Enterprise Architect

User Guide Series

Software Engineering

How to model software engineering? Sparx Systems Enterprise Architect has tools to model software in many coding languages, and to automate code generation, reverse engineer code, synchronize code with a model, analyze execution and trace development.

Author: Sparx Systems
Date: 2026-05-04
Version: 17.1

CREATED WITH ENTERPRISE ARCHITECT

Table of Contents

Software Engineering 11
Getting Started 14
Example Diagram 18
Integrated Development 19
Feature Overview 22
Generate Source Code 25
Generate a Single Class 30
Generate a Group of Classes 33
Generate a Package 35
Update Package Contents 40
Synchronize Model and Code 43
Namespaces 45
Importing Source Code 47
Import Projects 51
Import Source Code 56
Notes on Source Code Import 58
Import Resource Script 63
Import a Directory Structure 66
Import Binary Module 70
Classes Not Found During Import 72
Editing Source Code 73
Languages Supported 79
Configure File Associations 81
Compare Editors 83

Code Editor Toolbar 86
Code Editor Context Menu 93
Create Use Case for Method 100
Code Editor Functions 103
Function Details 104
Intelli-sense 109
Find and Replace 112
Search in Files 119
Find File 125
Search Intelli-sense 129
Code Editor Key Bindings 134
Application Patterns (Model + Code) 142
MDG Integration and Code Engineering 146
Behavioral Model Code Generation 147
Code Generation - Activity Diagrams 152
Code Generation - Interaction Diagrams 155
Code Generation - StateMachines 157
Legacy StateMachine Templates 167
Java Code Generated From Legacy StateMachine Template 171
StateMachine Modeling For HDLs 180
Win32 User Interface Dialogs 183
Modeling UI Dialogs 186
Import Single Dialog from RC File 190
Import All Dialogs from RC File 191
Export Dialog to RC File 192
Design a New Dialog 193

Gang of Four (GoF) Patterns 196
ICONIX 199
Configuration Settings 202
Source Code Engineering Options 203
Code Generation Options 206
Import Component Types 209
Source Code Options 211
Options - Code Editors 215
Editor Language Properties 218
Options - Object Lifetimes 223
Options - Attribute/Operations 225
Modeling Conventions 228
ActionScript Conventions 231
Ada 2012 Conventions 234
C Conventions 239
Object Oriented Programming In C 244
C# Conventions 248
C++ Conventions 255
Managed C++ Conventions 260
C++/CLI Conventions 262
Delphi Conventions 265
Java Conventions 268
AspectJ Conventions 272
PHP Conventions 274
Python Conventions 276
SystemC Conventions 278
VB.NET Conventions 282

Verilog Conventions 287
VHDL Conventions 291
Visual Basic Conventions 297
Language Options 300
ActionScript Options - User 303
ActionScript Options - Model 305
Ada 2012 Options - User 307
Ada 2012 Options - Model 309
ArcGIS Options - User 311
ArcGIS Options - Model 313
C Options - User 315
C Options - Model 318
C# Options - User 321
C# Options - Model 323
C++ Options - User 326
C++ Options - Model 329
Delphi Options - User 332
Delphi Options - Model 334
Delphi Properties 336
Java Options - User 338
Java Options - Model 340
MySQL Options - User 343
MySQL Options - Model 345
PHP Options - User 347
PHP Options - Model 349
Python Options - User 351
Python Options - Model 353

SystemC Options - User 355
SystemC Options - Model 357
Teradata Options - User 359
Teradata Options - Model 361
VB.NET Options - User 363
VB.NET Options - Model 365
Verilog Options - User 367
Verilog Options - Model 369
VHDL Options - User 371
VHDL Options - Model 373
Visual Basic Options - User 375
Visual Basic Options - Model 377
MDG Technology Language Options 379
Reset Options 381
Set Collection Classes 384
Example Use of Collection Classes 387
Local Paths 391
Local Paths Dialog 393
Language Macros 396
Developing Programming Languages 400
Code Template Framework 403
Code Template Customization 406
Code and Transform Templates 408
Base Templates 412
Export Code Generation and Transformation Templates 418
Import Code Generation and Transformation

Templates 420
Synchronize Code 422
Synchronize Existing Sections 425
Add New Sections 426
Add New Features and Elements 427
The Code Template Editor 428
Create New Custom Template 431
Code Template Syntax 433
Literal Text 434
Variables 436
Macros 440
Template Substitution Macros 442
Field Substitution Macros 445
Substitution Examples 447
Attribute Field Substitution Macros 450
Class Field Substitution Macros 453
Code Generation Option Field Substitution Macros 459
Connector Field Substitution Macros 469
Constraint Field Substitution Macros 478
Effort Field Substitution Macros 480
File Field Substitution Macros 481
File Import Field Substitution Macros 483
Link Field Substitution Macros 486
Linked File Field Substitution Macros 489
Metric Field Substitution Macros 491
Operation Field Substitution Macros 492

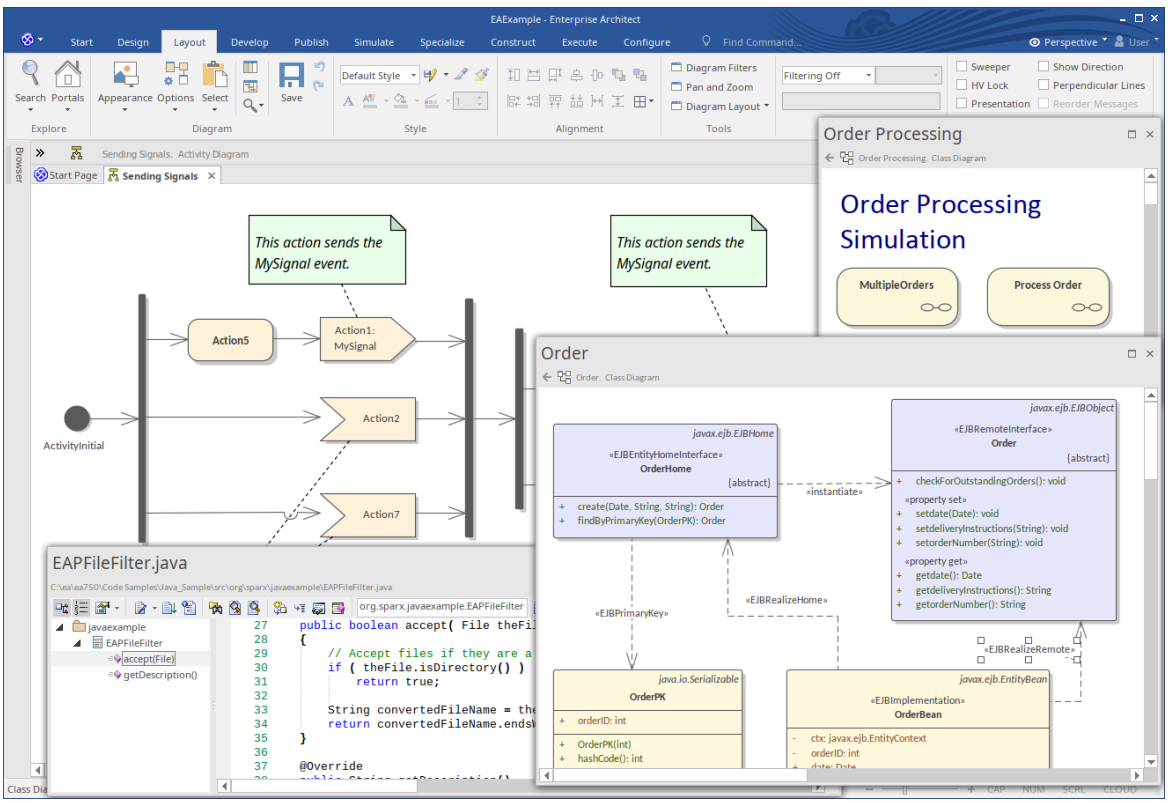
Package Field Substitution Macros 495
Parameter Field Substitution Macros 497
Problem Field Substitution Macros 499
Requirement Field Substitution Macros 501
Resource Field Substitution Macros 503
Risk Field Substitution Macros 505
Scenario Field Substitution Macros 506
Tagged Value Substitution Macros 507
Template Parameter Substitution Macros 510
Test Field Substitution Macros 512
Function Macros 514
Control Macros 525
List Macro 526
Branching Macros 529
Synchronization Macros 533
The Processing Instruction (PI) Macro 535
Code Generation Macros for Executable StateMachines 537
EASL Code Generation Macros 560
EASL Collections 566
EASL Properties 572
Call Templates From Templates 585
The Code Template Editor in MDG Development 587
Create Custom Templates 588
Customize Base Templates 591
Add New Stereotyped Templates 593
Override Default Templates 596

Grammar Framework 598
Grammar Syntax 600
Grammar Instructions 603
Grammar Rules 605
Grammar Terms 607
Grammar Commands 609
AST Nodes 614
Editing Grammars 632
Parsing AST Results 635
Profiling Grammar Parsing 637
Macro Editor 639
Example Grammars 641
Code Analyzer 642
Code Miner Framework 655
Code Miner Libraries 657
Creating a New Code Miner Database 662
Code Miner Queries 671
Code Miner Query Language (mFQL) 673
The mFQL Language 676
Set Extraction 697
Set Traversal 700
Set Joining 703
Sparx Intel Service 707
Sparx Intel Service Configuration 708
Sparx Intel Service Automatic Update 718
Service Configuration 722
Client Configuration - Configuring Enterprise

Software Engineering

Create and Manage Effective and Productive Structural and Behavioral Models of Software

Software engineering is the discipline of designing, implementing and maintaining software. The process of software engineering starts with requirements and constraints as inputs, and results in programming code and schemas that are deployed to a variety of platforms, creating running systems.



Enterprise Architect has a rich set of tools and features that assist Software Engineers to perform their work efficiently and reduce the number of errors in implemented solutions.

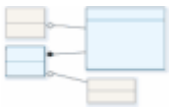
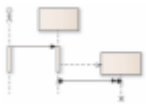
The features include design tools to create models of software, automated code generation, reverse engineering of source code, binaries and schemas, and tools to synchronize source code with the design models. The programming code can be viewed and edited directly in the integrated Code Editors within Enterprise Architect, which provide Intelli-sense and other features to aid in coding.

Another compelling aspect of the environment is the ability to trace the implementation Classes back to design elements and architecture, and then back to the requirements and constraints and other specifications, and ultimately back to stakeholders and their goals and visions.

Enterprise Architect supports a wide range of programming languages and platforms and provides a lightweight and seamless integration with the two most prevalent Integrated Development Environments: Visual Studio and Eclipse. In addition there is a fully featured Execution Analyzer that allows the Software Engineer to design, build debug and test software modules right inside Enterprise Architect.

Facilities

Facility	Description
Development Tools	Discover the tightly Integrated Development Environment with outstanding tools and functionality.

Code, Build and Debug 	Model, develop, debug, profile and manage an application from within the modeling environment.
Visual Analysis of Executing Code 	Understand your code base by visually analyzing running code. Use Test Points, profiling and automated diagram generation.
Generate Source Code	Explore some of the ways to generate source code for a single Class, a selection of Classes, or a whole Package. Generate from structural or behavioral models.
Importing Source Code	Examine existing systems by importing source code into Enterprise Architect. View and modify dialog definitions. Synchronize the model with the latest updates to source code.

Getting Started

Enterprise Architect offers a comprehensive suite of tools and features designed to enhance the productivity and accuracy of Software Engineers. These tools assist in creating software models, generating code automatically, reverse engineering source code and schemas, and synchronizing code with design models.

Configuration Settings

Selecting the Perspective

Enterprise Architect partitions the tool's extensive features into Perspectives, which ensures that you can focus on a specific task and work with the tools you need without the distraction of other features. To work with Software Model features you first need to select one of these Perspectives:

The Software Engineering Set:

<perspective name> > Software Engineering > Code Engineering

<perspective name> > Software Engineering > GoF Patterns

<perspective name> > Software Engineering > ICONIX

The UX Design Set:

<perspective name> > UX Design > Win 32 UI Models

Setting the Perspective ensures that the Case Management Model and Notation diagrams, their tool boxes and other features of the Perspective will be available by default.

Example Diagram

An example diagram provides a visual introduction to the topic and allows you to see some of the important elements and connectors that you use to specify or describe classes for the visualization of software and the forward and reverse engineering to and from a wide range of programming languages.

Integrated Development

In this topic you will learn how to use the fully featured integrated development environment. You will learn how to create structural and behavioral models of software artifacts in a rich code editor, generate and reverse engineer code, customize the way code is generated, run analyzer scripts to optimize code, use the debugger and set units test and much more.

Behavioral Models

In this topic you will learn how to generate code for

software, system and hardware description languages directly from behavioral diagrams including: StateMachine, Sequence and Activity Diagrams. This will add new dimensions and precisions to the way you work with software and engineering systems.

Gang of Four (GoF) Patterns

This topic introduces the renowned twenty-three design patterns collected together as the Gang of Four (GoF) patterns which refers to their four authors. You will have at hand the solutions to common problems facing software engineers and be able to inject these patterns into your own models adding to the quality and rigor to your software systems.

Win32 User Interface Dialogs

In this topic you will learn how to work with Enterprise Architect's User Interface modeling capability that allows you to model user interface screens using Win32® controls. The models can be forward or reverse engineered and can also provide an interface for StateMachine and Activity diagram simulation, allowing them to receive and process user input.

Code Template Framework

In this topic you will learn how to work with the Code

Template Framework which governs how models and converted to code. There are a standard set of templates but you can extended these to create your own templates and to generate code to suit your needs. There are also templates that control transformations and the generation of Database Definition Language (DDL.

Grammar Framework

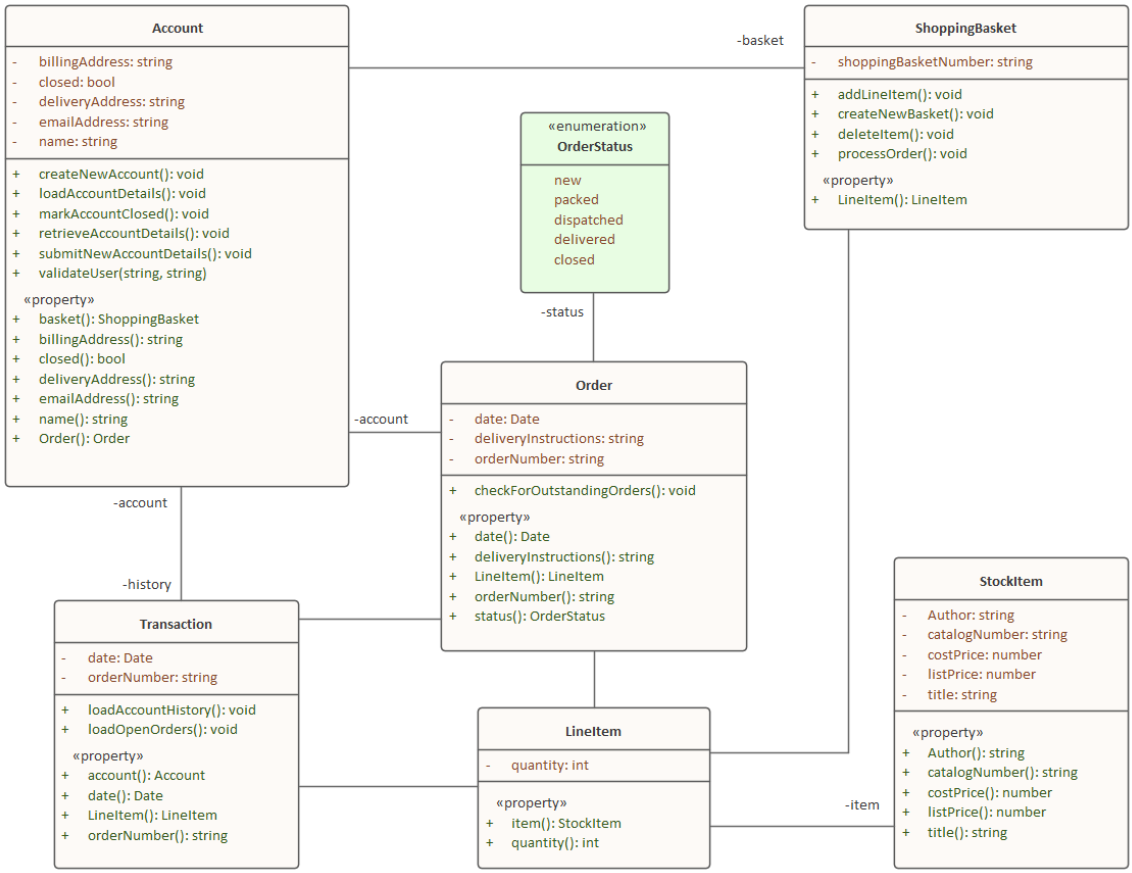
In this topic you will learn how to create a grammar to convert an unsupported programming language into a UML model. Enterprise Architect has built in support for a wide range of programming languages but if you need to work with an unsupported language you can use the Grammar Framework to write your own parser. The grammar is used to reverse engineer programming code in the form of text and is the direct compliment of the Code Template Framework which you would you to specify how a UML model for an unsupported language is converted to code.

Code Miner Framework

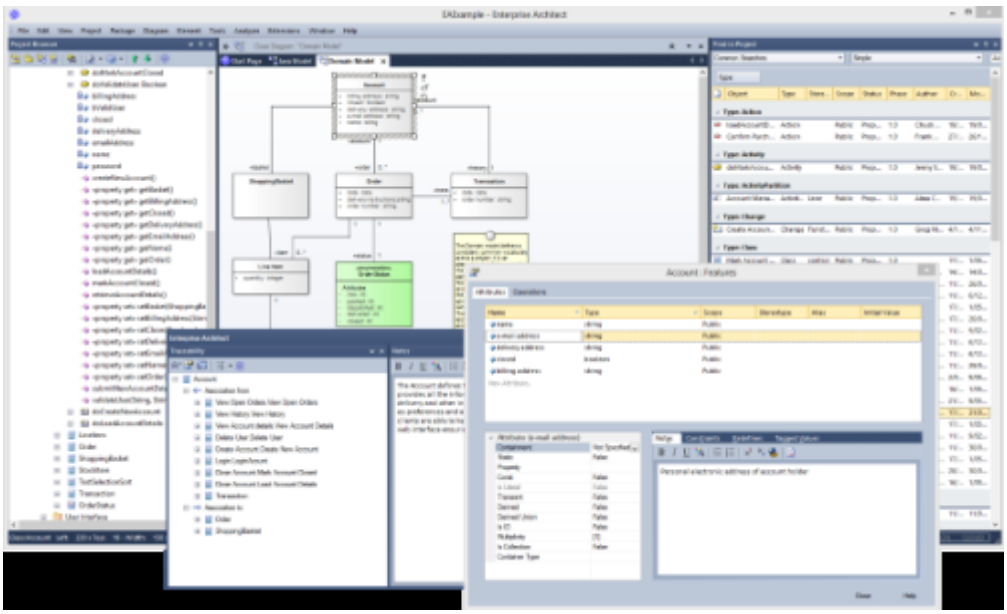
In this topic you will learn how to work with a database of source code which provides access to the data hidden within source code in a timely and effective manner. Source code is parsed creating a tree structure which can be used to analyze program structure, calculate metrics, trace relationships and even perform refactoring.

Example Diagram

Software diagrams allow you to model the structure and behavior of software including User Interfaces. Enterprise Architect has at its core fundamental support for modelling software and the tool supports a wide range of programming languages and paradigms. In this diagram we see Classes used to model an online shop, including Classes that contain compartments for Attributes, Operations and Properties. An Enumeration has also been used to model Order Status.



Integrated Development



Enterprise Architect provides an unmatched set of tools and features for the Software Engineer, to assist in the process of creating robust and error free software systems. The engineer can start by defining the architecture and ensuring that it traces back to the requirements and specification. Technology neutral models can be transformed to target a comprehensive range of programming languages. The Model Driven Development Environment fits the bill for various technologies.

Features

Development Tools	• Model driven development with best-in-class UML tools • Generate and reverse engineer code • Customize code generation with

templates
- Analyzer Scripts to manage your applications
- Code editors to author the code base
- Debuggers to investigate behavior
- Profilers to visualize behavior
- Analyzers to record behavior
- Testpoints for validation of programming contracts
- Integration with jUnit and nUnit
- Eclipse or Visual Studio Integration where required

Traceability At a glance traceability of Generalizations, Realizations, Associations, Dependencies and more. Customize relationship views. Easily navigate related elements in the model.

Usage Quickly browse element usage across all diagrams. Perform effective element searches using sophisticated queries.

Popular Languages
- C/ C++
- Java
- Microsoft .NET family
- ADA

- Python
- Perl
- PHP

Toolboxes Toolboxes are provided for a vast array of modeling technologies and programming languages.

Application Patterns Enterprise Architect provides complete starter projects, including model information, code and build scripts, for several basic application types.

Feature Overview

Code Engineering with Enterprise Architect broadly encompasses various processes for the design, generation and transformation of code from your UML model.

Features

Model Driven Code Engineering	• Source code generation and reverse engineering for many popular languages, including C++, C#, Java, Delphi, VB.Net, Visual Basic, ActionScript, Python and PHP • A built in 'syntax highlighting' source code editor • Code generation templates, which enable you to customize the generated source code to your company specifications
Transformations for Rapid Development	• Advanced Model Driven Architecture (MDA) transformations using transformation templates • Built-in transformations for DDL, C#, Java, EJB and XSD • One Platform Independent Model can be used to generate and synchronize

multiple Platform Specific Models, providing a significant productivity boost
- XSL Transform diagram, toolbox, editor and debugger.

Visual Execution Analysis / Debugging, Verification and Visualization

- Execute build, test, debug, run and deploy scripts
- Integrate UML development and modeling with source development and compilation
- Generate NUnit and JUnit test Classes from source Classes using MDA Transformations
- Integrate the test process directly into the Enterprise Architect IDE
- Debug .NET, Mono, Java and Microsoft Native (C, C++ and Visual Basic) applications
- Design and execute Test suites based on Programming by Contract principles
- XSL Stylesheet debugging

Database Modeling

Enterprise Architect enables you to:

- Reverse engineer from many popular DBMSs, including SQL Server, My SQL, Access, PostgreSQL and Oracle
- Model database tables, columns, keys,

foreign keys and complex relationships using UML and an inbuilt data modeling profile

- Forward generate DDL scripts to create target database structures

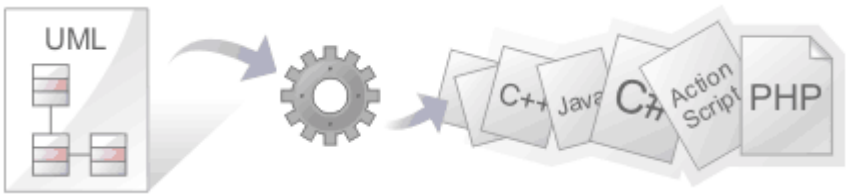
XML Technology Engineering

Enterprise Architect enables you to rapidly model, forward engineer and reverse engineer two key W3C XML technologies:

- XML Schema (XSD)
- Web Service Definition Language (WSDL)

XSD and WSDL support is critical for the development of a complete Service Oriented Architecture (SOA), and the coupling of UML 2.5 and XML provides the natural mechanism for implementing XML-based SOA artifacts within an organization.

Generate Source Code



Source code generation is the process of creating programming code from a UML model. There are great benefits in taking this approach as the source code Packages, Classes and Interfaces are automatically created and elaborated with variables and methods.

Enterprise Architect can also generate code from a number of behavioral models, including StateMachine, Sequence and Activity diagrams. There is a highly flexible template mechanism that allows the engineer to completely tailor the way that source code is generated, including the comment headers in methods and the Collection Classes that are used.

From an engineering and quality perspective, the most compelling advantage of this approach is that the UML models and therefore the architecture and design are synchronized with the programming code. An unbroken traceable path can be created from the goals, business drivers and the stakeholder's requirements right through to methods in the programming code.

