‘RentTest’ Class above:
Insert xUnit Code

The skeletal code generated above must have the Unit Test commands added to it; for example:

```csharp
Assert.AreEqual(param1, param2);
```

For more details see: Set up Unit Testing

The following is an example NUnit Test Class (supplied with the NUnit application). This Test-Class is shown as linked with the Classes it is being used to test.

```csharp
Money
- fAmount: int
- fCurrency: String
+ Money(int, String)
+ Add(IMoney) : IMoney
+ AddMoney(IMoney) : IMoney
+ AddMoneyBag(IMoneyBag) : IMoney
+ Equals(Object) : bool
+ GetHashCode() : int
+ Multiply(int) : IMoney
+ Negate() : IMoney
+ Subtract(int) : IMoney
+ ToString() : String

+ Amount() : int
+ Currency() : String
+ IsZero() : bool
```

```csharp
MoneyBag
- fMonies: ArrayList = new ArrayList(5)
+ MoneyBag(Money[])
+ MoneyBag(Money, MoneyBag)
+ MoneyBag(MoneyBag, MoneyBag)
+ Add(IMoney) : IMoney
+ AddMoneyBag(IMoneyBag) : IMoney
+ Append(IMoney) : IMoney
+ AppendBag(MoneyBag) : void
+ AppendMoney(Money) : void
+ Contains(Money) : bool
+ Equals(Object) : bool
+ GetHashCode() : int
+ Multiply(int) : IMoney
+ Negate() : IMoney
+ Simplify() : IMoney
+ Subtract(IMoney) : IMoney
+ ToString() : String
+ IsZero() : bool
```

```csharp
MoneyTest
- f12CHF: Money
- f14CHF: Money
- f7USD: Money
- f21USD: Money
- fMB1: MoneyBag
- fMB2: MoneyBag

- SetUp() : void
- BagMultiply() : void
- BagNegate() : void
- BagSimpleAdd() : void
- BagSubtract() : void
- IsZero() : void
- MixedSimpleAdd() : void
- MoneyBagEquals() : void
- MoneyBagHashCode() : void
- MoneyCopy() : void
- MoneyCopy2() : void
- Normalize() : void
- Normalize2() : void
- Normalize3() : void
- Normalize4() : void
- Print() : void
- SimpleAdd() : void
- SimpleBagAdd() : void
- SimpleMultiply() : void
- SimpleNegate() : void
- SimpleSubtract() : void

- Money
  - fAmount: int
  - fCurrency: String

- MoneyBag
  - IMonies: ArrayList = new ArrayList(5)
```
code can then be run as defined in Run Unit Tests. The results of the tests are stored in the Element.Tests of the Classes being tested.

View test results

After executing the xUnit tests you can view the test results recorded against the Unit-Test classes. These can be viewed both in the Testing window (Alt+3) and in a diagram with the diagram-properties set to: [✓] Show Tests;

Figure 63: On execution of xUnit tests in Enterprise Architect, the test results are automatically used to generate Test Cases in the Unit Classes.

This image shows a diagram with the Diagram-Properties set to show Test Cases. These Test Cases are also shown below, via the Testing window.

Once you have defined Unit Tests as Class Elements, these can be run regularly for checking your latest code updates and then reviewed using the Test results stored against the associated Unit-Test Classes.

Conclusion

In this paper we have covered the features of Enterprise Architect’s Integrated Development Environment for writing, building, debugging, testing and analyzing the code. This paper has provided an overview of the set of code development and analysis tools supported, along with broad pointers on their usage and interaction. A core point throughout this paper is the benefit of system development in the modeling environment and how Enterprise Architect provides immediate connectivity from requirements definitions through to the software code. Using Enterprise Architect for Model Driven Development you can by-pass the problems with modeling systems in an application separate to the development environment. You then
avoid the inevitable errors caused by the separation between the code and the design specification. Instead you help to ensure that your design is correctly implemented in your software.
Appendix

References:

Code, Build & Debug

Demos:

- Debug Work-Bench
- Debug session using Java
- Debug using C#

Execution Analysis

- Hints on recording Sequence diagrams

Testing Cycle:

- Integrate NUnit into Enterprise Architect
- Testing Whitepaper

Download site for xUnit testing:
- NUnit
- JUnit

Help references

- Integrated Development options