Facilities

Facility	Description

Languages	Enterprise Architect supports code generation in each of these software languages: • Action Script • Ada • ArcGIS • C • C# (for .NET 1.1, .NET 2.0 and .NET 4.0) • C++ (standard, plus .NET managed C++ extensions) • Delphi • Java (including Java 1.5, Aspects and Generics) • JavaScript • mFQL • MySql • PHP • Python • Teradata SQL • Visual Basic • Visual Basic .NET • WorkFlowScript You can also generate Hardware Definition Language code in these languages:

	• VHDL • Verilog • SystemC
Elements	Code is generated from Class or Interface model elements, so you must create the required Class and Interface elements to generate from. All other types of element to contribute to the code (such as StateMachines or Activities) must be child elements of a Class. Add attributes (which become variables) and operations (which become methods). Constraints and Receptions are also supported in the code.
Settings	Before you generate code, you should ensure the default settings for code generation match your requirements; set up the defaults to match your required language and preferences. Preferences that you can define include default constructors and destructors, methods for interfaces and the Unicode options for created languages. Languages such as Java support 'namespaces' and can be configured to specify a namespace root. In addition to the default settings for

	generating code, Enterprise Architect facilitates setting specific generation options for each of the supported languages.
Code Template Framework	The Code Template Framework (CTF) enables you to customize the way Enterprise Architect generates source code and also enables generation of languages that are not specifically supported by Enterprise Architect.
Local Paths	Local path names enable you to substitute tags for directory names.
Behavioral Code	You can also generate software code from three UML behavioral modeling paradigms: • Interaction (Sequence) diagrams • Activity diagrams • StateMachine diagrams (using Legacy StateMachine Templates in the code generation operations under 'Tasks') • StateMachine diagrams (using an Executable StateMachine Artifact)
Live Code Generation	On the 'Develop > Source Code > Options' drop-down menu, you have the option to update your source code

	instantly as you make changes to your model.
Tasks	When you generate code, you perform one or more of these tasks: • Generate a Single Class • Generate a Group of Classes • Generate a Package • Update Package Contents

Notes

- Most of the tools provided by Enterprise Architect for code engineering and debugging are available in the Professional and higher editions of Enterprise Architect; Behavioral Code Generation is available in the Unified and Ultimate Editions
- When security is enabled you require the access permissions 'Generate Source Code and DDL' and 'Reverse Engineer from DDL and Source Code'

Generate a Single Class

Before you generate code for a single Class, you:

- Complete the design of the model element (Class or Interface)
- Create Inheritance connectors to parents and Associations to other Classes that are used
- Create Inheritance connectors to Interfaces that your Class implements; the system provides an option to generate function stubs for all interface methods that a Class implements

Generate code for a single Class

Step	Action
1	Open the diagram containing the Class or Interface for which to generate code.
2	Click on the required Class or Interface and select the 'Develop > Source Code > Generate > Generate Single Element' ribbon option, or press F11. The 'Generate Code' dialog displays, through which you can control how and where your source code is generated.
	In the 'Path' field, click on the button and select a

3	path name for your source code to be generated to.
4	In the 'Target Language' field, click on the drop-down arrow and select the language to generate; this becomes the permanent option for that Class, so change it back if you are only doing one pass in another language.
5	Click on the Advanced button. The 'Object Options' dialog displays, providing subsets of the 'Source Code Engineering' and code language options pages on the 'Preferences' dialog.
6	Set any custom options (for this Class alone), then click on the Close button to return to the 'Generate Code' dialog.
7	In the 'Import(s) / Header(s)' fields, type any import statements, #includes or other header information. Note that in the case of Visual Basic this information is ignored; in the case of Java the two import text boxes are merged; and in the case of C++ the first import text area is placed in the header file and the second in the body (.cpp) file.
8	Click on the Generate button to create the source code.
9	When complete, click on the View button to see

	what has been generated. Note that you should set up your default viewer/editor for each language type first; you can also set up the default editor on the 'Code Editors' page of the Preferences window ('Start > Application > Preferences > Preferences > Source Code Engineering > Code Editors').

Generate a Group of Classes

In addition to being able to generate code for an individual Class, you can also select a group of Classes for batch code generation. When you do this, you accept all the default code generation options for each Class in the set.

Generate Class Group

Step	Detail
1	Select a group of Classes and/or interfaces in a diagram.
2	Click on an element in the group and select the 'Develop > Source Code > Generate > Generate Selected Element(s)' ribbon option (or press Shift+F11). If no code exists for the selected elements, the 'Save As' dialog displays on which you specify the file path and name for each code file; enter this information and click on the Save button.
3	The 'Batch Generation' dialog displays, showing the status of the process as it executes (the process might be too fast to see this dialog). If code already exists for the selected Class

	elements, and changes have been made to the Class name or structure, the 'Synchronize Element <package name>.<element name>' dialog might also display; this dialog helps synchronize the model and code.

Notes

- If any of the elements selected are not Classes or interfaces the option to generate code is not available

Generate a Package

In addition to generating source code from single Classes and groups of Classes, you can generate code from a Package. This feature provides options to recursively generate code from child Packages and automatically generate directory structures based on the Package hierarchy. This helps you to generate code for a whole branch of your project model in one step.

Access

Ribbon	Develop > Source Code > Generate > Generate All
Keyboard Shortcuts	Ctrl+Alt+K

Generate code from a Package, on the Generate Package Source Code dialog

Step	Action
1	In the 'Synchronize' field, click on the drop-down

	arrow and select the appropriate synchronize option: • 'Synchronize model and code': Code for Classes with existing files is forward synchronized with that file; code for Classes with no existing file is generated to the displayed target file • 'Overwrite code': All selected target files are overwritten (forward generated) • 'Do not generate': Generate code for only those selected Classes that do not have an existing file; all other Classes are ignored
2	Highlight the Classes for which to generate code; leave unselected any to not generate code for. If you want to display more of the information within the layout, you can resize the dialog and its columns.
3	To make Enterprise Architect automatically generate directories and filenames based on the Package hierarchy, select the 'Auto Generate Files' checkbox; this enables the 'Root Directory' field, in which you select a root directory under which the source directories are to be generated. By default, the 'Auto Generate Files' feature ignores any file paths that are already associated with a Class; you can change this behavior by also selecting the 'Retain Existing File Paths' checkbox.
	To include code for all sub-Packages in the output,

4	select the 'Include Child Packages' checkbox.
5	Click on the Generate button to start generating code. As code generation proceeds, Enterprise Architect displays progress messages. If a Class requires an output filename the system prompts you to enter one at the appropriate time (assuming Auto Generate Files is not selected). For example, if the selected Classes include partial Classes, a prompt displays to enter the filename into which to generate code for the second partial Class.

Further information on the dialog options

Option	Action
Root Package	Check the name of the Package for which code is to be generated.
Synchronize	Select options that specify how existing files should be regenerated.
Auto Generate Files	Specify whether Enterprise Architect should automatically generate file names and directories, based on the Package hierarchy.

Root Directory	If Auto Generate Files is selected, display the path under which the generated directory structures are created.
Retain Existing File Paths	If Auto Generate Files is selected, specify whether to use existing file paths associated with Classes. If Auto Generate Files is unselected, Enterprise Architect generates Class code to automatically determined paths, regardless of whether source files are already associated with the Classes.
Include all Child Packages	Also generate code for all Classes in all sub-Packages of the target Package in the list. This option facilitates recursive generation of code for a given Package and its sub-Packages.
Select Objects to Generate	List all Classes that are available for code generation under the target Packages; only code for selected (highlighted) Classes is generated. Classes are listed with their target source file.
Select All	Mark all Classes in the list as selected.

Select None	Mark all Classes in the list as unselected.
Generate	Start the generation of code for all selected Classes.
Cancel	Exit the 'Generate Package Source Code' dialog; no Class code is generated.

Update Package Contents

In addition to generating and importing code, Enterprise Architect provides the option to synchronize the model and source code, creating a model that represents the latest changes in the source code and vice versa. You can use either the model as the source, or the code as the source.

The behavior and actions of synchronization depend on the settings you have selected on the 'Attributes and Operations' page of the 'Preferences' dialog. Working with these settings, you can either protect or automatically discard information in the model that is not present in the code, and prompt for a decision on code features that are not in the model. In these two examples, the appropriate checkboxes have been selected for maximum protection of data:

- You generated some source code, but made subsequent changes to the model; when you generate code again, Enterprise Architect adds any new attributes or methods to the existing source code, leaving intact what already exists, which means developers can work on the source code and then generate additional methods as required from the model, without having their code overwritten or destroyed
- You might have made changes to a source code file, but the model has detailed notes and characteristics you do not want to lose; by synchronizing from the source code into the model, you import additional attributes and methods but do not change other model elements

Using the synchronization methods, it is simple to keep source code and model elements up to date and synchronized.

Access

Ribbon	Develop > Source Code > Synchronize > Synchronize Package

Synchronize Package contents against source code

Field/Button	Action
Update Type	Select the radio button to either Forward Engineer or Reverse Engineer the Package Classes, as appropriate.
Include child packages in generation	Select the checkbox to include child Packages in the synchronization.
OK	Click on the button to start synchronization. Enterprise Architect uses the directory

	names specified when the project source was first imported/generated and updates either the model or the source code depending on the option chosen. If: • Performing forward synchronization AND • There are differences between the model and code AND • The 'On forward synch, prompt to delete code features not in model' checkbox is selected in the 'Options - Attributes and Operations' dialog THEN the 'Synchronize Element <package name>.<element name>' dialog displays. Otherwise, no further action is required.

Notes

- Code synchronization does not change method bodies; behavioral code cannot be synchronized, and code generation only works when generating the entire file
- In the Corporate, Unified and Ultimate Editions of Enterprise Architect, if security is enabled you must have 'Generate Source Code and DDL' permission to synchronize source code with model elements

Synchronize Model and Code

You might either:

- Synchronize the code for a Package of Classes against the model in the Browser window, or
- Regenerate code from a batch of Classes in the model

In such processes, there might be items in the code that are not present in the model.

If you want to trap those items and resolve them manually, select the 'On forward synch, prompt to delete code features not in model' checkbox in the 'Options - Attributes and Operations' dialog, so that the 'Synchronize Element <package name>.<element name>' dialog displays, providing options to respond to each item.

Synchronize Items

Button	Detail
Select All	Highlight and select all items in the Feature column.
Clear All	Deselect and remove highlighting from all items in the Feature column.
Delete	Mark the selected code features to be removed from the code (the value in the

	Action column changes to Delete).
Reassign	Mark the selected code features to be reassigned to elements in the model. This is only possible when an appropriate model element is present that is not already defined in the code. The Select the Corresponding Class Feature dialog displays, from which you select the Class to reassign the feature to. Click on the OK button to mark the feature for reassignment.
Ignore	Mark the selected code elements not present in the model to be ignored completely (the default; the value in the Action column remains as or changes to <none>).
Reset to Default	Reset the selected items to Ignore (the value in the Action column changes to <none>).
OK	Make the assigned changes to the items, and close the dialog.

Namespaces

Languages such as Java support Package structures or namespaces. In Enterprise Architect you can specify a Package as a namespace root, which denotes where the namespace structure for your Class model starts; all subordinate Packages below a namespace root will form the namespace hierarchy for contained Classes and Interfaces.

To define a Package as a namespace root, click on the Package in the Browser window and select the 'Develop > Source Code > Options > Set as Namespace Root' ribbon option. The Package icon in the Browser window changes to show a colored corner indicating this Package is a namespace root.

Generated Java source code, for example, will automatically add a Package declaration at the beginning of the generated file, indicating the location of the Class in the Package hierarchy below the namespace root.

To clear an existing namespace root, click on the namespace root Package in the Browser window and deselect the 'Develop > Source Code > Options > Set as Namespace Root' ribbon option

To view a list of namespaces, select the 'Settings > Reference Data > Settings > Namespace Roots' ribbon option; the 'Namespaces' dialog displays. If you double-click on a namespace in the list, the Package is highlighted in the Browser window; alternatively, right-click on the namespace and select the 'Locate Package

in Browser' option.

You can also clear the selected namespace root by selecting the 'Clear Namespace Attribute' option.

To omit a subordinate Package from a namespace definition, select the 'Develop > Source Code > Options > Suppress Namespace' ribbon option; to include the Package in the namespace again, deselect the ribbon option.

Notes

- When performing code generation, any Package name that contains whitespace characters is automatically treated as a namespace root

Importing Source Code

The ability to view programming code and the models it is derived from at the same time brings clarity to the design of a system. One of Enterprise Architect's convenient code engineering features is the ability to Reverse Engineer source code into a UML model. A wide range of programming languages are supported and there are options that govern how the models are generated. Once the code is in the model it is possible to keep it synchronized with the model regardless of whether the changes were made directly in the code or the model itself. The code structures are mapped into their UML representations; for example, a Java class is mapped into a UML Class element, variables are defined as attributes, methods modeled as operations, and interactions between the Java classes represented by the appropriate connectors.

The representation of the programming code as model constructs helps you to gain a better understanding of the structure of the code and how it implements the design, architecture and the requirements, and ultimately how it delivers the business value.

It is important to note that if a system is not well designed, simply importing the source into Enterprise Architect does not turn it into an easily understandable UML model. When

working with a poorly designed system it is useful to assess the code in manageable units by examining the individual model Packages or elements generated from the code; for example, dragging a specific Class of interest onto a diagram and then using the 'Insert Related Elements' option at one level to determine the immediate relationships between that Class and other Classes. From this point it is possible to create Use Cases that identify the interaction between the source code Classes, providing an overview of the application's operation.

Several options guide how the code is reversed engineered, including whether comments are imported to notes and how they are formatted, how property methods are recognized and whether Dependency relationships are created for operation return and parameter types.

Copyright Ownership

Situations that typically lend themselves to reverse engineering tend to operate on source code that:

- You have already developed
- Is part of a third-party library that you have obtained permission to use
- Is part of a framework that your organization uses
- Is being developed on a daily basis by your developers

If you are examining code that you or your organization do not own or do not have specific permission to copy and edit, you must ensure that you understand and comply with the copyright restrictions on that code before beginning the

process of reverse engineering.

Supported languages for Reverse Engineering

Language
Action Script
Ada 2012 (Unified and Ultimate Editions)
C
C#
C++
CORBA IDL (MDG Technology)
Delphi
Java
PHP
Python

SystemC (Unified and Ultimate Editions)
Verilog (Unified and Ultimate Editions)
VHDL (Unified and Ultimate Editions)
Visual Basic
Visual Basic .NET

Notes

- Reverse Engineering is supported in the Professional, Corporate, Unified and Ultimate Editions of Enterprise Architect
- If security is enabled you must have 'Reverse Engineer From DDL And Source Code' permission to reverse engineer source code and synchronize model elements against code
- Using Enterprise Architect, you can also import certain types of binary file, such as Java .jar files and .NET PE files
- Reverse Engineering of other languages is currently available through the use of MDG Technologies listed on the MDG Technology pages of the Sparx Systems website

Import Projects

Enterprise Architect provides support for importing software projects authored in Visual Studio, Mono, Eclipse and NetBeans. Importing and working on projects in Enterprise Architect has multiple benefits, not least the immediate access to Enterprise Architect's renowned modeling tools and management features, but also the access to development tools such as simulation, debugging and profiling.

Access

Ribbon	Develop > Source Code > Solutions > Import a <project type>

Import a Visual Studio Solution

This option allows you to import one or more projects from an existing Visual Studio Solution file or a running instance of Visual Studio. The wizard will generate a Class model for each of the projects and the appropriate Analyzer Scripts for each Visual Studio configuration.

Import a Mono Solution

This option allows you to import Mono projects from a solution file. The dialog that is presented is the same as the 'Visual Studio Import' dialog, but you can choose to target either Linux or Windows. The wizard will generate a Class model for each of the projects and configure them for debugging. The generated Analyzer Scripts reference msbuild to build the projects.

Import an Eclipse Project

The Eclipse 'Wizard' can reverse engineer a Java project described by its Eclipse .project file and ANT build. The feature will result in a UML Class model and Analyzer Scripts for each of the ANT targets you select. The process will also generate a script for each debug protocol you select through the 'Wizard'. You will be presented with the choice of JDWP (Java Debug Wire Protocol), good for servers, and JVMTI (Java Virtual Machine Tools Interface), which is suited to standalone Java applications. These scripts should be used for debugging the project in Enterprise Architect.

Import a NetBeans Project

The NetBeans 'Wizard' can reverse engineer a Java project described by a NetBeans XML project file and ANT build. The 'Wizard' will create a UML Class model of the project and Analyzer Scripts for each of the ANT targets you select. The process will also generate a script for each debug protocol you select through the 'Wizard'. These scripts

should be used for debugging the project in Enterprise Architect. You will be presented with the choice of JDWP (Java Debug Wire Protocol), good for servers, and JVMTI (Java Virtual Machine Tools Interface), which is suited to standalone Java applications.

Import Options

When you select to import a Visual Studio or Mono Solution, the 'Visual Studio Solution Import' dialog displays. Complete the fields as directed in this table.

When you select to import an Eclipse or Netbeans solution, the appropriate Wizard start screen displays. Work through the screens as directed by the prompts on each screen.

Option	Description
<list of projects>	After you have selected the solution file, the projects in the solution are listed in the panel. Select the projects to be imported by the Wizard. You can use the All button to select all projects, and the None button to clear the selection of projects.
Select Solution File	Browse for and select the Solution file to import from. The Mono Solution files and Visual Studio Solution files have a .sln file extension.

Perform a Dry Run	Select this option to perform the import as a dry run, to check for any errors in the process or output before you repeat the import to change the model content. Click on the View Log button to check the log of the import.
Create Package per File	Select this option to perform the import with finer granularity, creating a separate Package for each file.
Import	Click on this button to start the import process.
Prompt for Missing Macro Definitions	Not applicable to Mono Solution imports. For C++ projects in Visual Studio, the parser might encounter unrecognized macros. If you select this option, you will be prompted when such an event occurs and will have the opportunity to define the macro. If you do not select this option, the resultant Class model could be missing certain items.
Create Diagram for Each Package	When selected, a Class diagram is created depicting the Class model for each Package. The result is a larger but more colorful model. Deselecting this option

	will cause diagram creation to be skipped and the import to run faster.
Generate Analyzer Scripts	For Visual Studio Solutions, selecting this option will generate Analyzer Scripts for each project configuration in addition to scripts for each Solution configuration. The scripts will allow for building and debugging the program(s) described by the solution immediately after the import completes. Select the 'Windows' checkbox; if you do not select this option, no Execution Analyzer features will be configured. For Mono Solutions, this option allows you to target either Linux or Windows. If you select Linux, it is assumed the machine on which Enterprise Architect is running is Linux, that the platform (Java or Mono) is installed there, and that the compiled programs run on Linux.
Startup Project	When this option is selected, the script for this Project will become the model default. The debugging tools, Execute ribbon and Toolbar buttons will automatically target this program.

Import Source Code

You can import source code into your Enterprise Architect model, to reverse-engineer a module. As the import proceeds, Enterprise Architect provides progress information. When all files are imported, Enterprise Architect makes a second pass to resolve associations and inheritance relationships between the imported Classes.

Procedure - Import source code

Step	Action
1	In the Browser window, select (or add) a diagram into which to import the Classes.
2	Click on the diagram background and either: • Select the 'Develop > Source Code > Files' ribbon option and click on the appropriate language, or • If the Code Generation toolbar is displayed, click on the 'Import' drop-down arrow and select the language to import The list of languages will include any customized languages you have created model structures for.
3	From the file browser that appears, locate and select one or more source code files to import.

4	Click on the Open button to start the import process.

Notes on Source Code Import

You can import code into your Enterprise Architect project, in a range of programming languages. Enterprise Architect supports most constructs and keywords for each coding language. You select the appropriate type of source file for the language, as the source code to import.

If there is a particular feature you require support for that you feel is missing, please contact Sparx Systems.

Notes

- When reverse engineering attributes with parameter substitutions (templated attributes):
  - If a Class with proper template parameter definitions is found, an Association connector is created and its parameter substitutions are configured
  - An Association connector is also created if a matching entry is defined as a Collection Class or in the 'Additional Collection Classes' option (for C#, C++ and Java); for an example, see *Example Use of Collection Classes*

Programming Language notes

Language	Notes
ActionScript	Appropriate type of source file: .as code

	file.
C	Appropriate type of source file: .h header files and/or .c files. When you select a header file, Enterprise Architect automatically searches for the corresponding .c implementation file to import, based on the options for extension and search path specified in the C options. Enterprise Architect does not expand macros that have been used, these must be added into the internal list of Language Macros.
C++	Appropriate type of source file: .h header file. Enterprise Architect automatically searches for the .cpp implementation file based on the extension and search path set in the C++ options; when it finds the implementation file, it can use it to resolve parameter names and method notes as necessary. When importing C++ source code, Enterprise Architect ignores function pointer declarations. To import them into your model you could create a typedef to define a

	function pointer type, then declare function pointers using that type; function pointers declared in this way are imported as attributes of the function pointer type. Enterprise Architect does not expand macros that have been used; these must be added into the internal list of Language Macros.
C#	Appropriate type of source file: .cs.
Delphi	Appropriate type of source file: .pas.
Java	Appropriate type of source file: .java. Enterprise Architect supports the AspectJ language extensions. Aspects are modeled using Classes with the stereotype aspect; these aspects can then contain attributes and methods as for a normal Class. If an intertype attribute or operation is

	required, you can add a tag 'className' with the value being the name of the Class it belongs to. Pointcuts are defined as operations with the stereotype <<pointcut>>, and can occur in any Java Class, Interface or aspect; the details of the pointcut are included in the 'behavior' field of the method. Advice is defined as an operation with the stereotype <<advice>>; the pointcut this advice operates on is in the 'behavior' field and acts as part of the method's unique signature. afterAdvice can also have one of the Tagged Values returning or throwing.
PHP	Appropriate type of source file: .php, .php4, or .inc. Nested if condition syntax is enabled.
Python	Appropriate type of source file: .py.
Visual Basic	Appropriate type of source file: .cls Class file.
Visual Basic .NET	Appropriate type of source file: .vb Class file.

Import Resource Script

Enterprise Architect supports the import and export of Microsoft Windows Resource Scripts (as .rc files), which contain the Win32® dialog definitions (those with the stereotype «win32Dialog») for an application's graphical user interface. Dialog resources are imported and exported for a specific language, defaulting to the locale of the current computer system.

Access

Ribbon	Develop > Source Code > Files > Import Resource Script
Keyboard Shortcuts	F7 (synchronize element with code)

Import dialog resources from a .rc file

Option	Action
Resource File	Click on the ... button and locate the .rc file to import the screen elements(s) from.

Resource ID	Either: • Leave the default value 'All' to import all screen elements from the file, or • Click on the drop-down arrow and select the screen ID of a specific dialog to import
Language	Click on the drop-down arrow and select the language version (such as English - United States) of the dialog(s) to import.
Import	Click on this button to import the screens from the resource file. The progress of the import is reported in the field underneath the 'Language' field.

Export a dialog to a .rc file

Option	Action
Screen ID	Defaults from the Win32UI ID Tagged Value of the selected Screen element. (If the dialog does not have this ID, open the 'Win32UI' page of the element's 'Properties' dialog and provide a value for the ID tag.)

Resource File	Click on the ... button and locate the .rc file into which to export the screen element(s). If the element was previously imported, this field defaults to the source file.
Language	Click on the drop-down arrow and select the language version (such as English - United States) of the exported dialog.
Export	Click on this button to export the screens from the resource file. The progress of the export is reported in the field underneath the 'Language' field.

Notes

- New dialogs are exported to an existing .rc file
- In an export to an existing .rc file, no dialogs are ever deleted from the file, even when they are deleted from the model
- In an import, no dialogs are deleted from the model even when omitted from the original .rc file

Import a Directory Structure

You can import from all source files in a complete directory structure, which enables you to import or synchronize multiple files in a directory tree in one pass.

Enterprise Architect creates the necessary Packages and diagrams during the import process.

Access

Ribbon	Develop > Source Code > Files > Import Source Directory
Keyboard Shortcuts	Ctrl+Shift+U

Import a directory structure, using the 'Import Source Directory' dialog

Field	Action
Root directory	Type in or browse for the name of the directory to import.
	Type in or select from the drop-down list

Source Type	the coding language of the files to import in the source directory.
File	Type in or select from the drop-down list, the file extensions to include in the import. Use a ';' to separate values.
Perform a Dry Run	If you want to perform the import as a dry run when you click on the OK button, select this check box. When processing is complete, click on the View Log button to check the predicted outcome of the process.
Recursively Process Subdirectories	If you want to include the contents of subdirectories in the import process, select this check box.
Import components from	If you want to import additional files (as described in the 'Import Component Types' dialog) select this checkbox. You then complete the prompt to specify where the components will come from.
Do not import private members	If you want to exclude private members from the model when importing libraries, select this checkbox.

Prompt for Missing Macro Definitions	During the import, the parser might encounter unrecognized macros. If you select this check box, you will be prompted when such an event occurs and will have the opportunity to define the macro. If you do not select this option, the resultant Package structure could be missing certain items.
Package Structure	Select the appropriate radio button to create a Package for every directory, every namespace or every file; this might be restricted depending on the source type selected.
Create Diagram for each Package	Select this checkbox to create a diagram in each Package created in the import. Click on the Options button to identify which element features to include on the diagrams.
Synchronization	Select the appropriate radio button to synchronize existing classes or overwrite existing classes. If a model Class is found that matches the one in code: • 'Synchronize' updates the model Class to include the details from the one in

	code, which preserves information not represented in code, such as the location of Classes in diagrams • 'Overwrite' deletes the model Class and generates a new one from code; any additional information is not preserved. If the option 'Use timestamps' is selected, then the representation with the latest time stamp (either model or code) will take precedence.
Remove Classes not found in code	Select the appropriate radio button to specify how to handle existing model classes that are not present in the imported code. • 'Never delete' retains all existing Classes in the model. • 'Prompt for action' enables you to review Classes individually • 'Always' delete' removes from the model any Class that is not present in the imported code.
OK	Click on this button to start the import.

Import Binary Module

Enterprise Architect enables you to reverse-engineer certain types of binary module.

Access

Ribbon	Develop > Source Code > Files > Import Binary Module

Use

Currently the permitted types are:

- Java Archive (.jar)
- .NET PE file (.exe, .dll) - Native Windows DLL and EXE files are not supported, only PE files containing .NET assembly data
- Intermediate Language file (.il)

Enterprise Architect creates the necessary Packages and diagrams during the import process; selecting the 'Do not import private members' checkbox excludes private members from libraries from being imported into the model.

When importing .NET files, you can import via reflection or via disassembly, or let the system select the best method - this might result in both types being used.

The reflection-based importer relies on a .NET program, and requires the .NET runtime environment to be installed.

The disassembler-based importer relies on a native Windows program called Ildasm.exe, which is a tool provided with the MS .NET SDK; the SDK can be downloaded from the Microsoft website.

A choice of import methods is available because some files are not compatible with reflection (such as mscorlib.dll) and can only be opened using the disassembler; however, the reflection-based importer is generally much faster.

You can also configure:

- Whether to Synchronize or Overwrite existing Classes when found; if a model Class is found matching the one in the file:
  - Synchronize updates the model Class to include the details from the one in the file, which preserves information not represented in the file, such as the location of Classes in diagrams
  - Overwrite deletes the model Class and generates a new one from the file, which deletes and does not replace the additional information
- Whether to create a diagram for each Package
- What is shown on diagrams created by the import

Classes Not Found During Import

When reverse engineering from your code, there might be times when Classes are deliberately removed from your source code.

The 'Import Source Directory' functionality keeps track of the Classes it expects to synchronize with and, on the 'Import Directory Structure' dialog, provides options for how to handle the Classes that weren't found.

You can select the appropriate option to make Enterprise Architect, at the end of the import, ignore the missing Classes, automatically delete them or prompt you to manage them.

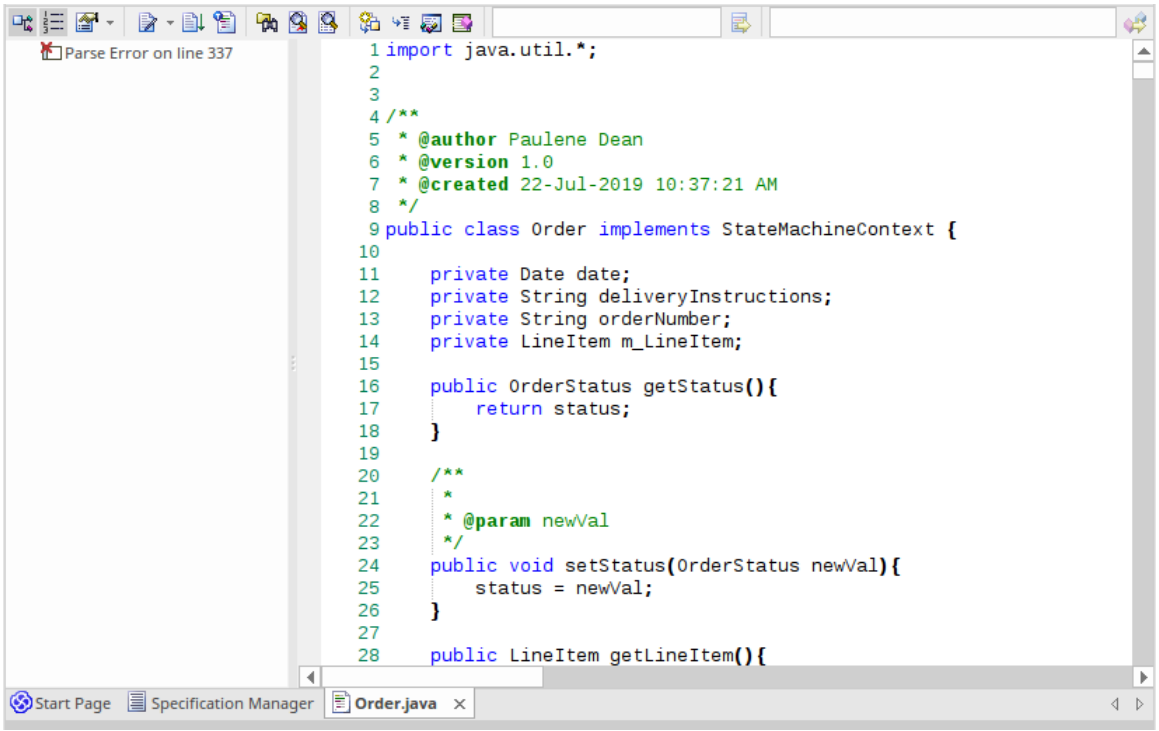
On the 'Import Directory Structure' dialog, if you select the 'Prompt For Action' radio button to manually review missing Classes, a dialog displays on which you specify the handling for each Class that was missing in the imported code.

By default, all Classes are marked for deletion; to keep one or more Classes, select them and click on the Ignore button.

Editing Source Code

Enterprise Architect contains a feature-rich source code editor that helps you to view, edit and maintain your source code directly inside the tool. Once source code has been generated for one or more Classes it can be viewed in this flexible editing environment. Seeing the code in the context of the UML models from which it is derived brings clarity to both the code and the models, and bridges the gap between design and implementation that has historically introduced errors into software systems.

The Source Code Editor is fully-featured, with a structure tree for easy navigation of attributes, properties and methods. Line numbers can be displayed and syntax highlight options can be configured. Many of the features that software engineers are familiar with in their favorite IDE, such as Intelli-sense and code completion are included in the editor. There are many additional features, such as macro recording that makes it easy to manage the source code inside Enterprise Architect. There are also many options for managing the code, available through the code editor context menu, toolbar and function keys.



For most programming languages a single file is created from a UML Class, but in the case of C++ both header and implementation classes are created and the source code editor displays these files in separate tabs.

A number of options change the way the source code editor works; they can be altered using the 'Preferences' dialog available from the Start ribbon:

'Start > Appearance > Preferences > Preferences > Source Code Engineering > Code Editors'

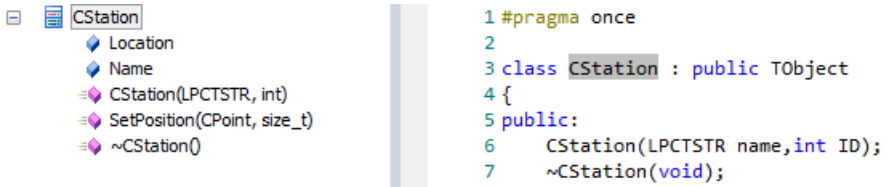
There are variants of the Source Code Editor, with different access methods. The variants are discussed in the *Compare Editors* topic.

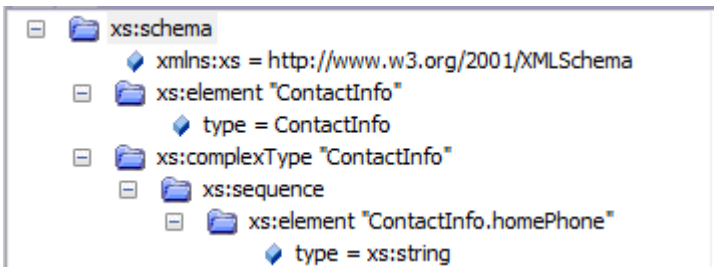
Access

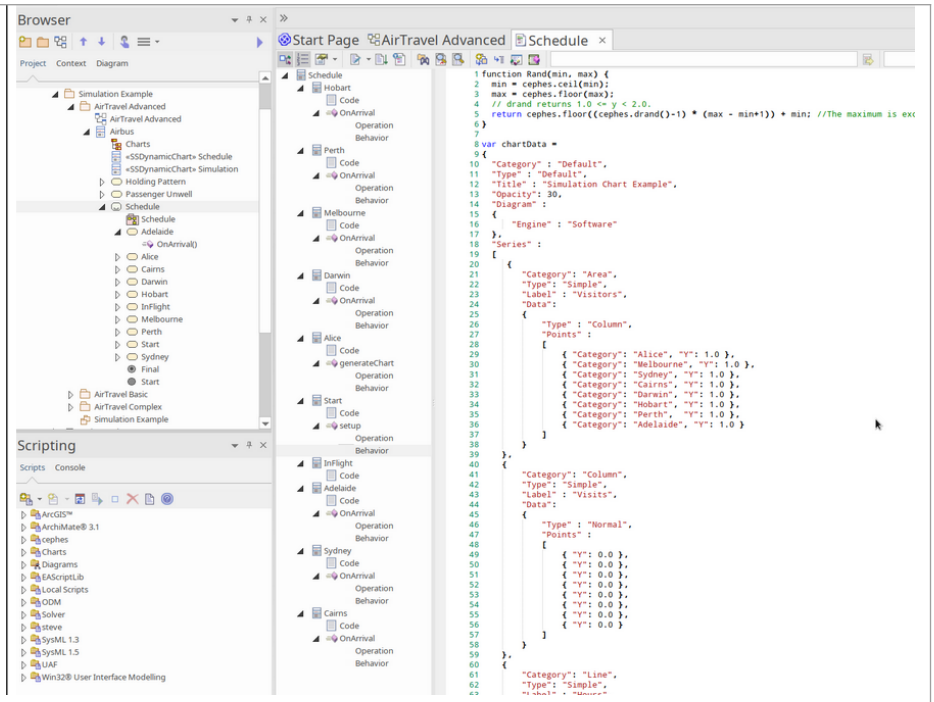
Ribbon	Execute > Source > Edit > Open Source

	File (external file) or Execute > Source > Edit > Edit Element Source (for an existing source file) or Execute > Source > Edit > Edit New Source File or Design > Element > Behavior or Develop > Source Code > Behavior
Keyboard Shortcuts	F12 or Ctrl+E (for existing code for model elements) Ctrl+Alt+O (to locate external files)

Facilities

Facility	Description
Source Code editor	By default the Source Code editor is set to: • Parse all opened files, and show a tree of the results • Show line numbers  If you are editing an XML file, the structure tree mirrors the exact order and

	structure of the document. 
Structure Tree	The file structure tree is available for supported language files, such as C++, C#, Java and XML. The tree can be helpful to navigate content quickly in much the same way a table of contents would for other documents.
Simulation Behaviors	If you are editing the behaviors of the elements in a StateMachine or Activity diagram, the Code Editor allows you to list and edit the behaviors of all elements in the diagram together, using a structure tree.



In this illustration you can see a number of States within a StateMachine, each of which has operations and Behaviors, and all of which are listed together and can be selected without leaving or changing the editor window.

Notes

- For most selected elements you can use the keys F12 or Ctrl+E to view the source code.
- When you select an element to view source code, if the element does not have a generation file (that is, code has not been or cannot be generated, such as for a Use Case), Enterprise Architect checks whether the element has a link to either an operation or an attribute of another element - if such a link exists, and that other element has

source code, the code for that element displays

- You can also locate the directory containing a source file that has been created in or imported to Enterprise Architect, and edit it or its related files using an external editor such as Notepad or Visual Studio; click on the element in the Browser window and press Ctrl+Alt+Y

Languages Supported

The Source Code Editors can display code in a wide range of languages, as listed here. For each language, the editor highlights - in colored text - the standard code syntax.

- Ada (.ada, .ads, .adb)
- ActionScript (.as)
- BPEL Document (.bpel)
- C++ (.h, .hh, .hpp, .c, .cpp, .cxx)
- C# (.cs)
- DDL Structured Query Language (.sql)
- Delphi/Pascal (.pas)
- Diff/Patch Files (.diff, .patch)
- Document Type Definition (.dtd)
- DOS Batch Files (.bat)
- DOS Command Scripts (.cmd)
- HTML (.html)
- Interface Definition Language (.idl, .odl)
- Java (.java)
- JavaScript (.javascript)
- JScript (.js)
- Modified Backus-Naur Form Grammar (.mbnf)
- PHP (.php, .php4, .inc)
- Python (.py)
- Standard Generalized Markup Language (.sgml)

- SystemC (.sc)
- Visual Basic 6 (.bas)
- VB.NET (.vb)
- VBScript (.vbs)
- Verilog (.v)
- VHSIC Hardware Description Language (.vhdl)
- Visual Studio Resource Configuration (.rc)
- XML (eXtensible Markup Language) (.xml)
- XSD (XML Schema Definition)
- XSL (XML Stylesheet Language)

Configure File Associations

If you are a Windows® user, you can configure Enterprise Architect to be the default document handler for your language source files.

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Code Editors : Configure Enterprise Architect File Associations ...

Actions

For each file type that you would prefer to open in Enterprise Architect, click on the checkbox to the left of the file type name. After selecting all of the document types you require, click on the Save button.

After this, clicking on any corresponding file in Windows? Explorer will open it in Enterprise Architect.

Notes

- You can change the default programs, or documents handled by them, directly through the 'Default Programs'

option in Windows ? Control panel.

Compare Editors

Enterprise Architect provides four principal code editor variants, available through a number of access paths. The most direct access options are identified in these descriptions.

The first three code editor variants listed have the same display format, option toolbar, context menu options and internal function keys. They differ in their method of access and display mechanism.

Editor Variants

Variant	Details
Source Code View	F12 Ctrl+E Class context menu | 'View Source Code' Description: Displays the code on a tab of the Diagram View; the tab label shows the file name and extension (such as .java); again, for C++, there are two tabs for the Header and Implementation files. You can display the source code for other Classes on additional tabs, by reselecting the menu option/keys on the next Class.
Source Code	

window (Dockable)	Alt+7 'Execute > Source > Edit > Open Source File' Description: Displays the contents of the source file for a selected Class (except if the language is C++, when the window displays a tab for the Header file and a tab for the Implementation file). If you select a different Class, the window changes to show the code for the new Class (unless the first Class calls the second, in which case the window scrolls down to the second Class's code instead).
Internal Editor, External Source Code	Ctrl+Alt+O 'Execute > Source > Edit > Open Source File' ribbon option Description: Use this option if you intend to edit external code, XML or DDL files (that is, code not imported to or generated in Enterprise Architect). Displays an external browser, then opens the specific selected code file as a tab of the Diagram View (for C++, not two code files); otherwise this is identical to the F12 option.
External Editor,	Ctrl+Alt+Y

Internal or External Source Code	Class context menu | Open Source Directory Description: Displays an external file browser, open to the directory containing the selected Class's source files; you can open the files in Notepad, Visual Studio or other tools you might have on your system.

Code Editor Toolbar

When you are reviewing the code for a part of your model in the Source Code editor, you can access a wide range of display and editing functions from the editor toolbar.

Code Editor Toolbar

Toolbar Options

Structure Tree	Click on this icon to show or hide the element hierarchy panel (the left panel of the Source Code editor).
Line Numbers	Click on this icon to show or hide the line numbers against the lines of code.
Source Code Engineering Properties	Click on the drop-down arrow to display a menu of options to select individual 'Source Code Engineering' pages of the 'Preferences' dialog, from which you can configure display and behavior options for source code engineering: • Language • Syntax Highlighting Options

- Code Editor Options
- Code Engineering Options
- Code Editor Key Bindings

Editor Functions

Click on the drop-down arrow to display a menu providing access to a range of code editing functions:

- Open Corresponding File (Ctrl+Shift+O) - opens the header or implementation file associated with the currently-open file
- Go to Matching Brace (Ctrl+E) - for a selected opening or closing brace, highlights the corresponding closing or opening brace in the pair
- Go to Line (Ctrl+G) - displays a dialog on which you select the number of the line to highlight; click on the OK button to move the cursor to that line
- Cursor History Previous (Ctrl+-) - the Source Code viewer keeps a history of the previous 50 cursor positions, creating a record when the cursor is moved either more than 10 lines away from its previous position, or in a find-and-replace operation; the menu option moves the cursor to the position in the immediately-previous cursor history record

- Cursor History Next (Ctrl+Shift+-) - if you have moved to an earlier cursor position, this option moves the cursor to the position in the immediately-following cursor history record
- Find (Ctrl+F) - displays a dialog in which you define a text string and search options to locate that text string in the code
- Replace (Ctrl+R) - displays a dialog in which you define a text string and search options to locate that text string in the code and replace it with another text string; the dialog has options to locate and replace each occurrence as you decide, or to replace all occurrences immediately
- Highlight Matching Words - (Ctrl+3) Enables or disables the highlighting of matching words during a find operation; by default this option is enabled
- Record Macro - records your next keystrokes to be saved as a macro
- Stop Recording and Save Macro - stops recording the keystrokes and displays the 'Save Macro' dialog on which you specify a name for the macro

- Play Macro - displays the 'Open Macro' dialog from which you select and execute a saved macro, to repeat the saved keystrokes
- Toggle Line Comment (Ctrl+Shift+C) - comments out (//) or re-establishes the code for each full line in which text is highlighted
- Toggle Stream Comment (Ctrl+Shift+X) - inserts a stream comment (/* */) at the cursor position (comments out only the highlighted characters and lines), or re-establishes the commented text as code
- Toggle Whitespace Characters (Ctrl+Shift+W) - shows or hides the spacing characters: --> (tab space) and . (character space)
- Toggle EOL Characters (Ctrl+Shift+L) - shows or hides the end-of-line characters: CR (carriage return) and LF (line feed)
- Toggle Tree Synchronization - selects the tree item automatically as context changes within code editor
- Open Containing Folder - opens the file browser at the folder containing the code file; you can open other files in your default external editor for

comparison and parallel work

Save Source and Resynchronize Class	Click on this icon to save the source code and resynchronize the code and the Class in the model.
Code Templates	Click on this icon to access the Code Templates Editor, to edit or create code templates for code generation.
Find in Project Browser	For a selected line of code, click on this icon to highlight the corresponding structure in the Browser window. If there is more than one possibility the 'Possible Matches' dialog displays, listing the occurrences of the structure from which you can select the required one.
Search in Files	Click on this icon to search for the selected object name in associated files, and display the results of the search in the File Search window. You can refine and refresh the search by specifying criteria on the Find in Files window toolbar.
Search in Model	Click on this icon to search for the selected text throughout the model, and display the results of the search in the

	Find in Project view.
Go to Declaration	Click on this icon to locate the declaration of a symbol in the source code.
Go to Definition	Click on this icon to locate the definition of a symbol in the source code (applicable to languages such as C++ and Delphi, where symbols are declared and defined in separate files).
Autocomplete List	Click on this icon to display the autocompletion list of possible values; double-click on a value to select it.
Parameter Information	When the cursor is between the parentheses of an operation's parameter list, click on this icon to display the operation's signature, highlighting the current parameter.
Find Current Class in Browser Window	Click on this icon to display the name of the currently-selected Class in the code, and highlight that name in the Browser window; if there is more than one possibility the 'Possible Matches' dialog displays, listing the occurrences of the Class from which you can select the

required one.

Find Member	Click on this icon to display the name of the currently-selected attribute or method in the code, and highlight that name in the Browser window; if there is more than one possibility the 'Possible Matches' dialog displays, listing the occurrences of the feature from which you can select the required one.

Notes

- The 'Record Macro' option disables Intelli-sense while the macro is being recorded
- You can assign key strokes to execute the macro, instead of using the toolbar drop-down and 'Open Macro' dialog

Code Editor Context Menu

When working on a file with a code editor, you can perform a number of code search and editing operations to review the contents of the file. These options are available through the editor context menu, and can vary depending on which code editor you are using.

Access

Context Menu	Right-click on the code text string you are working on

Options

Go to Declaration Locate and highlight the declaration of a symbol in the source code.

Go to Definition Locate and highlight the definition of a symbol in the source code (applicable to languages such as C++ and Delphi, where symbols are declared and defined in separate places).

Open in Opens a view that lets you examine or

Grammar Editor	validate the code using the appropriate grammar.
Synchronize Tree to Editor	Finds and displays the current element (method for example) in the structure tree.
Auto Synchronize Tree and Editor	When selected, the structure tree will automatically show the element being worked on in the editor.
XML Schema Validation	Allows an XML schema to validated.
Search for '<string>'	Display a submenu providing options to locate the selected text string in a range of locations. • 'Find in Project Browser' - Highlight the object containing the selected text in the Browser window • 'Search in Open Files' - Search for the selected text string in associated open files and display the results of the search in the Find in Files window; you can refine and refresh the search by specifying criteria on the Find in Files window toolbar

- 'Search in Files' - Search for the selected text string in all associated files (closed or open), and display the results of the search in the Find in Files window; you can refine and refresh the search by specifying criteria on the Find in Files window toolbar (shortcut key: F12)
- 'Search in Model' - Perform an 'Element Name' search in the Model Search facility, and display the results on the Model Search tab
- 'Search in Scripts' - (Available while working in the Script Editor) Open the Find in Files window, set the 'Search Path' field to 'Search in Scripts' and the 'Search Text' field to the selected text, then search all scripts for the text string and display the results of the search; you can refine and refresh the search by specifying criteria on the Find in Files window toolbar
- 'EA User Guide' - Display the description of the code item in the *Enterprise Architect User Guide*
- 'Google' - Display the results of a Google search on the text
- 'MSDN' - Display the results of a search on the text in the Microsoft

Developer Network (MSDN)

- 'Sun Java SE' - Display the results of a search on the text in the Sun Microsystems 'Sun Search' facility
- 'Wikipedia' - Display any entry on the object on the Wikipedia web site
- 'Koders' - Display the results of a search for the text string on Koders.com

Search Intelli-sense

Perform a search on the specified string using the Code Miner service or library specified in the current Analyzer Script. The results are displayed in the 'Code Miner' tab of the Find in Files window.

Shortcut key: Shift+F12

Set Debugger to Line

(If the debugger is executing and has reached a breakpoint.) Move the execution point to the current line. Check that you do not skip over any code or declarations that affect the next section of code being debugged.

Display Variable

(If the debugger is executing.) Open the Locals window and highlight the local variable for the current point in the code.

Show in

Display the full contents of a variable

String Viewer	string in the String Viewer.
Create Use Case for '<string>'	Display the 'Create Use Case For Method' dialog, through which you create a Use Case for the method containing the text string.
Breakpoint	Display a submenu of options for creating a recording marker on the selected line of code. The recording markers you can add include: • Breakpoint • Start Recording Marker • End Recording Marker • Stack Auto Capture Marker • Method Auto Record Marker • Tracepoint
Testpoints	Display options to add a new Testpoint, show the Testpoints Manager (Testpoints window) or edit an existing Testpoint if one or more are already defined at the selected location. (The sub-options depend on the type of code file you are reviewing.)
XML	Allows an XML document to be checked

Validation	for compliance with its own schema references or using a user-specified schema; either a local schema file or a URL.
Open (Close) IME	Open (or close) the Input Method Editor, so that you can enter text in a selected foreign language script, such as Japanese. You set the keyboard language using the Windows Control Panel - Regional and Language Options facility.
Copy Position Hyperlink	Copies the cursor position as a hyperlink that can be pasted into Rich Notes editors, such as a message in the 'Chat' tab of the Chat & Mail window. Simply use the 'Paste' context menu option in the message, and specify the link text. The reader can click on the link to open the source file and move the cursor to the selected cursor position in the file.
Copy Text Hyperlink	Copies the selected text string as a hyperlink that can pasted into Rich Notes editors, such as a message in the 'Chat' tab of the Chat & Mail window. Simply use the 'Paste' context menu option in the message. The reader can click on the link to open

	the source file and move the cursor to the first occurrence of that text string in the file.
Line Numbers	(Script Editor only.) Show or hide the code line numbers on the left hand side of the editor screen.
Undo **Cut** **Copy** **Paste** **Delete** **Select All**	These six options provide simple functions for editing the code.

Notes

- The options in the lower half of the 'Search for <string>' submenu (after 'Search in Scripts') are configurable; you can add new search tools or remove existing ones by editing the searchProviders.xml file in the Sparx Systems > EA > Config folder - this file is in OpenSearch description document format

Create Use Case for Method

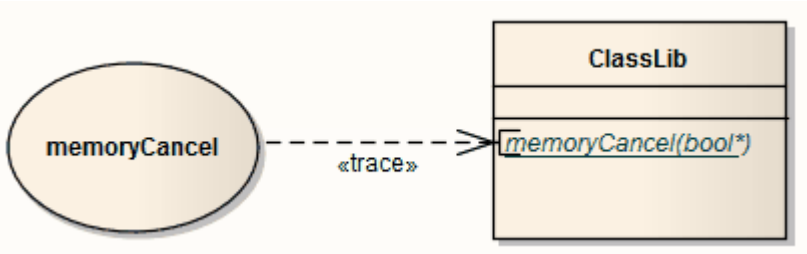
Using the code editor context menu, you can create a Use Case element for a method that you select from the code. You can also:

- Link the Use Case directly to the method
- Add the parent Class to a diagram (if it is not already in the selected diagram) and/or add the Use Case element to the diagram
- Block from display any attributes or methods that are not also the targets of feature links

Create a Use Case for a method, through the code editor

Step	Action
1	(If you want to depict the Use Case and its link to the method in a diagram) click on the diagram name in the Browser window.
2	In the code editor, right-click on either the method name or any part of the method body, and select the 'Create Method for <methodname>' option. The 'Create Use Case for Method' dialog displays.

3	The basic function of this dialog is to create a Use Case for the selected method: • If this is all that is required, click on the OK button; the Use Case element is created in the Browser window, in the same Package as the parent Class for the method, and with the same name as the method • If you intend to make the relationship tangible, continue with the procedure
4	To create a Trace connector linking the Use Case to the method, select the 'Link Use Case to Method' checkbox.
5	To add the method's parent Class to the diagram, if it is not already there, select the 'Add Class to Diagram' checkbox.
6	To add the newly-created Use Case to the diagram, select the 'Add Use Case to Diagram' checkbox; this would now show the Use Case, Class and Trace connector on the diagram.
7	To only show the features (attributes and methods) of the parent Class that are the targets of 'link to feature' relationships, select the 'Display only linked features in Class' checkbox. The Class might contain any number of attributes and methods, but those without a 'link to feature'

	relationship are hidden.
8	Click on the OK button to create and depict the Use Case and relationship; if you selected all options, the diagram now contains linked elements resembling this illustration: 

Code Editor Functions

The common Code Editor provides a variety of functions to assist with the code editing process, including:

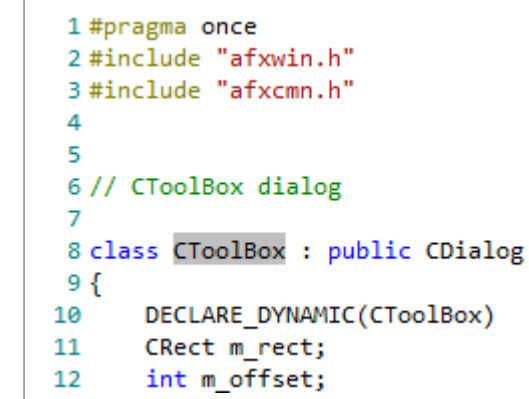
- Syntax Highlighting
- Bookmarks
- Cursor History
- Brace Matching
- Automatic Indentation
- Commenting Selections
- Scope Guides
- Zooming
- Line Selection
- Intelli-sense
- Find and Replace
- Find in Files

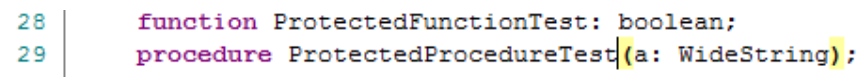
A range of these functions is available through keyboard key combinations and/or context menu options.

You can customize several of the Code Editor features by setting properties in the Code Editor configuration files; for example, by default the line containing the cursor is always highlighted, but you can turn the highlighting off.

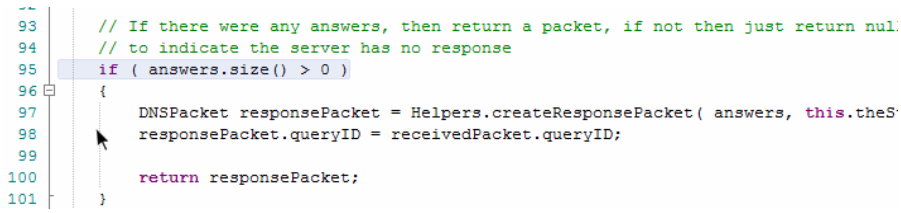
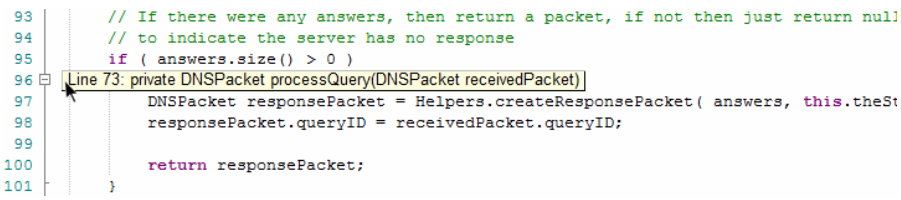
Function Details

Code Editor Functions

Function	Description
Syntax Highlighting	The Code Editor highlights - in colored text - the standard code syntax of all language file formats supported by Enterprise Architect  You can define how the Code Editor implements syntax highlighting for each language, through the 'Code Editors' page of the 'Preferences' dialog.
Bookmarks	Bookmarks denote a line of interest in the document; you can toggle them on and off for a particular line by pressing Ctrl+F2. Additionally, you can press F2 and Shift+F2 to navigate to the next or

	previous bookmark in the document. To clear all bookmarks in the code file, press Ctrl+Shift+F2.
Cursor History	The Code Editor Control keeps a history of the previous 50 cursor positions; an entry in the history list is created when: • The cursor is moved more than 10 lines from its previous position • The cursor is moved in a find/replace operation You can navigate to an earlier point in the cursor history by pressing Ctrl+-, and to a later point by pressing Ctrl+Shift+-.
Brace Matching	When you place the cursor over a brace or bracket, the Code Editor highlights its corresponding partner; you can then navigate to the matching brace by pressing Ctrl+E. 
Automatic Indentation	For each supported language, the Code Editor adjusts the indentation of a new line according to the presence of control statements or scope block tokens in the lines leading up to the cursor position.

| | ```
358 {
359     for(size_t t = 0; t < Stations.size(); t++)
360     {
361         if(Stations[t]->Location == loc)
362             return Stations[t];
363     }
364     return NULL;
365 }
```<br><br>The levels of indent are indicated by pale horizontal lines.<br>You can also manually indent selected lines and blocks of code by pressing the Tab key; to un-indent the selected code, press Shift+Tab. |
| Commenting Selections | For languages that support comments, the Code Editor can comment entire selections of code.
The Code Editor recognizes two types of commenting:
• Line Commenting - entire lines are commented from the start (for example:
// This is a comment)
• Stream Commenting - sections of a line are commented from a specified start point to a specified end point (for example:
/* This is a comment */)
You can toggle comments on the current line or selection by pressing: |
```

	• Ctrl+Shift+C for line comments, or • Ctrl+Shift+X for stream comments
Scope Guides	If the cursor is placed over an indentation marker, the Code Editor performs a 'look back' to find the line that started the scope at that indentation level; if the line is found and is currently on screen, it is highlighted in light blue.  Alternatively if the line is off screen, a calltip is displayed advising of the line number and contents: 
Zooming	You can zoom into and out of the contents of the Code Editor using: • Ctrl+keypad + and • Ctrl+keypad - Zoom can be restored to 100% using Ctrl+keypad /.
Line	If you want to move the cursor to a

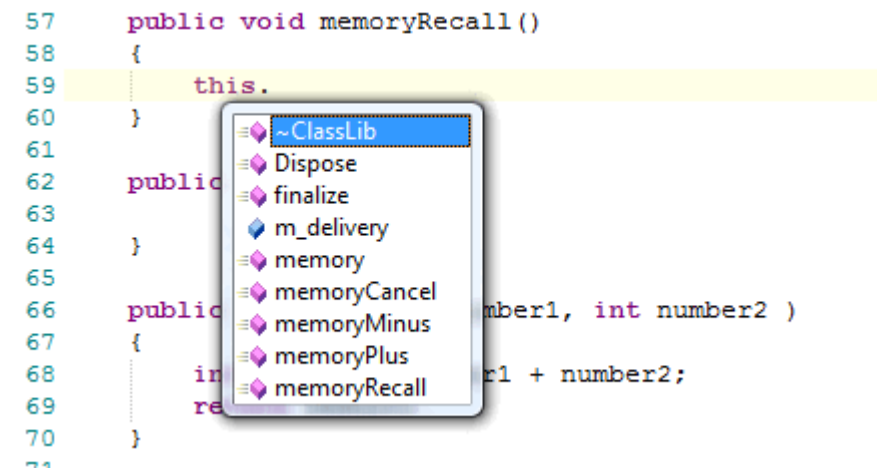
Selection	specific line of code, press Ctrl+G and, in response to the prompt, type in the line number. Press the OK button; the editor displays the specified line of code with the cursor at the left.

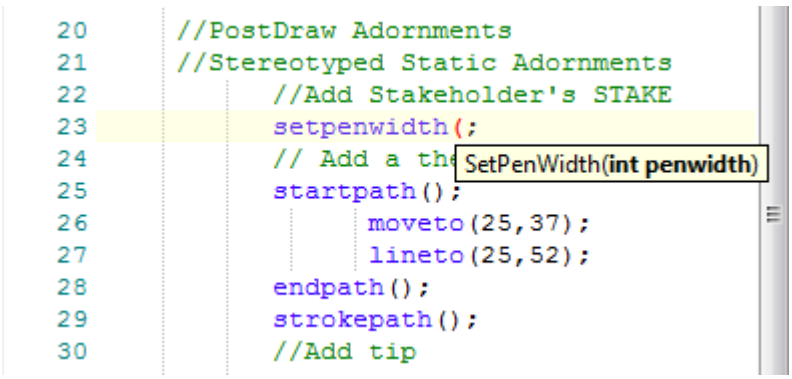
Intelli-sense

Intelli-sense is a feature that provides choices of code items and values as you type. Not all code editors use Intelli-sense; for example, Intelli-sense is disabled while you record a macro in the Source Code Viewer.

Intelli-sense provides you with context-based assistance through autocompletion lists, calltips and mouseover information.

Facilities

Facility	Description
Autocompletion List	An autocompletion list provides a list of possible completions for the current text; the list is automatically invoked when you enter an accessor token (such as a period or pointer accessor) after an object or type that contains members. 

	You can also invoke the autocompletion list manually by pressing Ctrl+Space; the Code Editor then searches for matches for the word leading up to the invocation point. Select an item from the list and press the Enter key or Tab key to insert the item into the code; to dismiss the autocompletion list, press Esc.
Calltips	Calltips display the current method's signature when you type the parameter list token (for example, opening parenthesis); if the method is overloaded, the calltip displays arrows that you can use to navigate through the different method signatures 
Mouseover Information	You can display supporting documentation for code elements (for example, attributes and methods) by hovering the cursor over the element in question.

```
11      dockable = "none";
12
13      Tagged V
14
15      //PreDraw Derived Attribute I
```

string
Dock elements together.
Valid Values: none, standard

Find and Replace

Each of Enterprise Architect's code editors facilitates searching for and replacing terms in the editor, through the 'Find and Replace' dialog.

Access

Keyboard Shortcuts	Highlight the required text string and press: • Ctrl+F for the find controls only, or • Ctrl+R for both find and replace controls In each instance, the 'Find what' field is populated with the text currently selected in the editor. If no text is selected in the editor, the 'Find what' field is populated with the word at the current cursor position. If no word exists at the current cursor position, the last searched-for term is used.

Basic Operations - Commands

Command	Action

Find Next	Locate and highlight the next instance (relative to the current cursor position) of the text specified in the 'Find what' field.
Replace	Replace the current instance of the text specified in the 'Find what' field with the text specified in the 'Replace with' field, and then locate and highlight the next instance (relative to the current cursor position) of the text specified in the 'Find what' field.
Replace All	Automatically replace all instances of the text specified in the 'Find what' field with the text specified in the 'Replace with' field.

Basic Operations - Options

Option	Action
Match Case	Specify that the case of each character in the text string in the 'Find what' field is significant when searching for matches in the code.

Match whole word	Specify that the text string in the 'Find what' field is a complete word and should not be matched with instances of the text that form part of a longer string. For example, searches for ARE should not match those letters in instances of the words AREA or ARENA.
Search up	Perform the search from the current cursor position up to the start of the file, rather than in the default direction of current cursor position to end of file.
Use Regular Expressions	Evaluate specific character sequences in the 'Find what' and 'Replace with' fields as Regular Expressions.

Concepts

Concept	Description
Regular Expressions	A Regular Expression is a formal definition of a Search Pattern, which can be used to match specific characters, words or patterns of characters. For the sake of simplicity, the Code Editor's 'find and replace' mechanism

	supports only a subset of the standard Regular Expression grammar. Text in the 'Find what' and 'Replace with' fields is only interpreted as a Regular Expression if the 'Use Regular Expressions' checkbox is selected in the 'Find and Replace' dialog.
Metasequences	If the 'Use Regular Expressions' checkbox is selected, most characters in the 'Find what' field are treated as literals (that is, they match only themselves). The exceptions are called metasequences; each metasequence recognized in the Code Editor 'Find and Replace' dialog is described in this table: • \< - Indicates that the text is the start of a word; for example: \<cat is matched to *catastrophe* and *cataclysm*, but not *concatenate* • \> - Indicates that the text is the end of a word; for example: hat\> is matched to *that* and *chat*, but not *hate* • (...) - Indicates alternative single characters that can be matched - the characters can be specific (chr) or in an alphabetical or numerical range (a-m); for example: (hc) at is matched to *hat* and *cat* but not *bat*, and (a-m) Class is

	matched to any name in the range *aClass-mClass* • (^...) - Indicates alternative single characters that should be excluded from a match - the characters can be specific (^chr) or in an alphabetical or numerical range (^a-m); for example: (^hc) at is matched to *rat* and *bat*, but *hat* and *cat* are excluded, and (^a-m) Class is matched to any name in the range *nClass* to *zClass*, but *aClass* to *mClass* are excluded • ^ - Matches the start of a line • $ - Matches the end of a line • * - Matches the preceding character (or character set) 0 or more times; for example: ba*t is matched to *bt*, *bat*, *baat*, *baaat* and so on, and b(ea) *t is matched to *bt*, *bet*, *bat*, *beat*, *beet*, *baat* and so on • + - Matches the preceding character (or character set) 1 or more times; for example: ba+t is matched to *bat*, *baat* and *baaat* but not *bt*, and b(ea) +t is matched to *bet*, *bat*, *beat*, *beet* and *baat* but not *bt* If a single character metasequence is preceded by a backslash (\) it is treated as a literal character: c\(at\) matches c(at) as

	the brackets are treated literally. When the 'Use Regular Expressions' checkbox is selected, a metasequence helper menu is available to the right of both of the 'Find what' and 'Replace with' fields; selecting a metasequence from this menu inserts the metasequence into the field, replacing or wrapping the currently selected text as appropriate.
Tagged Regions	When 'find and replacing' with Regular Expressions, up to nine sections of the original term can be substituted into the replacement term. The metasequences '\(' and '\)' denote the start and the end of a tagged region; the section of the matched text that falls within the tagged region can be included in the replacement text with the metasequence '\n' (where *n* is the tagged region number between 1 and 9). For example: Find: \((A-Za-z) +\)*'s things* Replace with *items that belong to \1* Original text: *These are all Michael's things*. Replaced text: *These are all items that belong to Michael.*

Search in Files

File Text Searches are provided by the Find in Files window and from within the Code Editors, to search files for data names and structures. These files can be external code files, code files that you have already opened in Enterprise Architect, internal model scripts or the Help subsystem.

The 'File Search' tab maintains a history of the file paths you have explored, helping you to quickly return to frequently-used folders in your file system. You can similarly select a previously-used search string, if you need to repeat a search several times. When you are searching code files, you can also confine the search to files of specific types, by selecting the file extensions, and to include just the selected folder or all of its sub-folders as well. Another useful facility is being able to select to show the results of the search as either a list of every instance of the string, or a list of files containing the string with the instances grouped under the file in which they are found.

For all searches, you can qualify the search to be case-sensitive and/or to match the search string to complete words.

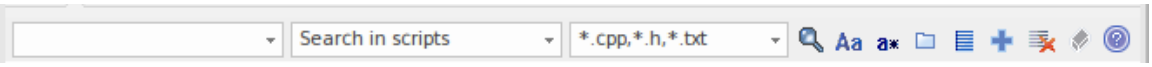
Access

Ribbon	Explore > Search > Files Execute > Source > Find

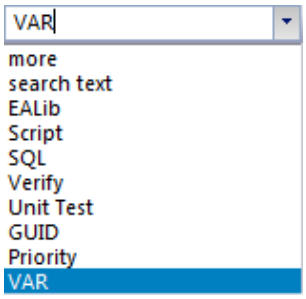
	Execute > Source > Edit > Search in Files
Context Menu	Right-click on selected text | Search for <selected text> | Search in Files
Keyboard Shortcuts	F12, Ctrl+Shift+Alt+F

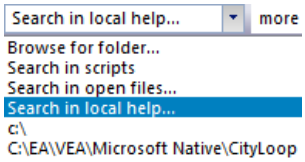
Search Toolbar

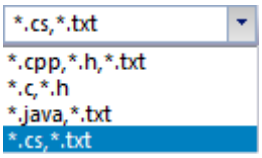
You can use the toolbar options in the Find in Files window to control the search operation. The state of each button persists over time to always reflect your previous search criteria.



Options

Option	Action
	The 'Search Text' field. Type the text string to search for. Any text you type in is automatically saved in the drop-down list, up to a maximum of ten strings; text added after that overwrites the oldest text string in

	the list. You can click on the drop-down arrow and select one of these saved text strings, if you prefer.
	The 'Search Path' field. Specify the folder to search, or the type of search. You can type the folder path to search directly into the text box, or click on the drop-down arrow and select 'Browse for folder' to search using the 'Browse for Folder' dialog. Any paths you enter are automatically saved in the drop-down list, up to a maximum of ten; paths added after that overwrite the oldest path in the list. You can select one of these saved paths if you prefer. Apart from 'Browse for folder', there are three other fixed options in the drop-down list: • 'Search in scripts', which searches the local and user-defined scripts in the Scripting window • 'Search in open files', which confines the search to the files that you have open in Enterprise Architect • 'Search in local help', which searches the local Help files that have been installed from the Sparx Systems web

	site; the results list the Help topics containing the search term, and the line number and line in which the text occurs These options disable the 'Search File Types' list box.
	The 'Search File Types' field. Click on the drop-down arrow and select the file types (file extensions) to search.
	Click on this icon to begin the search. During the course of the search all other buttons in the toolbar are disabled. You can cancel the search at any time by clicking on the Search button again. If you switch any of these toggle buttons, you must run the search again to change the output.
	Click on this icon to toggle the case sensitivity of the search. The tool-tip message identifies the current setting.
	Click on this icon to toggle between searching for any match and searching for only those matches that form an entire word. The tool-tip message identifies the current setting.

	Click on this icon to toggle between limiting the search to a single path and including all subfolders under that path. The tool-tip message identifies the current setting.
	Click on this icon to select the presentation format of the search results; you have two options: • List View - (as shown) each result line consists of the file path and line number, followed by the line text; multiple lines from one file are listed as separate entries • Tree View - ( ) each result line consists of the file path that matches the search criteria, and the number of lines matching the search text within that file; you can expand the entry to show the line number and text of each line
	Click on this icon to add a new search tab. You can create up to four new search tabs. Searches can also run concurrently.
	Click on this icon to clear the results.

	If necessary, click on this icon to remove all the entries in the Search Path, Search Text and Search File Types drop-down lists.

Find File

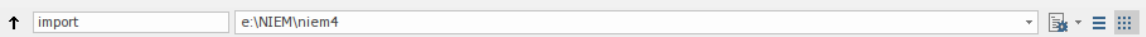
The Find in Files window 'Find File' tab provides a tool that can help you find files quicker. The tab acts as a file system explorer and offers a speedy alternative to the common open file dialog. File searches are quick and simple, allowing you to look up files of interest without losing your current workflow. The display can be switched between report and list view.

Access

Ribbon	Explore > Search > Files > Find File
Keyboard Shortcuts	Ctrl+Shift+Alt+F

Toolbar

The toolbar provides a search filter and folder navigation combo box. The toolbar provides options to remember search locations and alternate between list and report views.



Options

	Click to navigate to the parent folder.
	The filter control allows you to exclude files that do not match the criteria you type. The wildcard symbol * is automatically appended to the text so it is not necessary to add it yourself. To search for all files that contain the term 'jvm' simply type 'jvm'. To find .png images containing the term 'red' you could type *red*.png. Press the Enter key to update the results.
	Enter the path of a directory and press the Enter key to display the files in that location Use the drop down list to select from book-marked locations for the current model. Locations can be managed by using the toolbar menu.
	Allows you to manage the locations displayed in the directory combo. • Remember Path - stores the current value of the 'Directory' field so that, when you return to the Find in Files

	window at a later point the 'Directory' field either defaults to that value (if it is the only 'remembered' value) or offers the value in the drop-down list • Forget Path - clears the current value from memory so that it is not offered as a possible value for the 'Directory' field • Remember Filter - stores the current value in the 'Filter' field so that when you return to the Find in Files window at a later point the 'Filter' field defaults to that value • Forget Filter - removes the 'Filter' field value from memory so that it is not placed in the field next time you access the window
	In this view the list displays the columns 'Name', 'Modified Date', 'Type' and 'Size'. Columns can be sorted in either ascending or descending order. Click the column a third time to remove the sort order.
	The list view removes columns and is convenient when a folder contains many files.

Keyboard Shortcuts

	Sets focus to the filter control.
	Navigates to the parent folder.
	Navigates to the parent folder.
	If a folder is selected, opens the folder, otherwise opens the selected files.

Search Intelli-sense

The Intelli-sense capabilities of Enterprise Architect are built using Sparx Systems' Code Miner tool. The Code Miner provides fast and comprehensive access to the information in an existing code base. The system provides complete access to all aspects of the original source code, either 'on the fly' as one might search in a code editor, or as search results produced by queries written in the Code Miner mFQL language.

Access

On the Find in Files window, click on the 'Code Miner' tab.

Ribbon	Explore > Search > Files
Keyboard Shortcuts	Ctrl+Shift+Alt+F

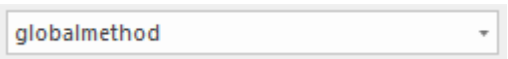
The Code Miner Control

This control presents an interface for performing queries on several code bases at once. The code bases it uses are databases built using Enterprise Architect's Code Miner tool. These databases form a library, which can also be shared when deployed as a service. The queries that can be run are listed and selected using the toolbar, which allows easy

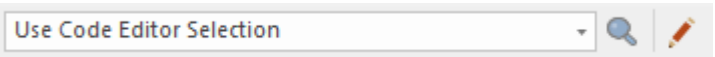
access to the source code for the queries, for editing and composition. Queries do not need to be compiled; they are viewed, edited and saved as one would any source code file. Queries that take a single parameter can utilize any selection in an open code editor. The interface also supports manual parameter entry for queries that take multiple arguments.

The first control on the toolbar lists the namespaces available. Selecting a namespace limits the queries that are displayed to those within that namespace.

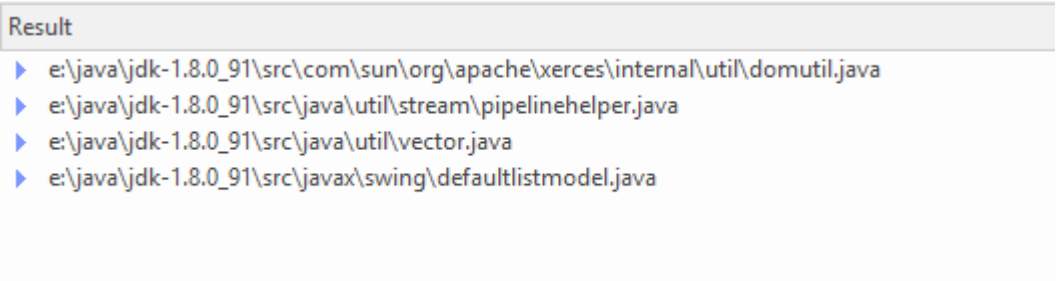
The next control provides a drop-down list of all the queries in the query file for the selected namespace.



The third control is an edit combo box. By default a single query parameter is taken from the selected text in an open code editor, but you can also type the parameter(s) directly into this field. Multiple parameters should be separated by commas. This is followed by the Search button to run the query. Queries can be edited at any time using the Edit button next to the Search button.



The 'Result' panel is a tree control that lists the results of the query grouped by file.



Code Miner Libraries

Code Miner libraries are a collection of databases that can be used by Enterprise Architect Intelli-sense providers to obtain and query for information across several code bases. Each database is created from the root source code directory of a code base, using a specialized grammar appropriate for its language (C++, Java or C#).

The libraries are created, updated, removed or added in the 'Analyzer Script Editor'. A typical scenario for using this feature would be to create a database for a development project and additional databases for frameworks referenced by the project. Your development database can be updated frequently as code changes accrue, while the static frameworks would be updated less often. Libraries can be searched in a similar way to the 'File Search' tool, but Code Miner offers advanced search capabilities due to its mFQL language.

- Multiple domains / frameworks can be searched at once
- A query can be run in a fraction of the time required for a File Search
- Queries can be coded to assist with complex search criteria
- Queries can take multiple parameters
- All files are indexed based on equivalent UML constructs, allowing intelligent searches producing meaningful results in a modeling setting

Code Miner Query Files

Code Miner queries are maintained in a single source code file which should have the .mFQL extension. A basic set of queries is provided with each Enterprise Architect installation; these can be located in the config\codeminer sub directory. This query file should be named by default in any Analyzer Script you edit.

Before editing any queries it is advisable that you copy this file to a working location and name the copy in any Analyzer Script you use. This way you will always have a reference file to go back to.

Queries are best considered as functions that are written in the mFQL language. As such they have unique names, can be qualified by a single namespace and can specify parameters. The file provides the queries listed in the Intelli-sense control's toolbar. Whenever edits to a query file are saved, the queries listed in the search toolbar combo box will be updated accordingly. This image is an example of a simple query written in mFQL.

```
188
189 namespace java
190 {
191 //
192 // Find all references
193 //
194 query::findByName($param1)
195 {
196     distinct(GetByValue( $param1 +))
197 }
198
199 query::findMethodByName($name)
200 {
201     move( 1, "METHOD", intersect( GetByNode("NAME"), GetByValue( $name ) )  )
202 }
203
204 query::findMethodCall($name)
205 {
206     filter( "METHOD_ACCESS", intersect(GetByNode("NAME"), GetByValue( $name )) )
207 }
208
```

Code Editor Key Bindings

Keys

Key	Description
Ctrl+G	Move cursor to a specified line
↓	Move cursor down one line
Shift+↓	Extend selection down one line
Ctrl+↓	Scroll down one line
Alt+Shift+↓	Extend rectangular selection down one line
↑	Move cursor up one line
Shift+↑	Extend selection up one line
Ctrl+↑	Scroll up one line
Alt+Shift+↑	Extend rectangular selection up one line
Ctrl+(	Move cursor up one paragraph

Ctrl+Shift+(	Extend selection up one paragraph
Ctrl+)	Move cursor down one paragraph
Ctrl+Shift+)	Extend selection down one paragraph
←	Move cursor left one character
Shift+←	Extend selection left one character
Ctrl+←	Move cursor left one word
Ctrl+Shift+←	Extend selection left one word
Alt+Shift+←	Extend rectangular selection left one character
→	Move cursor right one character.
Shift+→	Extend selection right one character
Ctrl+→	Move cursor right one word
Ctrl+Shift+→	Extend selection right one word
Alt+Shift+→	Extend rectangular selection right one

	character
Ctrl+/	Move cursor left one word part
Ctrl+Shift+/	Extend selection left one word part
Ctrl+\	Move cursor right one word part
Ctrl+Shift+\	Extend selection right one word part
Home	Move cursor to the start of the current line
Shift+Home	Extend selection to the start of the current line
Ctrl+Home	Move cursor to the start of the document
Ctrl+Shift+Home	Extend selection to the start of the document
Alt+Home	Move cursor to the absolute start of the line
Alt+Shift+Home	Extend rectangular selection to the start of the line
End	Move cursor to the end of the current line

Shift+End	Extend selection to the end of the current line
Ctrl+End	Move cursor to the end of the document
Ctrl+Shift+End	Extend selection to the end of the document
Alt+End	Move cursor to the absolute end of the line
Alt+Shift+End	Extend rectangular selection to the end of the line
Page Up	Move cursor up a page
Shift+Page Up	Extend selection up a page
Alt+Shift+Page Up	Extend rectangular selection up a page
Page Down	Move cursor down a page
Shift+Page Down	Extend selection down a page
Alt+Shift+Pa	Extend rectangular selection down a page

ge Down	
Delete	Delete character to the right of the cursor
Shift+Delete	Cut selection
Ctrl+Delete	Delete word to the right of the cursor
Ctrl+Shift+Delete	Delete until the end of the line
Insert	Toggle overtype
Shift+Insert	Paste
Ctrl+Insert	Copy selection
Backspace	Delete character to the left of the cursor
Shift+Backspace	Delete character to the left of the cursor
Ctrl+Backspace	Delete word to the left of the cursor
Ctrl+Shift+Backspace	Delete from the start of the line to the cursor

Alt+Backspace	Undo delete
Tab	Indent cursor one tab
Ctrl+Shift+I	Indent cursor one tab
Shift+Tab	Unindent cursor one tab
Ctrl+keypad(+)	Zoom in
Ctrl+keypad(-)	Zoom out
Ctrl+keypad(/)	Restore Zoom
Ctrl+Z	Undo
Ctrl+Y	Redo
Ctrl+X	Cut selection
Ctrl+C	Copy selection
Ctrl+V	Paste

Ctrl+L	Cut line
Ctrl+T	Transpose line
Ctrl+Shift+T	Copy line
Ctrl+A	Select entire document
Ctrl+D	Duplicate selection
Ctrl+U	Convert selection to lowercase
Ctrl+Shift+U	Convert selection to uppercase
Ctrl+E	Move cursor to matching brace
Ctrl+Shift+E	Extend selection to matching brace
Ctrl+Shift+C	Toggle line comment on selection
Ctrl+Shift+X	Toggle stream comment on selection.
Ctrl+F2	Toggle bookmark
F2	Go to next bookmark
Shift+F2	Go to previous bookmark

Ctrl+Shift+F2	Clear all bookmarks in current file
Ctrl+Shift+W	Toggle whitespace characters
Ctrl+Shift+L	Toggle EOL characters
Ctrl+Space	Invoke autocomplete.
Ctrl+-	Go backwards in cursor history
Ctrl+Shift+-	Go forwards in cursor history
F12	Start/Cancel search for keyword in file(s).
Ctrl+F	Find text
Ctrl+R	Replace text

Notes

- In addition to these keys, you can assign (Ctrl+Alt+<n>) key combinations to macros that you define within the Source Code Editor

Application Patterns (Model + Code)

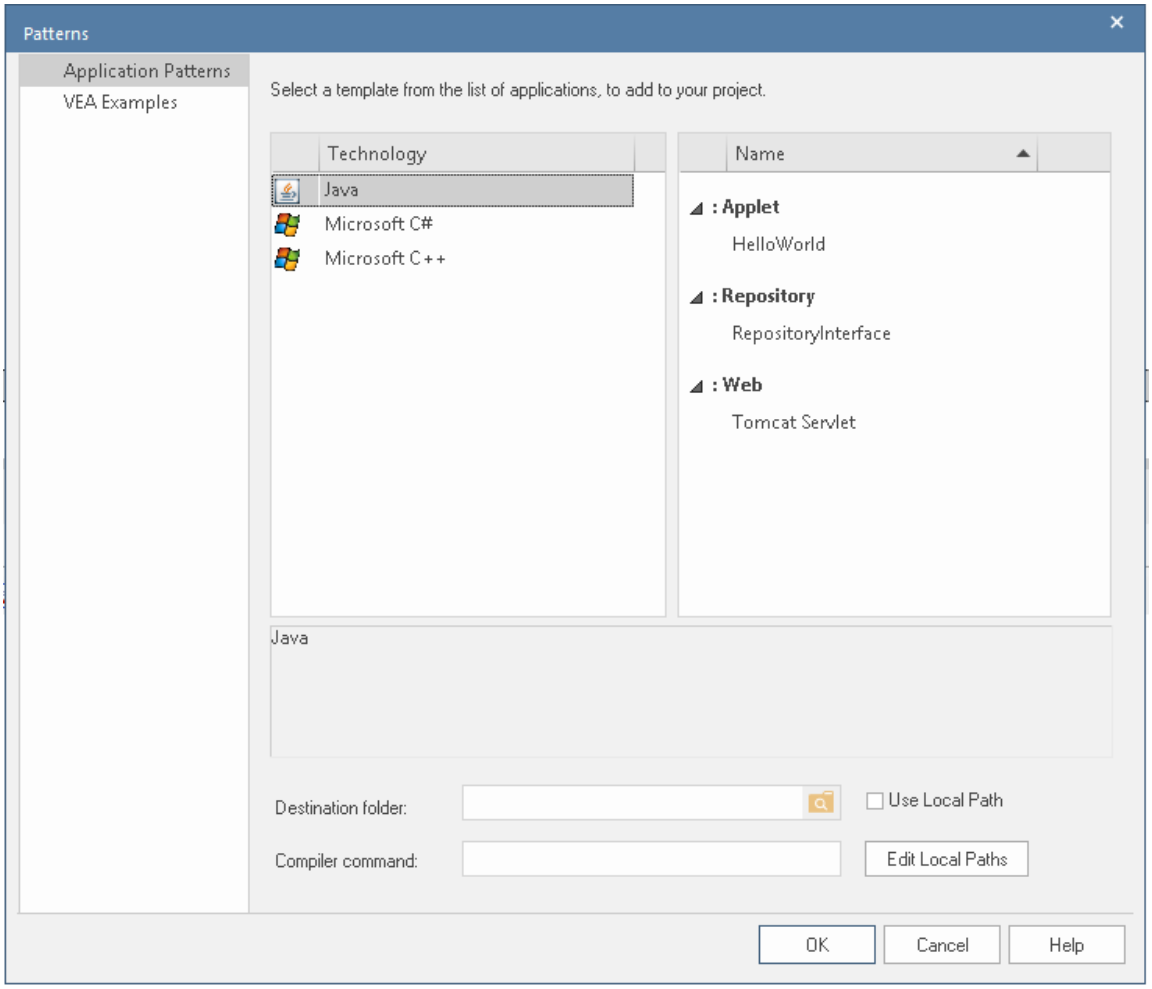
To get you going with a code based project as fast as possible, Enterprise Architect helps you to generate starter projects including model information, code and build scripts for one of several basic application types. Patterns include:

- MFC Windows applications
- Java programs
- ASP.NET web services

Access

Ribbon	Develop > Source Code > Create From Pattern > Application Patterns

Generate Models



Option	Action
Technology	Select the appropriate technology.
Name	Displays the Application Patterns available for the selected technology; select the required Pattern to import.
<description>	Displays a description of the selected Pattern.
Destination folder	Browse for and select the directory in which to load the source code for the

	application.
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 application Patterns specify their compiler using a local path. The first time you use any Pattern you must click on this button to ensure the local path points to the correct location. The 'Local Paths' dialog displays.

Notes

- If required, you can publish custom application Patterns by adding files to the *AppPatterns* directory where 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 identified 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

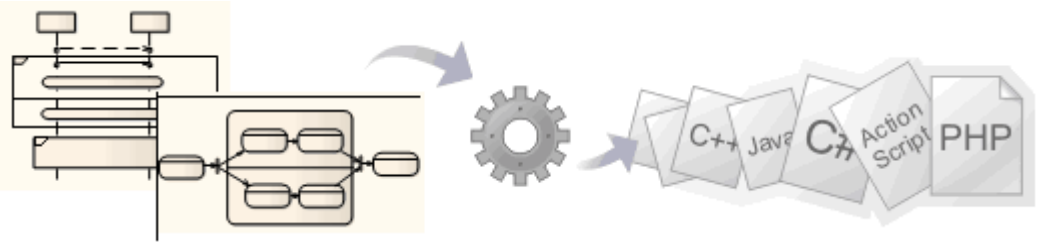
MDG Integration and Code Engineering

MDG Integration for Eclipse and MDG Integration for Visual Studio are products that help you to create and maintain your UML models directly inside these two popular Integrated Development Environments, using the Enterprise Architect Browser window. Models can be generated to source code using the rich and flexible template engine that gives the engineer complete control over how the code is generated. Existing source code can also be reverse engineered and synchronized with the UML models. With the integration installed the IDE will become a feature-rich modeling platform, saving time and effort and reducing the risk of error by linking Requirement Management, Architecture and Design to Source Code Engineering.

Rich and expressive documentation can be generated automatically into a wide range of formats including DOCX, PDF and HTML. The documentation can include diagrams of requirements, design and architecture as well as source code descriptions, putting the source code into context.

You can purchase MDG Integration for Eclipse™ and MDG Integration for Visual Studio™ or download Trial Editions, from the Sparx Systems web site.

Behavioral Model Code Generation



Enterprise Architect's multi-featured system engineering capability can be used to generate code for software, system and hardware description languages directly from behavioral models, such as StateMachine, Sequence (Interaction) and Activity diagrams. The supported languages include C(OO), C++, C#, Java, VB.Net, VHDL, Verilog and SystemC.

Software code can be generated from StateMachine, Sequence and Activity diagrams, and hardware description languages from StateMachine diagrams (using the Legacy StateMachine templates).

Access

Ribbon	Develop > Source Code > Generate

Behavioral Model Specifics

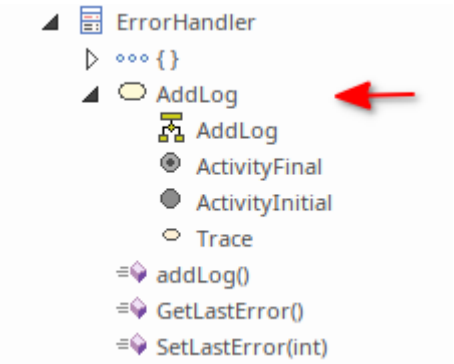
Behavioral model code generation is supported for the three key types of behavioral model; however, each behavioral model-type has its own characteristics based on the

element-type involved. These topics provide guidance and references for the core element-types used.

Type	Description
Activity	An Overview of the key Action-types and details on using these in code generation.
Interaction	Details covering using Messages and Fragments for code generation of Interaction (Sequence) diagrams.
StateMachines	Details covering the options for defining the code to be generated using States, including behaviors - Entry/Exit/Do, and Transitions in a StateMachine.

Structure

Behavioral model code generation primarily requires that all behavioral constructs are be contained within a Class (as a child of that Class).



If any behavioral constructs refer to external elements outside the current Package, you must add an Import connector from the current Package to the Package containing the external elements. For more detail see the *Import* connector-type in the *Package Diagram* Help topic.

Generate code from behavioral diagrams using the EAExample project

Step	Action
1	Open the EAExample.eap file by selecting the 'Start > Help > Help > Open the Example Model' ribbon option.
2	From the Browser window, select any of these Packages: **Software Language Examples:** • Example Model > Software Engineering > Java Model With Behaviors Generate the Account and Order classes • Example Model > Systems Engineering > Implementation Model > Software > C# Generate the DataProcessor Class • Example Model > Systems Engineering > SysML Example > Implementation Model > Software >

	C++ Generate the IO Class • Example Model > Systems Engineering > SysML Example > Implementation Model > Software > Java Generate the IO Class • Example Model > Systems Engineering > SysML Example > Implementation Model > Software > VBNet Generate the IO Class **Hardware Language Examples:** • Example Model > Systems Engineering > SysML Example: Portable Audio Player > Implementation Model > Hardware > SystemC Generate the PlayBack Class • Example Model > Systems Engineering > SysML Example: Portable Audio Player > Implementation Model > Hardware > VHDL Generate the PlayBack Class • Example Model > Systems Engineering > SysML Example: Portable Audio Player > Implementation Model > Hardware > Verilog Generate the PlayBack Class
3	When completed: • Select the Class that was used for the generation • Press Ctrl+E to open the generated source code.

	You should see methods generated in the code.

Notes

- Software code generation from behavioral models is available in the Unified and Ultimate Editions of Enterprise Architect
- Hardware code generation from StateMachine models is available in the Unified and Ultimate Editions of Enterprise Architect
- For C(OO), on the 'C Specifications' page of the 'Manage Model Options' dialog, set the 'Object Oriented Support' option to True.
  See the *C Options - Model* Help topic.
- Code synchronization is not supported for behavioral code.

Code Generation - Activity Diagrams

Code generation from Activity diagrams in a Class requires a validation phase, during which Enterprise Architect uses the system engineering graph optimizer to analyze the diagram and render it into various constructs from which code can be generated. Enterprise Architect also transforms the constructs into one of the various action types (if appropriate), similar to the Interaction diagram constructs.

Actions

Action	Description
Call Actions (Invocation Actions)	Used to invoke operations or behaviors in an Activity diagram; the two main variants of Call Actions supported in behavioral code generation are: • CallOperation Action - used to invoke operations, which can be within the same Class or in other Classes within the same Package; if referencing operations from other Classes within the same Package, you must have a target to which the request is passed • CallBehavior Action - used to invoke another Activity in an activity flow; the referenced Activity is expected to be

	within the same Class **Arguments** Call Actions can specify argument values corresponding to the parameters in the associated behavior or behavioral feature. You can add the arguments manually or create them automatically using the Synchronize button of the 'Arguments' dialog.
CreateObject Action	Used to denote an object creation in the activity flow; you can set the result Pin of the CreateObjectAction as the object to be created, using the Properties window for the Action element. The Classifier of the CreateObjectAction signifies the Classifier for which an instance is to be created.
DestroyObjectAction	Used to denote an object deletion in the activity flow; you can set the target Pin of the DestroyObjectAction as the object to be destroyed, using the Properties window for the Action element.
Loops	Enterprise Architect's system engineering graph optimizer is also capable of analyzing and identifying loops; an identified loop is internally rendered as

	an Action Loop, which is translated by the EASL code generation macros to generate the required code. You can have a single loop, nested loops, and multiple levels of nested loops.
Conditional Statements	To model a conditional statement, you use Decision/Merge nodes. Alternatively, you can imply Decisions/Merges internally; the graph optimizer expects an associated Merge node for each Decision node, to facilitate efficient tracking of various branches and analysis of the code constructs within them.

Notes

- To be able to generate code from behavioral models, all behavioral constructs should be contained within a Class

Code Generation - Interaction Diagrams

During code generation from Interaction (Sequence) diagrams in a Class, Enterprise Architect applies its system engineering graph optimizer to transform the Class constructs into programmatic paradigms. Messages and Fragments are identified as two of the several action types based on their functionality, and Enterprise Architect uses the code generation templates to render their behavior accordingly.

Actions

Action	Description
Action Call	A Message that invokes an operation.
Action Create	A Message with Lifecycle = New.
Action Destroy	A Message with Lifecycle = Delete.
Action Loop	A Combined Fragment with Type = Alt.
Action If	A Combined Fragment with Type = loop.

Assign To	A Call Message with a valid target attribute set using the 'Assign To' field is rendered in the code as the target attribute of a Call Action.

Notes

- To be able to generate code from behavioral models, all behavioral constructs should be contained within a Class
- For an Interaction (Sequence) diagram, the behavioral code generation engine expects the Sequence diagram and all its associated messages and interaction fragments to be encapsulated within an Interaction element

Code Generation - StateMachines

A StateMachine illustrates how an object (represented by a Class) can change state, each change of state being a transition initiated by a trigger arising from an event, often under conditions or constraints defined as guards. As you model how the object changes state, you can generate and build (compile) code from it in the appropriate software language and execute the code, visualizing the execution via the Model Simulator.

It is also possible, in Enterprise Architect, to combine the StateMachines of separate but related objects to see how they interact (via Broadcast Events), and to quickly create and generate code from variants of the model. For example, you might model the behavior of:

- The rear off-side wheel of a vehicle in rear-wheel drive and front-wheel drive modes (one StateMachine)
- The steering wheel and all four drive wheels of a vehicle in 4-wheel drive mode (five StateMachines)
- The wheels of an off-road vehicle and of a sports car (two Artifacts, instances of a combination of StateMachines)

Of critical importance in generating and testing code for all of these options is the Executable StateMachine Artifact element. This acts as the container and code generation unit for your StateMachine models.

You do not use this method to generate code for Hardware Definition Languages, but you can also generate both HDL code and software code from StateMachines using the

generic Code Generation facilities in Enterprise Architect (see the *Generate Source Code* procedures).

Prerequisites

- Select 'Settings > Model > Options > Source Code Engineering' and, for the appropriate software coding language (Java, C, C# or ANSI C++), set the 'Use the new Statemachine Template' option to 'True'
- If working in C++, select 'Settings > Model > Options > Source Code Engineering > C++' and set the 'C++ Version' option to 'ANSI'

This code generation method does not apply to the Legacy StateMachine code generation templates developed prior to Enterprise Architect Release 11.0, nor to generating Hardware Definition Language code.

Access

Drag an Executable StateMachine Artifact from the 'Simulation' page of the Diagram Toolbox, onto your diagram. The 'Simulation' page of the Diagram Toolbox can be accessed using any of the methods outlined in this table.

Ribbon	Design > Diagram > Toolbox > Simulation
Keyboard Shortcuts	Ctrl+Shift+3 > Simulation

Other	You can display or hide the Diagram Toolbox by clicking on the » or « icons at the left-hand end of the Caption Bar at the top of the Diagram View.

Prepare your StateMachine diagram(s)

Step	Action
1	For each StateMachine you want to model, create a Class diagram.
2	From the 'Class' page of the Diagram Toolbox, drag the 'Class' icon onto your diagram and give the element an appropriate name.
3	Right-click on the Class element and select the 'New Child Diagram | StateMachine' context menu option. Give the StateMachine diagram an appropriate name.
4	Create the StateMachine model to reflect the appropriate transitions between States.

Set up the Executable StateMachine Artifact

Step	Action
1	Create a new Class diagram to contain the modeled StateMachine(s) from which you intend to generate code.
2	From the 'Simulation' page of the Diagram Toolbox, drag the 'Executable StateMachine' icon onto the diagram to create the Artifact element. Name the element and drag its borders out to enlarge it.
3	From the Browser window, drag the (first) Class element containing a StateMachine diagram onto the Artifact element on the diagram. The 'Paste <element name>' dialog displays. In the 'Drop as' field, click on the drop-down arrow and select the value 'Property'. (If the dialog does not display, press Ctrl as you drag the Class element from the Browser window.)
4	Click on the OK button. The Class element is pasted inside the Artifact as a Part.
5	Repeat steps 3 and 4 for any other Classes with

	StateMachines that you want to combine and generate code for. These might be: • Repeat 'drops' of the same Class and StateMachine, modeling parallel objects • Different Classes and StateMachines, modeling separate interacting objects
6	Right-click on the Artifact element and select the 'Properties > Properties' option, expand the 'Advanced' category and, in the 'Language' field, click on the drop-down arrow and set the code language to the same language as is defined for the Class elements. You can now drag this Executable StateMachine Artifact element from the Browser window onto the diagram any number of times, and modify the Parts to model variations of the system or process, or the same system or process with different programming languages.

Generate Code From Artifact

Step	Action
1	Click on the Executable StateMachine Artifact element and select the 'Simulate > Executable States

	> Statemachine > Generate' ribbon option. The 'Executable Statemachine Code Generation' dialog displays.
2	In the 'Project output directory' field, type or browse for the directory path under which to create the output files. During code generation, all existing files in this directory are deleted.
3	Select the Target System. If you are running on Windows select the 'Local' option. If you are working on Linux choose the 'Remote' option. The choice affects the scripts generated to support the Simulation.
4	In the 'Location of <compiler> installation directory' field, type or browse for the path of the compiler installation directory, to be automatically mapped to the local path (displayed to the left of the field). For each programming language, the paths might resemble these examples: • Java JAVA_HOME C:\Program Files (x86)\Java\jdk1.7.0_17 • C/C++ VC_HOME C:\Program Files (x86)\Microsoft Visual Studio 9.0 • C#

	CS_HOME C:\Windows\Microsoft.NET\Framework\V3.5
5	Click on the Generate button. The code files are created appropriate to the programming language. The System Output window displays with an 'Executable StateMachine Output' tab, showing the progress and status of the generation. During code generation, an automatic validation function is executed to check for diagram or model errors against the UML constraints. Any errors are identified by error messages on the 'Executable StateMachine Output' tab. Double-click on an error message to display the modeling structure in which the error occurs, and correct the mistake before re-generating the code.
6	When the code generates without error, click on the Artifact element and select the 'Simulate > Executable States > Statemachine > Build' ribbon option to compile the code. The System Output window displays with a 'Build' tab, showing the progress and status of the compilation. Notice that the compilation includes configuration of the simulation operation.

Code Generation Macros

You can also use two macros in the code generation for StateMachines.

Macro Name	Description
SEND_EVENT	Send an event to a receiver (the Part). For example: %SEND_EVENT("event1", "Part1")%
BROADCAST_EVENT	Broadcast an event to all receivers. For example: %BROADCAST_EVENT("event2")%

Execute/Simulate Code From Artifact

Step	Action
1	Select the ribbon option 'Simulate > Dynamic Simulation > Simulator > Apply Workspace' to display the Simulation window and the Simulation Events window together Dock the two windows in a convenient area of the screen.

2	On the diagram or Browser window, click on the Artifact element and select the 'Simulate > Executable States > Statemachine > Run' ribbon option. The first StateMachine diagram in the series displays with the simulation of the process already started. In the Simulation window, the processing steps are indicated in this format: [03516677] Part1[Class1].Initial_367_TO_State4_142 Effect [03516683] Part1[Class1].StateMachine_State4 ENTRY [03516684] Part1[Class1].StateMachine_State4 DO [03518375] Blocked
3	Click on the appropriate Simulation window toolbar buttons to step through the simulation as you prefer. When the simulation finishes at the Exit or Terminate element, click on the Stop button in the Simulation window toolbar.
4	Where the trace shows Blocked, the simulation has reached a point where a Trigger event has to occur before processing can continue. On the Simulation Events window, in the 'Waiting Triggers' column, double-click on the appropriate Trigger. When the Trigger is fired, the simulation continues

	to the next pause point, Trigger or exit.

Notes

- If you are making small changes to an existing StateMachine model, you can combine the code generation, build and run operations by selecting the 'Simulate > Executable States > Statemachine > Generate, build and run' ribbon option
- You can also generate code in JavaScript

Legacy StateMachine Templates

Code generation operates using a set of generation templates. From Release 11.0 of Enterprise Architect, a different set of templates are available as the default for software code generation from a StateMachine diagram into Java, C, ANSI C++ or C# code. You can still use the original templates, as described here, for models developed in earlier releases of Enterprise Architect, if you do not want to upgrade them for the new template facilities.

Switch Between Legacy and Release 11 templates

Access

Display the 'Manage Model Options' dialog, then show the 'Language Specifications' page for your chosen language, using one of the methods outlined in this table. If necessary, expand the 'StateMachine Engineering (for current model)' grouping and set the 'Use the new StateMachine Template' option to True (to use the later templates) or False (to use the Legacy templates).

Ribbon	Settings > Model > Options > Source Code Engineering > [language name]

Legacy Template Transformations

A StateMachine in a Class internally generates a number of constructs in software languages to provide effective execution of the States' behaviors (do, entry and exit) and also to code the appropriate transition's effect when necessary.

Model Objects	Code Objects
Enumerations	• StateType - consists of an enumeration for each of the States contained within the StateMachine • TransitionType – consists of an enumeration for each transition that has a valid effect associated with it; for example, ProcessOrder_Delivered_to_ProcessOrder_Closed • CommandType – consists of an enumeration for each of the behavior types that a State can contain (Do, Entry, Exit)
Attributes	• currState:StateType - a variable to hold the current State's information • nextState:StateType - a variable to hold the next State's information, set by each

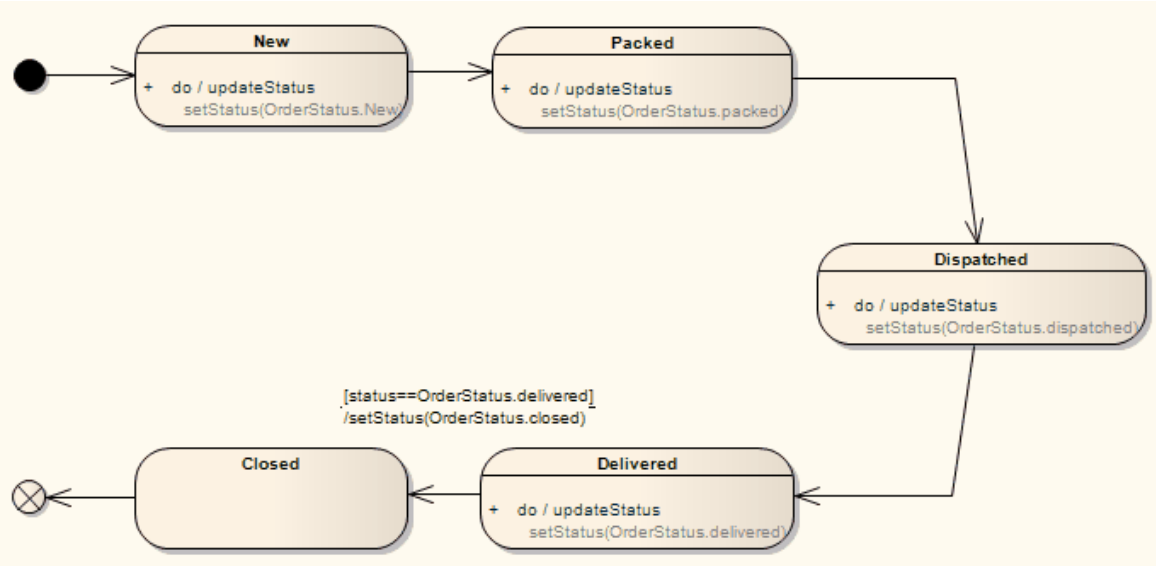
	State's transitions accordingly • currTransition:TransitionType - a variable to hold the current transition information; this is set if the transition has a valid effect associated with it • transcend:Boolean - a flag used to advise if a transition is involved in transcending between different StateMachines (or Submachine states) • xx_history:StateType - a history variable for each StateMachine/Submachine State, to hold information about the last State from which the transition took place
Operations	• StatesProc - a States procedure, containing a map between a State's enumeration and its operation; it de-references the current State's information to invoke the respective State's function • TransitionsProc - a Transitions procedure, containing a map between the Transition's enumeration and its effect; it invokes the Transition's effect • <<State>> - an operation for each of the States contained within the StateMachine; this renders a State's behaviors based on the input

	CommandType, and also executes its transitions • initializeStateMachine - a function that initializes all the framework-related attributes • runStateMachine - a function that iterates through each State, and executes their behaviors and transitions accordingly

Notes

- To be able to generate code from behavioral models, all behavioral constructs should be contained within a Class

Java Code Generated From Legacy StateMachine Template




```
private enum StateType: int
{
    ProcessOrder_Delivered,
    ProcessOrder_Packed,
    ProcessOrder_Closed,
    ProcessOrder_Dispatched,
    ProcessOrder_New,
    ST_NOSTATE
}
private enum TransitionType: int
{
    ProcessOrder_Delivered_to_ProcessOrder_Closed,
    TT_NOTRANSITION
```

```
    }
    private enum CommandType
    {
        Do,
        Entry,
        Exit
    }
    private StateType currState;
    private StateType nextState;
    private TransitionType currTransition;
    private boolean transcend;
    private StateType ProcessOrder_history;
    private void processOrder_Delivered(CommandType
command)
    {
        switch(command)
        {
            case Do:
            {
                // Do Behaviors..
                setStatus(Delivered);
                // State's Transitions
                if((status==Delivered))
                {
                    nextState = StateType.ProcessOrder_Closed;
```

```
                    currTransition =
TransitionType.ProcessOrder_Delivered_to_ProcessOrder_
Closed;
                }
                break;
            }
            default:
            {
                break;
            }
        }
    }
    private void processOrder_Packed(CommandType
command)
    {
        switch(command)
        {
            case Do:
            {
                // Do Behaviors..
                setStatus(Packed);
                // State's Transitions
                nextState =
StateType.ProcessOrder_Dispatched;
                break;
```

```
            }
            default:
            {
                break;
            }
        }
    }
    private void processOrder_Closed(CommandType
command)
    {
        switch(command)
        {
            case Do:
            {
                // Do Behaviors..
                // State's Transitions
                break;
            }
            default:
            {
                break;
            }
        }
    }
    private void processOrder_Dispatched(CommandType
```

```
command)
    {
        switch(command)
        {
            case Do:
            {
                // Do Behaviors..
                setStatus(Dispatched);
                // State's Transitions
                nextState = StateType.ProcessOrder_Delivered;
                break;
            }
            default:
            {
                break;
            }
        }
    }
    private void processOrder_New(CommandType
command)
    {
        switch(command)
        {
            case Do:
            {
```

```
                // Do Behaviors..
                setStatus(new);
                // State's Transitions
                nextState = StateType.ProcessOrder_Packed;
                break;
            }
            default:
            {
                break;
            }
        }
    }
    private void StatesProc(StateType currState,
CommandType command)
    {
        switch(currState)
        {
            case ProcessOrder_Delivered:
            {
                processOrder_Delivered(command);
                break;
            }
            case ProcessOrder_Packed:
            {
                processOrder_Packed(command);
```

```
            break;
        }
        case ProcessOrder_Closed:
        {
            processOrder_Closed(command);
            break;
        }
        case ProcessOrder_Dispatched:
        {
            processOrder_Dispatched(command);
            break;
        }
        case ProcessOrder_New:
        {
            processOrder_New(command);
            break;
        }
        default:
        break;
    }
}
private void TransitionsProc(TransitionType transition)
{
    switch(transition)
    {
```

```
            case
ProcessOrder_Delivered_to_ProcessOrder_Closed:
            {
                setStatus(closed);
                break;
            }
            default:
            break;
        }
    }
    private void initalizeStateMachine()
    {
        currState = StateType.ProcessOrder_New;
        nextState = StateType.ST_NOSTATE;
        currTransition =
TransitionType.TT_NOTRANSITION;
    }
    private void runStateMachine()
    {
        while (true)
        {
            if (currState == StateType.ST_NOSTATE)
            {
                break;
            }
```

```
            currTransition =
TransitionType.TT_NOTRANSITION;
            StatesProc(currState, CommandType.Do);
            // then check if there is any valid transition
assigned after the do behavior
            if (nextState == StateType.ST_NOSTATE)
            {
                break;
            }
            if (currTransition !=
TransitionType.TT_NOTRANSITION)
            {
                TransitionsProc(currTransition);
            }
            if (currState != nextState)
            {
                StatesProc(currState, CommandType.Exit);
                StatesProc(nextState, CommandType.Entry);
                currState = nextState;
            }
        }
    }
```

StateMachine Modeling For HDLs

To efficiently generate Hardware Description Language (HDL) code from StateMachine models, apply the design practices described in this topic. Hardware Description Languages include VHDL, Verilog and SystemC.

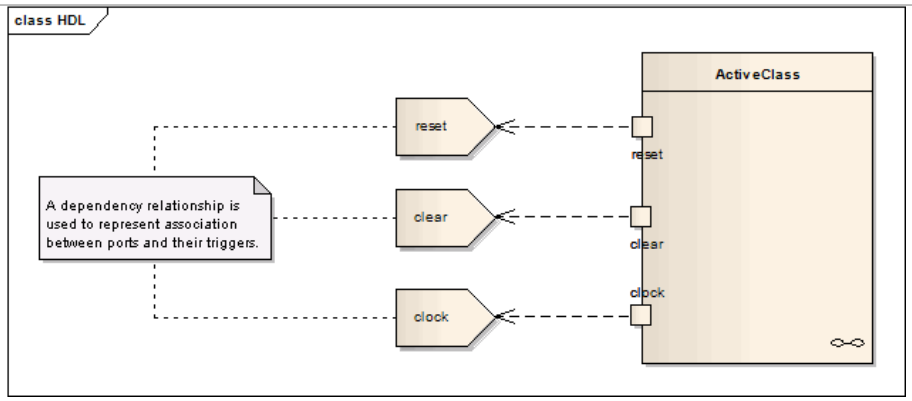
In an HDL StateMachine model, you might expect to:

- Designate Driving Triggers
- Establish Port–Trigger Mapping
- Add to Active State Logic

Operations

Operation	Description
Designate Driving Triggers	• A 'Change' Trigger is deemed to be an asynchronousTrigger if: - There is a transition from the actual SubMachine State (which encapsulates the actual logic) that it triggers, and - The target State of that transition has a self transition triggered by the same Trigger • Asynchronous Triggers should be modeled according to this pattern: - The Trigger should be of type Change (specification: True / False)

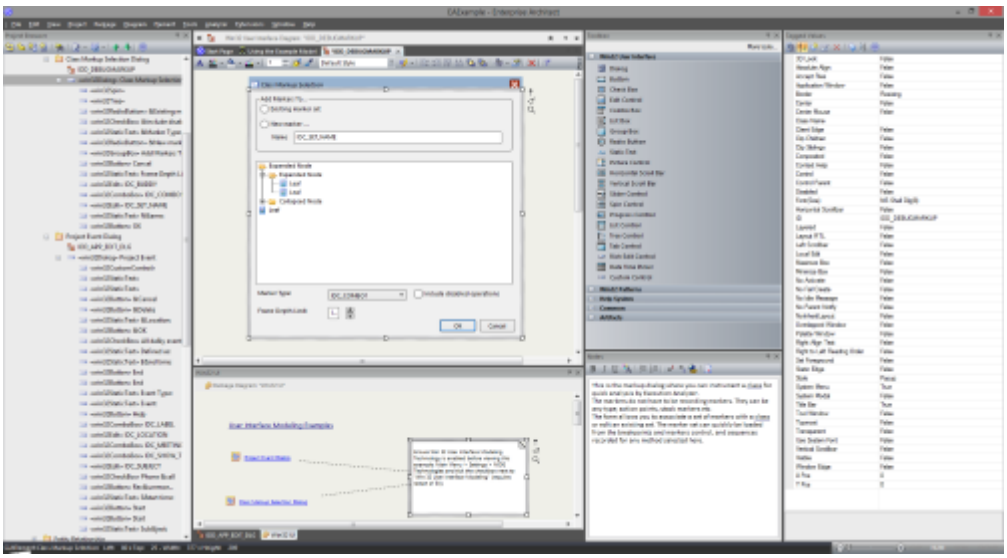
	- The active State (SubMachine State) should have a transition triggered by it - The target State of the triggered transition should have a self transition with the same Trigger • A Trigger of type 'Time', which triggers the transitions to the active state (SubMachine State), is deemed to be the Clock; the specification of this trigger should conform to the target language: - VHDL - rising_edge / falling_edge - Verilog - posedge / negedge - SystemC - positive / negative
Establish Port-Trigger Mapping	After successfully modeling the different operating modes of the component, and the Triggers associated with them, you must associate the Triggers with the component's Ports. A Dependency relationship from the Port to the associated Trigger is used to signify that association.

	
Active State Logic	Designating the driving Trigger and establishing the Port-Trigger mapping put in place the preliminaries required for efficiently interpreting the hardware components. We now model the actual StateMachine logic within the Active (SubMachine) State.

Notes

- To be able to generate code from behavioral models, all behavioral constructs should be contained within a Class
- The current code generation engine supports only one clock Trigger for a component

Win32 User Interface Dialogs



Using the MDG Win32 UI Technology, you can design user interface screens that render as Win32® controls. The user interface produced can be used in any resource definition script. Resource definition scripts, or RC files, are a Microsoft technology that - as for other code - can be compiled and the assets used by native desktop applications. User interface screens or dialogs can be created from scratch or reverse engineered. User interface models can also be forward engineered using the synchronize code function (F7).

Interface modeling takes place on diagrams in the exact same fashion as you would work with any technology in Enterprise Architect. An interesting aspect of User Interface design in Enterprise Architect is that components can take an active role in the simulation of StateMachines and Activities, enabling a simulation to interact with users, much like a real program!

Access

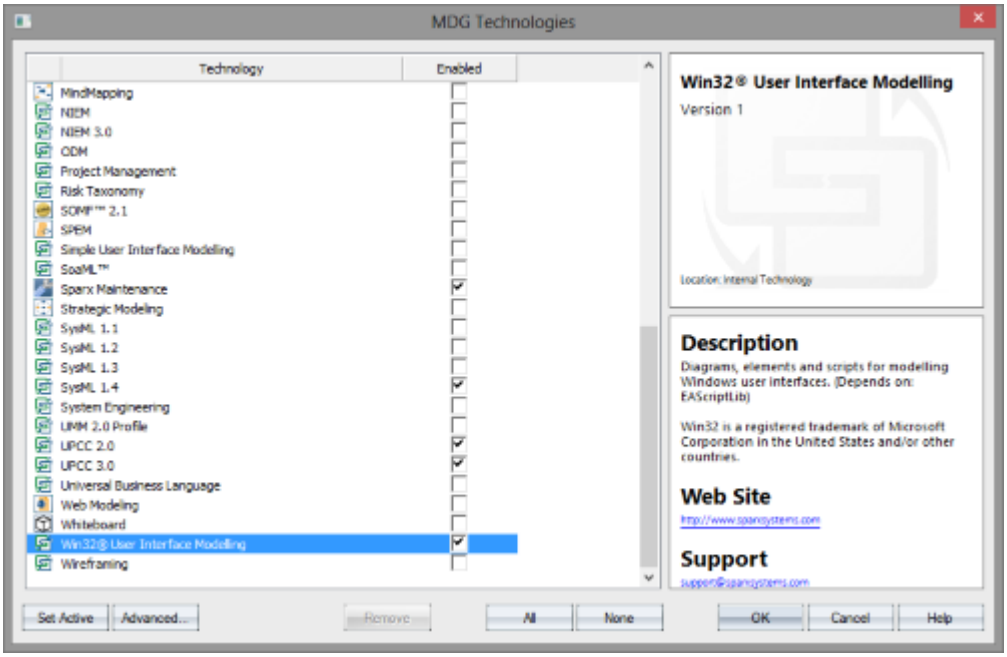
Display the 'Diagram Builder' using any of the methods outlined in the table below. Then, in the 'Diagram Builder', click on the ≡ icon and choose the perspective 'UX Design > Win32 UI Models'. In the 'Select From' panel, select 'User Interface - Win 32' and then select the Diagram Type 'UserInterfaceWin32'. Finally, click on 'Create Diagram' to create a new diagram on which to begin modeling your Win 32 User Interface.

Ribbon	Design > Diagram > Add Diagram
Context Menu	Right-click on Package | New Diagram
Other	Browser window caption bar menu | New Diagram

Support

The MDG Win32® User Interface Technology is available in the Enterprise Architect Professional, Corporate, Unified and Ultimate editions

Enabling Win32 User Interface Technology



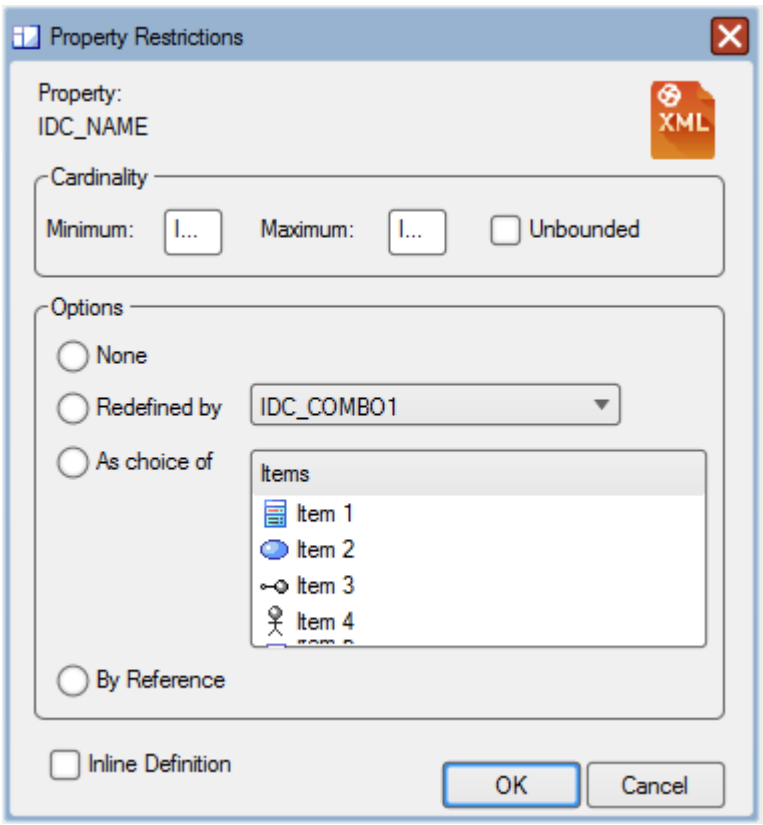
The Win32® UI Technology in Enterprise Architect is enabled or disabled using the 'MDG Technologies' dialog (select the 'Specialize > Technologies > Manage Technology' ribbon option).

Default technology

You can set the MDG Win32® UI Technology as the active default technology to access the Toolbox pages directly.

Modeling UI Dialogs

The Win32 User Interface MDG Technology provides the tools to help you design a user interface that closely emulates the visual style and available options for Windows dialogs.



Win32 Dialog

These user interface components are supported, each matching the equivalent-named RC resource.

Component	Details
win32Dialog	The equivalent of the RC format DIALOG and DIALOGEX resources.

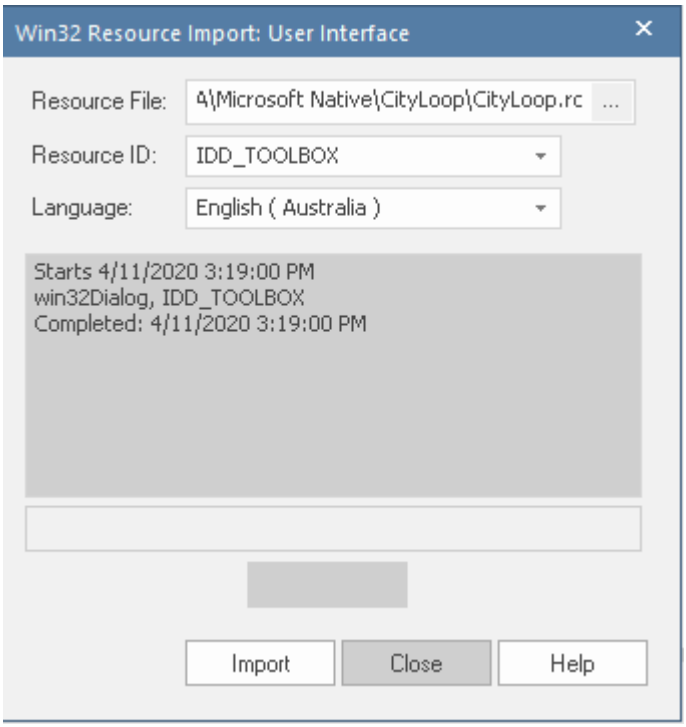
win32StaticText	The equivalent of the RC format LTEXT, RTEXT, CTEXT resources.
win32Edit	The equivalent of the RC format EDITTEXT resource.
win32Button	The equivalent of the RC format BUTTON, DEFPUSHBUTTON and other resources.
win32CheckBox	The equivalent of the RC format CHECKBOX resource.
win32ScrollBarH	The equivalent of the RC format SCROLLBAR resource with SBS_HORZ style
win32ScrollBarV	The equivalent of the RC format SCROLLBAR resource with SBS_VERT style.
win32GroupBox	The equivalent of the RC format GROUPBOX resource.
win32ComboBox	The equivalent of the RC format COMBOBOX resource. Note: When you initially drag the 'Combo Box' icon - of type 'Drop Down' or 'Drop

	Down List' - onto a diagram, the middle 'tracking handle' on each side of the element is white, indicating that you can only adjust the width of the element. To adjust the height of the element as well as the width, click on the drop-down arrow part of the image; the middle 'tracking handle' on the bottom edge is now white, indicating that you can drag the base down to set the virtual height (the height of the element when it is expanded to show all possible values in the drop-down list).
win32ListBox	The equivalent of the RC format LISTBOX resource.
win32RadioButton	The equivalent of the RC format RADIOBUTTON resource.
win32TabPane	The equivalent of the RC format TABPANE resource.
win32Picture	The equivalent of the RC format STATIC resource with SS_BITMAP style. The control can render an image when applied from your model. An image can be applied by selecting it first and pressing Ctrl+Shift+W to display the

	Image Manager. Afterwards, you might need to change the value of the resource ID in the appropriate Tagged Value.
win32CustomControl	The equivalent of the RC format CONTROL resource.

Import Single Dialog from RC File

You can quickly import a single dialog by name.



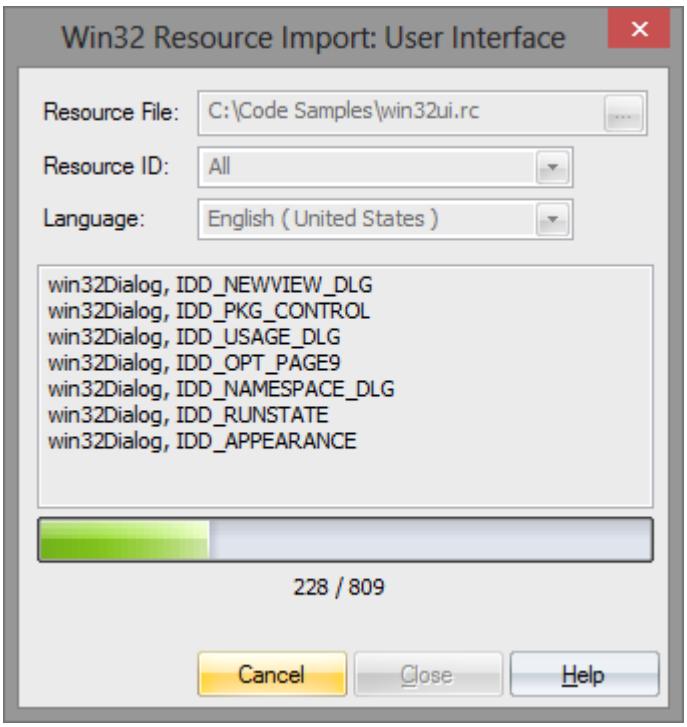
Access

In the Browser window, click on the target Package.

Ribbon	Develop > Source Code > Files > Import Resource Script

Import All Dialogs from RC File

All dialogs in a single RC file can be imported into your model. This image was captured one minute into the import, at which time over 200 large dialog definitions had been imported.

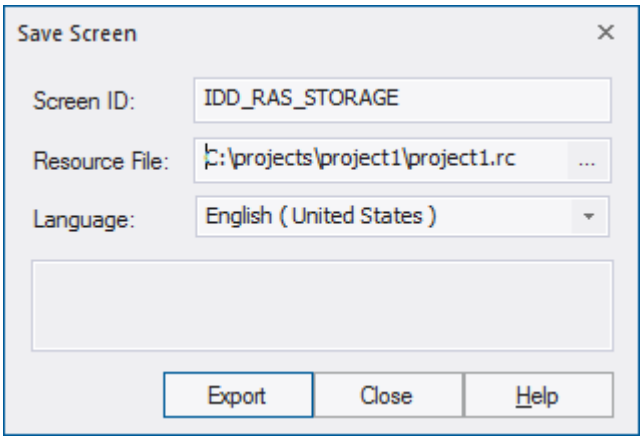


Access

Ribbon	Develop > Source Code > Files > Import Resource Script

Export Dialog to RC File

Once a screen design is modified or a new one created, you might want to get it back to the RC file you use to build your application, so that you can see how it looks with real data. Begin by selecting the Win32Dialog element in the Browser window, then use the ribbon to perform the synchronization.



Access

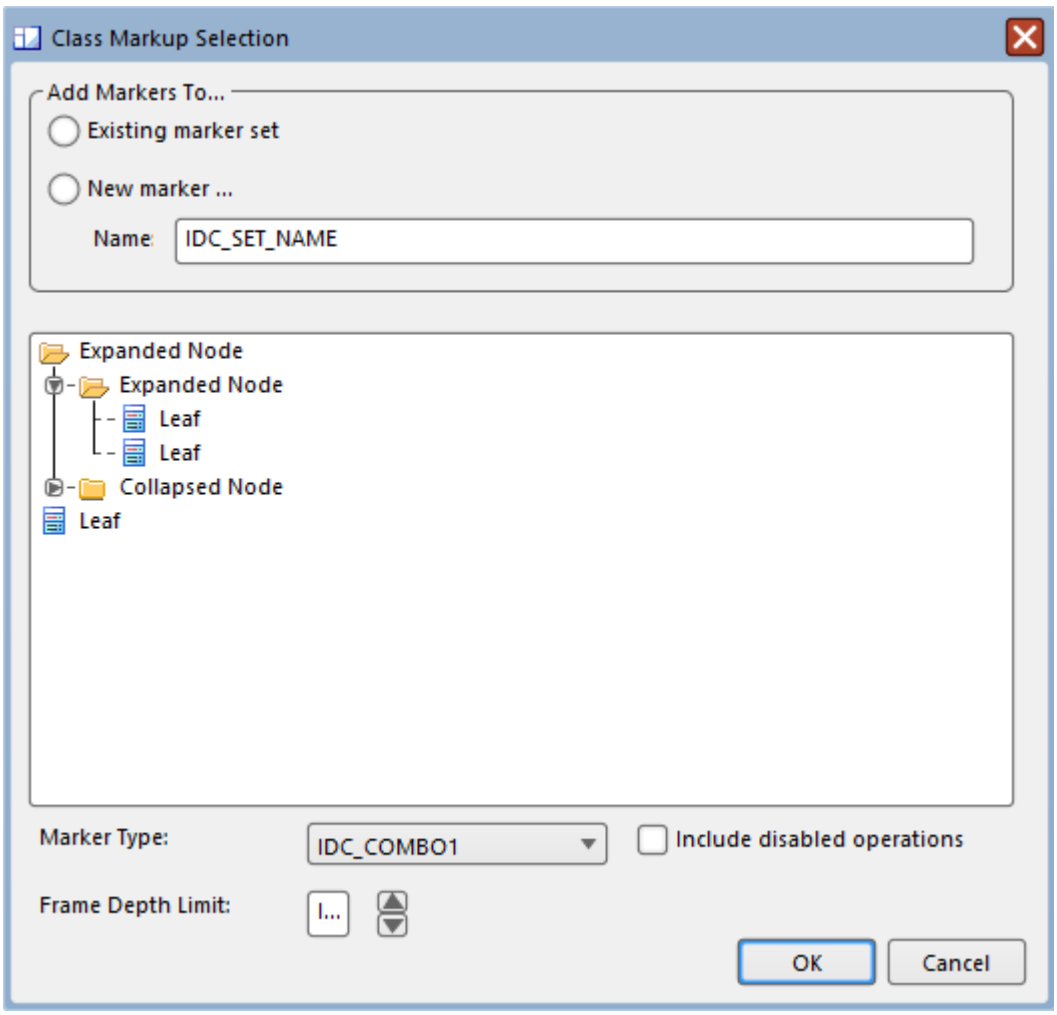
Click on the win32Dialog element.

Ribbon	Develop > Source Code > Generate > Generate Single Element
Keyboard Shortcuts	F11

Design a New Dialog

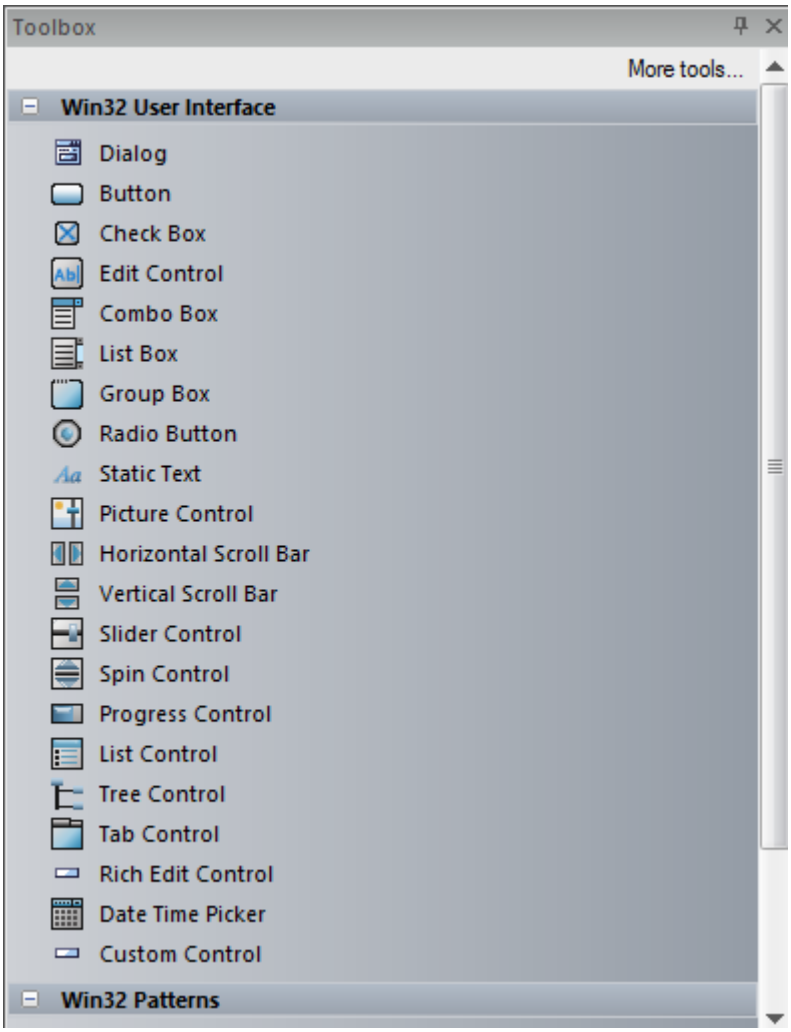
Creating a new Win32 dialog is easy and mostly visual. You will probably need a workspace that shows:

- The new diagram (select the 'Design > Diagram > Add Diagram > User Interface - Win32 > User Interface - Win32' ribbon path)
- The Win32 User Interface Toolbox (select the 'Design > Diagram > Tooolbox' ribbon option) and
- The Tagged Values tab of the Properties window



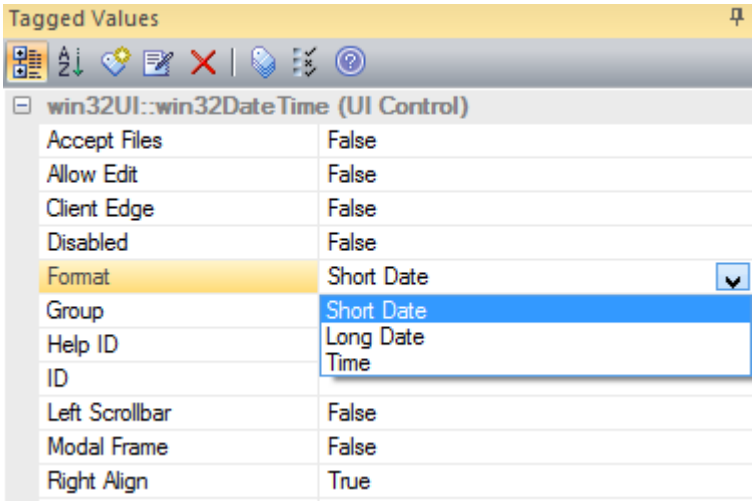
The UI Toolbox

All of the common RC elements can be found on the UI toolbox



The Tags Tab

This tab is provided on the Properties window and 'Properties' dialog for an object, and is where all the properties of a control can be viewed and edited.



Using the Picture Control

Images from your model (see *Image Manager*) can be applied by selecting the control on the dialog and pressing Ctrl+Shift+W. You might have to enter the value of the resource ID in the appropriate Tagged Value.

Note

- You can copy and paste dialog Packages

Gang of Four (GoF) Patterns

A Design Pattern is a template for solving commonly recurring design problems; it consists of a series of elements and connectors that can be reused in a new context. The advantage of using Patterns is that they have been tested and refined in a number of contexts and so are typically robust solutions to common problems. Enterprise Architect provides the Gang of Four Patterns as an MDG Technology that can be loaded into the current repository.

The Gang of Four (Gof) Patterns are a group of twenty three Design Patterns originally published in a seminal book entitled *Design Patterns: Elements of Reusable Object-Oriented Software*; the term 'Gang of Four' refers to the four authors. Enterprise Architect displays these Patterns in its Pattern engine, helping you to visualize the elements of the Pattern and adjust the Pattern to the context of your software design problem.

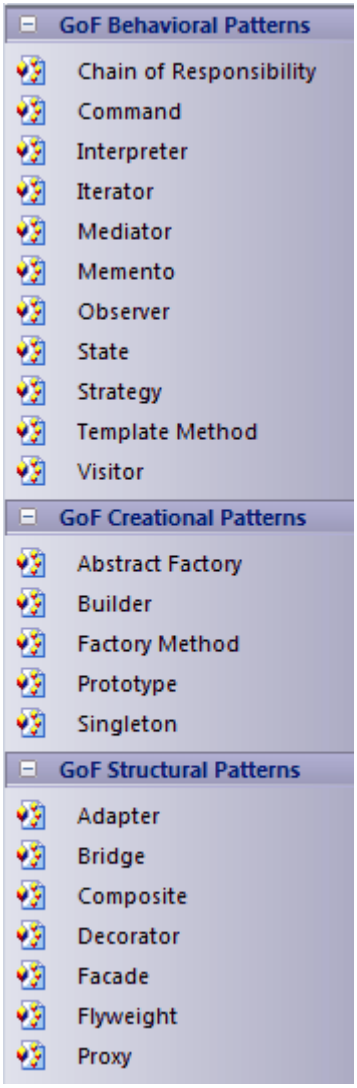
GoF Patterns in Enterprise Architect

Features	Description
GoF Pattern Facilities	The GoF Patterns are provided in the form of: • GoF Behavioral Patterns, GoF Creational Patterns and GoF Structural Patterns pages in the Toolbox

- Gang of Four Pattern entries in the Toolbox Shortcut Menu

GoF Pattern Toolbox Pages

You can access the 'GoF Pattern' pages of the Toolbox by clicking on to display the 'Find Toolbox Item' dialog and specifying 'GoF Patterns'; these icons are available:



When you drag one of the Pattern elements onto a new diagram, the 'Add

	Pattern GoF <pattern group><pattern type>' dialog displays; if necessary, modify the action and/or default for the component elements, then click on the OK button to create a diagram based on the Pattern.

ICONIX

The ICONIX process is a proprietary software development methodology based on UML. The process is Use Case driven and uses UML-based diagrams to define four milestones. The main feature of the process is a concept called robustness modeling, based on the early work of Ivar Jacobson, which helps bridge the gap between analysis and design.

This text is derived from the ICONIX entry in the online Wikipedia:

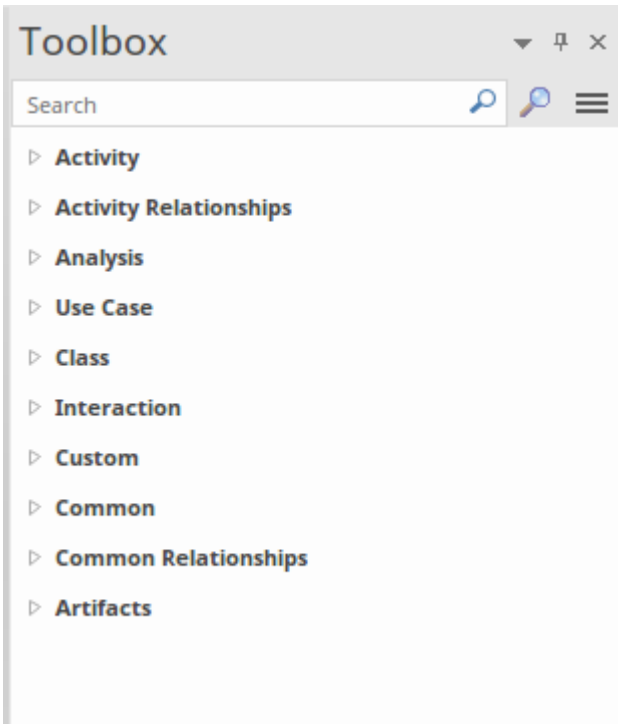
'The ICONIX Process is a minimalist, streamlined approach to Use Case driven UML modeling that uses a core subset of UML diagrams and techniques to provide thorough coverage of object-oriented analysis and design. Its main activity is robustness analysis, a method for bridging the gap between analysis and design. Robustness analysis reduces the ambiguity in use case descriptions, by ensuring that they are written in the context of an accompanying domain model. This process makes the use cases much easier to design, test and estimate.'

The ICONIX Process was developed by Doug Rosenberg; for more information on ICONIX, refer to ICONIX Software Engineering Inc.

Aspects

Aspect	Detail

ICONIX in Enterprise Architect	Enterprise Architect enables you to develop models under ICONIX quickly and simply, through use of an MDG Technology integrated with the Enterprise Architect installer. The ICONIX facilities are provided in the form of: • A set of ICONIX pages in the Toolbox • ICONIX element and relationship entries in the 'Toolbox Shortcut' menu and Quick Linker To further help you develop and manage a project under ICONIX, Enterprise Architect also provides a white paper on the ICONIX Roadmap.
ICONIX Toolbox Pages	Within the Toolbox, Enterprise Architect provides ICONIX versions of the pages for UML Analysis, Use Case, Class, Interaction (Sequence), Activity and Custom diagrams (which often form the basis for Robustness diagrams). Compared to the standard Toolbox pages, these have slightly different element and relationship sets; you can access them by either: • Specifying 'ICONIX' in the 'Find Toolbox Item' dialog and selecting a

	specific Toolbox page • Selecting the 'ICONIX' option in the drop-down field of the Default Tools toolbar, which adds all six pages to the Toolbox; all pages are closed up 

Configuration Settings

You can set the default code options such as the editors for each of the programming languages available for Enterprise Architect and special options for how source code is generated or reverse engineered. These options are defined according to whether they apply to:

- All users of the current model, set on the 'Manage Model Options' dialog, or
- All models that you access (other users can define their own settings that apply to the same models), set on the 'Preferences' dialog

You can also:

- For each programming language used in the model, for all users working on the model, define Collection Classes for generating code from Association connectors where the target role has a multiplicity setting greater than 1
- Define a local path for yourself, using the 'Local Path' dialog; these settings apply to all Enterprise Architect models that you access
- Define language macros within the model, which are useful in reverse engineering and can be exported from and imported to the model

Source Code Engineering Options

The 'Source Code Engineering' options apply to the languages in which you generate code from Enterprise Architect. They are divided into Model-specific options and User-specific options, as explained here.

Model-Specific Options

These options are defined on the 'Manage Model Options' dialog.

Access

Ribbon	Settings > Model > Options > Source Code Engineering

Types of Option

Option Type	Detail
Source Code Generation Options	You can define a number of settings for generating code in the model, such as the default language to generate code in and the Unicode character set for code

	generation.
Options - Object Lifetimes	You can configure various options concerning Object Lifetimes.
Code Language Options	For each of the code languages that Enterprise Architect supports, you can define the model-specific options and set any Collection Classes required.

User-Specific Options

These options are defined on the 'Preferences' dialog.

Access

On the 'Preferences' dialog, click on 'Source Code Engineering' in the left-hand list.

Ribbon	Start > Appearance > Preferences > Preferences
Keyboard Shortcuts	Ctrl+F9

Types of Option

Option Type	Detail
Source Code Generation Options	You can define a number of settings for generating code in any model that you access under the same user ID.
Code Editors	These are options for accessing and configuring the source code editor.
Attributes/Operations	Use these options for configuring attributes and operations.
Code Language Options	For each of the code languages that Enterprise Architect supports, you can define the user-specific options that apply to any model that you access under your user ID.

Code Generation Options

When you generate code for your model, you can set certain options. These include:

- The default language
- Whether to generate methods for implemented interfaces
- The Unicode options for code generation

Access

Ribbon	Settings > Model > Options > Source Code Engineering

Configure code generation options

Option	Action
Always synchronize with existing file (recommended)	Select the radio button to synchronize imported code with an existing file.
Replace	Select the radio button to overwrite the

(overwrite) existing source file	existing source file with imported code.
Component Types	Click on this button to open the 'Import component types' dialog, to set up the importation of component types.
Default Language for Code Generation	Click on the drop-down arrow and select the default language for code generation.
DDL Name Templates	Click on the ... button to define the template names for Primary Key, Unique Constraint, Foreign Key and Foreign Key Index Name templates.
Default name for associated attrib	Type in a default name to be generated from imported attributes.
Generate methods for implemented interfaces	Select the checkbox to indicate that methods are generated for implemented interfaces.
Code page for source	Click on the drop-down arrow and select the appropriate Unicode character

editing	embedding format to apply.

Notes

- It is worthwhile to configure these settings, as they serve as the defaults for all Classes in the model; you can override most of these on a per-Class basis using the custom settings (from the 'Code Generation' dialog)

Import Component Types

Using the 'Import Component Types' dialog you can configure what elements you want to be created for files of any extension found while importing a source code directory.

Access

Ribbon	Settings > Model > Options > Source Code Engineering: Component Types

Define Import Component Types

Option	Action
Extension	Type in the extension name for a component type.
Type	Click on the drop-down arrow and select the component type.
Stereotype	Type in any stereotype name that further identifies a component of this type.

Component List	Lists the currently-defined component types.
Save	Click on this button to saves the component definition and add it to the component list.
New	Click on this button to clear the dialog fields so that you can define a new component type.
Delete	Click on this button to delete the selected component type from the component list.

Notes

- You can transport these import component types between models, using the 'Settings > Model > Transfer > Export Reference Data' and 'Import Reference Data' ribbon options

Source Code Options

You can set a wide range of options for generating code in the models you work with. These include:

- How to format the generated code
- How to respond to certain events during code generation
- Whether to generate a diagram from the code

Access

On the 'Preferences' dialog, select the 'Source Code Engineering' option

Ribbon	Start > Appearance > Preferences > Preferences
Keyboard Shortcuts	Ctrl+F9

Configure code generation options

Field	Action
Wrap long comment lines at	Type in the number of characters to allow in a comment line before wrapping the text to the next line.

Auto Layout Diagram on Import	Click on the drop-down arrow and select if and when a diagram is automatically generated on code import.
Default Layout Diagram type	Click on the drop-down arrow and select the layout type to apply to diagrams generated from code.
Output files use both CR & LF	Select the checkbox to include carriage returns and line feeds; set this option according to what operating system is currently in use, as code might not render correctly.
Prompt when synchronizing (reversing)	Select the checkbox to display a prompt when synchronization occurs.
Remove hard breaks from comments on import	Select the checkbox to remove hard breaks from commented sections on importation.
Auto generate role names when creating code	Select the checkbox to generate role names when creating code.

Do not generate members where association direction is 'Unspecified'	Select the checkbox to prevent generation of members if the Association direction is unspecified.
Create dependencies for operation returns and parameter types	Select the checkbox to generate dependencies for operation returns and parameter types.
Comments: Generate	Select the checkbox to generate comments.
Comments: Reverse	Select the checkbox to generate reverse comments.
Remove prefixes when generating Get/Set properties	Type in the prefixes, separated by semi-colons, used in your variable naming conventions, to be removed in the variables' corresponding get/set functions.
Treat as	Select the checkbox to use the prefixes

suffixes	defined in the 'Remove prefixes when generating Get/Set properties' field as suffixes.
Capitalized Attribute Name for Properties	Select the checkbox to capitalize attribute names for properties.
Use 'Is' for Boolean property Get()	Select the checkbox to use the Is keyword for the Boolean property Get().

Notes

- It is worthwhile to configure these settings, as they serve as the defaults for all Classes in the model; you can override most of these on a per-Class basis using the custom settings (from the 'Code Generation' dialog)

Options - Code Editors

You access the source code editor options via the 'DDL' page of the 'Preferences' dialog. On this page you can configure options for Enterprise Architect's internal editor, as well as the default editor for DDL scripts. You can configure external editors for code languages on each language options page.

Access

On the 'Preferences' dialog, select the 'Source Code Engineering > Code Editors' option.

Ribbon	Start > Appearance > Preferences > Preferences
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
DDL Editor	Defaults to blank, indicating that the Enterprise Architect code editor is the DDL editor in use.

	You can select a different default editor if necessary; click on the ... button to browse for and select the required DDL editor. The editor name then displays in the 'DDL Editor' field.
Default Database	Click on the drop-down arrow and select the default database to be used.
MySQL Storage Engine	Click on the drop-down arrow and select the MySQL storage engine to be used.
Use inbuilt editor if no external editor set	Select the checkbox to use the inbuilt editor for code in any language if no external editor is defined for that language in the user-specific options.
Show Line Numbers	Select the checkbox to display line numbers in the editor.
Show Structure Tree	Select the checkbox to show a tree with the results of parsing the open file (if the file is parsed successfully).
Automatically Reverse Engineer on File Save	If you select this checkbox, pressing Ctrl+S to save in the source code editor automatically reverse engineers the code in the same way as the Save Source and

	Re-Synchronize Class button does.
Don't parse files larger than	Click on the drop-down arrow and select the upper limit on file size for parsing. Setting this option prevents performance decrease due to parsing very large files.
Font, Style and Syntax Highlighting	Click on the button to display the 'Editor Language Properties' dialog, in which you can set both global and language-specific editor language properties.
Configure Enterprise Architect File Associations	Click on the button to display the 'Set Associations for a Program' dialog, and select the file extensions for files that you want to open through the Enterprise Architect Document Handler.

Editor Language Properties

Using the 'Editor Language Properties' dialog, you can specify syntax highlighting properties for any of the programming languages that Enterprise Architect supports at installation.

Access

In the 'Preferences' dialog, select the 'Source Code Engineering | Code Editors' option and click on the ... button next to 'Syntax Highlighting Options'.

Ribbon	Start> Appearance > Preferences > Preferences, select 'Source Code Engineering | Code Editors' option > click on the ... button next to 'Syntax Highlighting Options'
Other	In the Code Editor window, click on the toolbar icon | Syntax Highlighting Options

Options

Panel	Description

Language Panel	The panel on the left of the dialog lists the languages for which you can set properties. At the top of the list are three non-language options: • (Dark Theme) - assigns a dark background to the property fields and to the code panel in the code editor screen (you can apply a different color to specific properties) • (Light Theme) - assigns a pale background to the property fields and to the code panel in the code editor screen (you can apply a different color to specific properties) You can also set the background themes on the 'Application Look' dialog • (Global) provides properties that you can set for all programming languages; however, you can reset a global property to a different value for a particular language, in the properties specifically for that language Resetting a global property for one language does not affect that property's value for the other languages Click on the required language in the list, to display the properties for that

	language: • Properties shown in bold indicate that this is the highest level at which this property can be defined (for most language options other than 'Global', this is effectively the only point at which the property is defined) • Properties shown in normal font are generally the global properties that you can reset just for the current language
Properties Panel	Scroll through the property categories and individual properties for the language. You can collapse and expand categories as necessary, using the expansion box next to the category name (⊟). When you click on a property name, an explanation of that property displays in the panel at the bottom right of the dialog. To define a property, click on the value field following the property name; depending on the type of property, either the field is enabled for direct editing or a drop-down arrow or ... button displays (as described for the 'Tags' tab of the Properties window) so that you can select the values to define the property. Select or type in the required values.

	Use the Toolbar icons to: • Save your changes to the properties • Reset all properties fields to the default settings shipped with Enterprise Architect • Reset the current style field to the default setting (not enabled for non-style fields)
Assign Keys to Macros	In the 'Macros' category of the properties, you can assign (Ctrl+Alt+<n>) keystroke combinations to coding macros that you have created yourself in the 'Source Code Viewer'. When you click on the Browse button in a selected 'Macro' field, the 'Open Macro' dialog displays; this dialog lists the existing macros and, if a key combination has been assigned to a macro, what that key combination is. Click on the name of the macro and on the Open button to assign the selected keys to the macro.

Notes

- You cannot currently set properties for any additional

languages you include through an MDG Technology

- You can resize this dialog, if required

Options - Object Lifetimes

You can use these options to configure various Object Lifetime settings such as:

- Defining constructor details when generating code
- Specifying whether to create a copy constructor
- Defining Destructor details

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Object Lifetimes

Options

Option	Action
Constructor	If necessary, select the checkboxes to specify that a constructor is generated and (for C++) that the constructor is in-line. Click on the drop-down arrow and select the appropriate visibility of the default constructor - Private, Protected or Public.

Copy Constructor	If necessary, select the checkboxes to specify that a copy constructor is generated and (for C++) that the copy constructor is in-line. Click on the drop-down arrow and select the appropriate visibility of the default copy constructor - Private, Protected or Public.
Destructor	If necessary, select the checkboxes to specify that a destructor is generated and (for C++) that the destructor is in-line and/or virtual. Click on the drop-down arrow and select the appropriate visibility of the default destructor - Private, Protected or Public.

Options - Attribute/Operations

Your use of attributes and operations can be configured in a number of ways. You can set options to:

- Delete model attributes not included in the code during reverse synchronization
- Delete model methods not included in the code during reverse synchronization
- Delete code from features contained in the model during forward synchronization
- Delete model associations and aggregations that correspond to attributes not included in the code during reverse synchronization
- Define whether or not the bodies of methods are included and saved in the model when reverse engineering
- Create features in quick succession, clearing the Properties window when you click on 'Save' so that you can enter another feature name

You configure these options on the 'Attribute/Operations' page of the 'Preferences' dialog.

Access

On the 'Preferences' dialog, select the 'Source Code Engineering > Attribute/Operations' option.

Ribbon	Start > Appearance > Preferences > Preferences

Keyboard Shortcuts	Ctrl+F9

Options

Field	Action
On reverse synch, delete model attributes not in code	Select the checkbox to indicate that on reverse synchronization, attributes in the model that are not included within code are automatically removed from the model.
On reverse synch, delete model associations not in code	Select the checkbox to indicate that on reverse synchronization, associations in the model that are not included within code are automatically removed from the model.
On reverse synch, delete model methods not in code	Select the checkbox to indicate that on reverse synchronization, methods in the model that are not included within code are automatically removed from the model.
Include	Select the checkbox to indicate that on

method bodies in model when reverse engineering	reverse engineering code, method bodies in the code are included within your model.
After save, re-select edited item	Select the checkbox to indicate that after saving an attribute or operation, the properties definition continues to display the details of the selected feature. If deselected, indicates that the fields of the properties definition will clear so that you can enter another attribute or operation name and details immediately.
On forward synch, prompt to delete code features not in model	Select the checkbox to indicate that, during forward synchronization, the 'Synchronize Element <package name>.<element name>' dialog displays, so that you can either ignore, reassign or delete features in the code that are not in the model.

Modeling Conventions

The synchronization between UML models and programming code is achieved using a set of modeling conventions (mappings) between UML constructs and programming code syntax. The Software Engineer is advised to become familiar with these conventions in order to work with the code generation process for the programming languages they intend to target. There are a range of constructs used, including elements, features, connectors, connector ends, stereotypes and Tagged Values. The newcomer will require a little time to become familiar with these conventions but after a short time they will be translating between programming code and UML constructs without effort.

Supported Languages

Language
Action Script
Ada 2012 (Unified and Ultimate Editions)
C

C#
C++
Delphi
Java
PHP
Python
SystemC (Unified and Ultimate Editions)
Verilog (Unified and Ultimate Editions)
VHDL (Unified and Ultimate Editions)
Visual Basic
Visual Basic .NET

Notes

Enterprise Architect incorporates a number of visibility indicators or scope values for its supported languages; these

include, for:

- All languages - Public (+), Protected (#) and Private (-)
- Java - Package (~)
- Delphi - Published (^)
- C# - Internal (~), Protected Internal (^)
- ActionScript - Internal (~)
- VB.NET - Friend (~), Protected Friend (^)
- PHP - Package (~)
- Python - Package (~)
- C - Package (~)
- C++ - Package (~)

ActionScript Conventions

Enterprise Architect supports round trip engineering of ActionScript 2 and 3, where these conventions are used.

Stereotypes

Stereotype	Applies To
literal	Operation Corresponds To: A literal method referred to by a variable.
property get	Operation Corresponds To: A 'read' property.
property set	Operation Corresponds To: A 'write' property.

Tagged Values

Tag	Applies To
attribute_name	Operation with stereotype property get or property set

	Corresponds To: The name of the variable behind this property.
dynamic	Class or Interface Corresponds To: The 'dynamic' keyword.
final	ActionScript 3: Operation Corresponds To: The 'final' keyword.
intrinsic	ActionScript 2: Class Corresponds To: The 'intrinsic' keyword.
namespace	ActionScript 3: Class, Interface, Attribute, Operation Corresponds To: The namespace of the current element.
override	ActionScript 3: Operation Corresponds To: The 'override' keyword.
prototype	ActionScript 3: Attribute Corresponds To: The 'prototype' keyword.
rest	ActionScript 3: Parameter Corresponds To: The rest parameter (...)

Common Conventions

- Package qualifiers (ActionScript 2) and Packages (ActionScript 3) are generated when the current Package is not a namespace root
- An unspecified type is modeled as 'var' or an empty 'Type' field

ActionScript 3 Conventions

- The Is Leaf property of a Class corresponds to the sealed keyword
- If a namespace tag is specified it overrides the Scope that is specified

Ada 2012 Conventions

Enterprise Architect supports round trip engineering of Ada 2012, where these conventions are used.

Stereotypes

Stereotype	Applies To
adaPackage	Class Corresponds To: A Package specification in Ada 2012 without a tagged record.
adaProcedure	Class Corresponds To: A procedure specification in Ada 2012.
delegate	Operation Corresponds To: Access to a subprogram.
enumeration	Inner Class Corresponds To: An enumerated type.
struct	Inner Class Corresponds To: A record definition.
typedef	Inner Class

	Corresponds To: A type definition, subtype definition, access type definition, renaming.

Tagged Values

Tag	Applies To
Aspect	Inner Class with stereotype typedef Operation Corresponds to: Aspect specification (Precondition and Postcondition of Subprogram type 'invariant', subtype 'predicate').
InstantiatedUnitType	Inner Class with stereotype typedef Corresponds To: The instantiated unit's type (Package / Procedure / Function).
IsAccess	Parameter Corresponds To: Determination of whether the parameter is an access variable.
IsAliased	Function parameter Corresponds to: Aliased function

	parameter.
Discriminant	Inner Class with stereotype typedef Corresponds To: The type's discriminant.
PartType	Inner Class with stereotype typedef Corresponds To: The part type ('renames' or 'new').
Type	Inner Class with stereotype typedef Corresponds To: If 'Value' = 'SubType', set 'subtype' If 'Value' = 'Access', set 'access type'.

Other Conventions

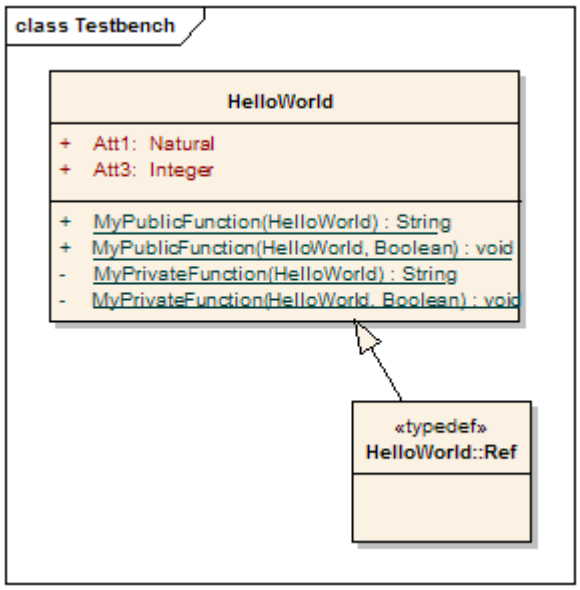
- Appropriate type of source files: Ada specification file, .ads
- Ada 2012 imports Packages defined as either <<adaPackage>> Class or Class, based on the settings in the Ada 2012 options
- A Package in the Ada specification file is imported as a Class if it contains a Tagged Record, the name of which is governed by the options 'Use Class Name for Tagged Record' and 'Alternate Tagged Record Name'; all attributes defined in that Tagged Record are absorbed as the Class's attributes

- A procedure / function in an Ada specification file is considered as the Class's member function if its first parameter satisfies the conditions specified in the options 'Ref Param Style', 'Ignore Reference parameter name' and 'Ref parameter name'
- The option 'Define Reference for Tagged Record', if enabled, creates a reference type for the Class, the name of which is determined by the option 'Reference Type Name'; for example:

```
HelloWorld.ads
package HelloWorld is
    type HelloWorld is tagged record
        Att1: Natural;
        Att3: Integer;
    end record;

    -- Public Functions
    function MyPublicFunction (P: HelloWorld)
return String;
    procedure MyPublicFunction (P1: in out
HelloWorld; AFlag: Boolean);
private
    -- Private Functions
    function MyPrivateFunction (P: HelloWorld)
return String;
    procedure MyPrivateFunction (P1: in out
HelloWorld; AFlag: Boolean);
```

```
end HelloWorld;
```



Notes

- Ada 2012 support is available in the Unified and Ultimate Editions of Enterprise Architect

C Conventions

Enterprise Architect supports round trip engineering of C, where these conventions are used:

Stereotype

Stereotype	Applies To
enumeration	Inner Class Corresponds To: An enumerated type.
struct	Inner Class Corresponds To: A 'struct' type.
Attribute	A keyword struct in variable definition.
typedef	Inner Class Corresponds To: A 'typedef' statement, where the parent is the original type name.
union	Inner Class Corresponds To: A union type.
Attribute	A keyword union in variable definition.

Tagged Values

Tag	Applies To
anonymous	Class also containing the Tagged Value typedef Corresponds To: The name of this Class being defined only by the typedef statement.
bitfield	Attribute Corresponds To: The size, in bits, allowed for storage of this attribute.
bodyLocation	Operation Corresponds To: The location the method body is generated to; expected values are header, classDec or classBody.
typedef	Class with stereotype other than 'typedef' Corresponds To: This Class being defined in a 'typedef' statement.
typeSynonyms	Class Corresponds To: The 'typedef' name and/or fields of this type.

C Code Generation for UML Model

UML	C Code
A Class	A pair of C files (.h + .c) Notes: File name is the same as Class name
Operation (public & protected)	Function declaration in .h file and definition in .c file Notes:
Operation (private)	Function definition in .c file only Notes:
Operation (static)	Function definition in .c file only Notes: Static functions will only appear in the .c file regardless of their scope.
Attribute (public & protected)	Variable definition in .h file Notes:
Attribute (private)	Variable definition in .c file Notes:

Inner Class (without stereotype)	(N/A) Notes: This inner Class would be ignored

Capture #define value to be generated in C code

For example, #define PI 3.14.

Step	Process
1	Add an attribute to the Class, with Name = PI and Initial Value = 3.14.
2	In the properties panel of the 'Attributes' page, update the 'Static' and 'Const' fields.
3	On the 'Tagged Values' tab of the 'Attributes' page, add a tag called 'define' with the value True.

Notes

- Separate conventions apply to Object Oriented programming in C

Object Oriented Programming In C

In Enterprise Architect, you apply a number of conventions for Object-Oriented programming in C.

To configure the system to support Object-Oriented programming using C, you must set the 'Object Oriented Support' option to True on the 'C Specifications' page of the 'Preferences' dialog.

Stereotypes

Stereotype	Applies To
enumeration	Class Corresponds To: An enumerated type.
struct	Class Corresponds To: A 'struct' type.
Attribute	A keyword struct in variable definition.
typedef	Class Corresponds To: A 'typedef' statement, where the parent is the original type name.
union	Class

	Corresponds To: A union type.
Attribute	A keyword union in variable definition.

Tagged Values

Tag	Applies To
anonymous	Class with stereotype of 'enumeration', 'struct' or 'union' Corresponds To: The name of this Class being defined only by the typedef statement.
bodyLocation	Operation Corresponds To: The location the method body is generated to; expected values are 'header', 'classDec' or 'classBody'.
define	Attribute Corresponds To: '#define' statement.
typedef	Class with stereotype of 'enumeration', 'struct' or 'union' Corresponds To: This Class being defined in a 'typedef' statement.

Object-Oriented C Code Generation for UML Model

The basic idea of implementing a UML Class in C code is to group the data variable (UML attributes) into a structure type; this structure is defined in a .h file so that it can be shared by other Classes and by the client that referred to it.

An operation in a UML Class is implemented in C code as a function; the name of the function must be a fully qualified name that consists of the operation name, as well as the Class name to indicate that the operation is for that Class.

A delimiter (specified in the 'Namespace Delimiter' option on the 'C Specifications' page) is used to join the Class name and function (operation) name.

The function in C code must also have a reference parameter to the Class object - you can modify the 'Reference as Operation Parameter', 'Reference Parameter Style' and 'Reference Parameter Name' options on the 'C Specifications' page to support this reference parameter.

Limitations of Object-Oriented Programming in C

- No scope mapping for an attribute: an attribute in a UML Class is mapped to a structure variable in C code, and its scope (private, protected or public) is ignored

- Currently an inner Class is ignored: if a UML Class is the inner Class of another UML Class, it is ignored when generating C code
- Initial value is ignored: the initial value of an attribute in a UML Class is ignored in generated C code

C# Conventions

Enterprise Architect supports the round trip engineering of C#, where these conventions are used.

Stereotypes

Stereotype	Applies To
enumeration	Class Corresponds To: An enumerated type.
event	Operation Corresponds To: An event.
extension	Operation Corresponds To: A Class extension method, represented in code by a 'this' parameter in the signature.
indexer	Operation Corresponds To: A property acting as an index for this Class.
partial	Operation Corresponds To: The 'partial' keyword on an operation.

property	Operation Corresponds To: A property possibly containing both read and write code.
record	Class Corresponds To: A 'record' type.
record_struct	Class Corresponds to: A 'record struct' type.
struct	Class Corresponds To: A 'struct' type.

Tagged Values

Tag	Applies To
argumentName	Operation with stereotype extension Corresponds To: The name given to this parameter.
attribute_name	Operation with stereotype property or event Corresponds To: The name of the variable behind this property or event.

className	Operation with stereotype extension Corresponds To: The Class that this method is being added to.
const	Attribute Corresponds To: The const keyword.
ConstructorParameters	Class, <<struct>>Class Corresponds To: The parameter(s) for primary constructor of class and struct definition.
definition	Operation with stereotype partial Corresponds To: Whether this is the declaration of the method, or the definition.
delegate	Operation Corresponds To: The 'delegate' keyword.
enumType	Operation with stereotype property Corresponds To: The datatype that the property is represented as.
expressionBody	Operation, Operation with stereotype property or indexer Corresponds To: 'True' if the 'Behavior

	Code' is from an expression body function member.
extensionAttribute	Operation with stereotype extension. Corresponds to: The attribute given to this parameter.
extern	Operation Corresponds To: The 'extern' keyword.
fixed	Attribute Corresponds To: The 'fixed' keyword.
generic	Operation Corresponds To: The generic parameters for this operation.
genericConstraints	Templated Class or Interface, Operation with tag 'generic' Corresponds To: The constraints on the generic parameters of this type or operation.
Implements	Operation Corresponds To: The name of the method this implements, including the interface name.

ImplementsExplicit	Operation Corresponds To: The presence of the source interface name in this method declaration.
initializer	Operation Corresponds To: A constructor initialization list.
new	Class, Interface, Operation Corresponds To: The 'new' keyword.
override	Operation Corresponds To: The 'override' keyword.
params	Parameter Corresponds To: A parameter list using the 'params' keyword.
partial	Class, Interface Corresponds To: The 'partial' keyword.
propertyInitializer	Operation with stereotype property Corresponds To: A property initializer.
positionalParameters	<<record>> Class Corresponds To: The position parameter

	in the record definition.
readonly	Operation, <<struct>>Class Corresponds To: The 'readonly' keyword.
ref	Operation, <<struct>>Class Corresponds To: The 'ref' keyword.
required	Operation, Attribute Corresponds To: The 'required' keyword.
sealed	Operation Corresponds To: The 'sealed' keyword.
static	Class Corresponds To: The 'static' keyword.
unsafe	Class, Interface, Operation Corresponds To: The 'unsafe' keyword.
virtual	Operation Corresponds To: The 'virtual' keyword.
writeonly	Operation with stereotype property Corresponds To: This property only defining 'write' code.

Other Conventions

- Namespaces are generated for each Package below a namespace root
- The Const property of an attribute corresponds to the readonly keyword, while the tag const corresponds to the const keyword
- The value of inout for the Kind property of a parameter corresponds to the ref keyword
- The value of out for the Kind property of a parameter corresponds to the out keyword
- Partial Classes can be modeled as two separate Classes with the partial tag
- The Is Leaf property of a Class corresponds to the sealed keyword

C++ Conventions

Enterprise Architect supports round trip engineering of C++, including the Managed C++ and C++/CLI extensions, where these conventions are used.

Stereotypes

Stereotype	Applies To
enumeration	Class Corresponds To: An enumerated type.
friend	Operation Corresponds To: The 'friend' keyword.
property get	Operation Corresponds To: A 'read' property.
property set	Operation Corresponds To: A 'write' property.
struct	Class Corresponds To: A 'struct' type.
typedef	Class Corresponds To: A 'typedef' statement,

	where the parent is the original type name.
alias	Class Corresponds to an 'Alias' declaration, where the parent is the original type name.
union	Class Corresponds To: A union type.

Tagged Values

Tag	Applies To
afx_msg	Operation Corresponds To: The afx_msg keyword.
anonymous	Class also containing the Tagged Value typedef Corresponds To: The name of this Class being only defined by the typedef statement.
attribute_name	Operation with stereotype property get or property set

	Corresponds To: The name of the variable behind this property.
bitfield	Attribute Corresponds To: The size, in bits, allowed for storage of this attribute.
bodyLocation	Operation Corresponds To: The location the method body is generated to; expected values are header, classDec or classBody.
callback	Operation Corresponds To: A reference to the CALLBACK macro.
constexpr	Attribute and Operation Corresponds To: The constexpr keyword.
explicit	Operation Corresponds To: The 'explicit' keyword.
initializer	Operation Corresponds To: A constructor initialization list.
inline	Attribute and Operation Corresponds To: The 'inline' keyword and

	inline generation of the member variable definition and method body.
mutable	Attribute Corresponds To: The 'mutable' keyword.
scoped	Class with stereotype enumeration Corresponds To: Either the 'class' or 'struct' keyword.
throws	Operation Corresponds To: The exceptions that are thrown by this method.
typedef	Class with stereotype other than 'typedef' Corresponds To: This Class being defined in a 'typedef' statement.
typeSynonyms	Class Corresponds To: The 'typedef' name and/or fields of this type.
volatile	Operation Corresponds To: The 'volatile' keyword.

Other Conventions

- Namespaces are generated for each Package below a namespace root
- By Reference attributes correspond to a pointer to the type specified
- The Transient property of an attribute corresponds to the volatile keyword
- The Abstract property of an attribute corresponds to the virtual keyword
- The Const property of an operation corresponds to the const keyword, specifying a constant return type
- The Is Query property of an operation corresponds to the const keyword, specifying the method doesn't modify any fields
- The Pure property of an operation corresponds to a pure virtual method using the "= 0" syntax
- The Fixed property of a parameter corresponds to the const keyword

Managed C++ Conventions

These conventions are used for managed extensions to C++ prior to C++/CLI. In order to set the system to generate managed C++ you must modify the C++ version in the C++ Options.

Stereotypes

Stereotype	Applies To
property	Operation Corresponds To: The '__property' keyword.
property get	Operation Corresponds To: The '__property' keyword and a read property.
property set	Operation Corresponds To: The '_ _property' keyword and a 'write' property.
reference	Class Corresponds To: The '__gc' keyword.
value	Class

	Corresponds To: The '__value' keyword.

Tagged Values

Tag	Applies To
managedType	Class with stereotype reference, value or enumeration; Interface Corresponds To: The keyword used in declaration of this type; expected values are 'class' or 'struct'.

Other Conventions

- The typedef and anonymous tags from native C++ are not supported
- The Pure property of an operation corresponds to the keyword __abstract

C++/CLI Conventions

These conventions are used for modeling C++/CLI extensions to C++. In order to set the system to generate managed C++/CLI you must modify the C++ version in the C++ Options.

Stereotypes

Stereotype	Applies To
event	Operation Description: Defines an event to provide access to the event handler for this Class.
property	Operation, Attribute Description: This is a property possibly containing both read and write code.
reference	Class Description: Corresponds to the 'ref class' or 'ref struct' keyword.
value	Class Description: Corresponds to the 'value class' or 'value struct' keyword.

Tagged Values

Tag	Applies To
attribute_name	Operation with stereotype property or event Description: The name of the variable behind this property or event.
generic	Operation Description: Defines the generic parameters for this Operation.
genericConstraints	Templated Class or Interface, Operation with tag generic Description: Defines the constraints on the generic parameters for this Operation.
initonly	Attribute Description: Corresponds to the 'initonly' keyword.
literal	Attribute Description: Corresponds to the literal keyword.
managedTyp	Class with stereotype reference, value or

e	enumeration; Interface Description: Corresponds to either the 'class' or 'struct' keyword.

Other Conventions

- The typedef and anonymous tags are not used
- The property get/property set stereotypes are not used
- The Pure property of an operation corresponds to the keyword abstract

Delphi Conventions

Enterprise Architect supports round trip engineering of Delphi, where these conventions are used:

Stereotypes

Stereotype	Applies To
constructor	Operation Corresponds To: A constructor.
destructor	Operation Corresponds To: A destructor.
dispinterface	Class, Interface Corresponds To: A dispatch interface.
enumeration	Class Corresponds To: An enumerated type.
metaclass	Class Corresponds To: A metaclass type.
object	Class Corresponds To: An object type.

operator	Operation Corresponds To: An operator.
property get	Operation Corresponds To: A 'read' property.
property set	Operation Corresponds To: A 'write' property.
struct	Class Corresponds To: A record type.

Tagged Values

Tag	Applies To
attribute_name	Operation with stereotype property get or property set Corresponds To: The name of the variable behind this property.
overload	Operation Corresponds To: The 'overload' keyword.
override	Operation

	Corresponds To: The 'override' keyword.
packed	Class Corresponds To: The 'packed' keyword.
property	Class Corresponds To: A property; see *Delphi Properties* for more information.
reintroduce	Operation Corresponds To: The 'reintroduce' keyword.

Other Conventions

- The Static property of an attribute or operation corresponds to the 'class' keyword
- The Fixed property of a parameter corresponds to the 'const' keyword
- The value of inout for the Kind property of a parameter corresponds to the 'Var' keyword
- The value of out for the Kind property of a parameter corresponds to the 'Out' keyword

Java Conventions

Enterprise Architect supports round trip engineering of Java - including AspectJ extensions - where these conventions are used.

Stereotypes

Stereotype	Applies To
annotation	Interface Corresponds To: An annotation type.
CompactConstructor	Operation Corresponds to: A compact canonical constructor for the record.
record	Class Corresponds To: A record type.
default	Operation Corresponds To: The 'default' keyword.
enum	Attributes within a Class stereotyped enumeration Corresponds To: An enumerated option, distinguished from other attributes that

	have no stereotype.
enumeration	Class Corresponds To: An enumerated type.
operator	Operation Corresponds To: An operator.
property get	Operation Corresponds To: A 'read' property.
property set	Operation Corresponds To: A 'write' property.
static	Class or Interface Corresponds To: The 'static' keyword.

Tagged Values

Tag	Applies To
annotations	Anything Corresponds To: The annotations on the current code feature.

arguments	Attribute with stereotype enum Corresponds To: The arguments that apply to this enumerated value.
attribute_name	Operation with stereotype property get or property set Corresponds To: The name of the variable behind this property.
dynamic	Class or Interface Corresponds To: The 'dynamic' keyword.
generic	Operation Corresponds To: The generic parameters to this operation.
nonsealed	Class, Interface Corresponds To: The 'non-sealed' keyword.
permits	Class, Interface Corresponds To: The 'permits' expression.
parameterList	Parameter Corresponds To: A parameter list with the ... syntax.

sealed	Class, Interface Corresponds To: The 'sealed' keyword.
RecordHeader	<<record>>Class Corresponds To: The record header of the record definition.
throws	Operation Corresponds To: The exceptions that are thrown by this method.
transient	Attribute Corresponds To: The 'transient' keyword.

Other Conventions

- Package statements are generated when the current Package is not a namespace root
- The Const property of an attribute or operation corresponds to the final keyword
- The Transient property of an attribute corresponds to the volatile keyword
- The Fixed property of a parameter corresponds to the final keyword

AspectJ Conventions

These are the conventions used for supporting AspectJ extensions to Java.

Stereotypes

Stereotype	Applies To
advice	Operation Corresponds To: A piece of advice in an AspectJ aspect.
aspect	Class Corresponds To: An AspectJ aspect.
pointcut	Operation Corresponds To: A 'pointcut' in an AspectJ aspect.

Tagged Values

Tag	Applies To
className	Attribute or operation within a Class

	stereotyped aspect Corresponds To: The Classes this AspectJ intertype member belongs to.

Other Conventions

- The specifications of a pointcut are included in the 'Behavior' field of the method

PHP Conventions

Enterprise Architect supports the round trip engineering of PHP 4 and 5, where these conventions are used.

Stereotypes

Stereotype	Applies To
trait	Class Corresponds To: A 'trait'.
property get	Operation Corresponds To: A 'read' property.
property set	Operation Corresponds To: A 'write' property.

Tagged Values

Tag	Applies To
attribute_name	Operation with stereotype property get or property set Corresponds To: The name of the

	variable behind this property.
final	Operations in PHP 5 Corresponds To: The 'final' keyword.

Common Conventions

- An unspecified type is modeled as var
- Methods returning a reference are generated by setting the Return Type to var*
- Reference parameters are generated from parameters with the parameter Kind set to inout or out

PHP 5 Conventions

- The final Class modifier corresponds to the Is Leaf property
- The abstract Class modifier corresponds to the Abstract property
- Parameter type hinting is supported by setting the Type of a parameter
- The value of inout or out for the Kind property of a parameter corresponds to a reference parameter

Python Conventions

Enterprise Architect supports the round trip engineering of Python, where these conventions are used.

Stereotypes

Stereotype	Corresponds to
TypeAlias	Class Corresponds To: Explicit Type Alias

Tagged Values

Tag	Applies To
async	Operation Corresponds To: The 'async' keyword in function definition.
Decorators	Class, Operation Corresponds To: The decorators applied to this element in the source.

Other Conventions

- Model members with Private Scope correspond to code members with two leading underscores
- Attributes are only generated when the Initial value is not empty
- All types are reverse engineered as var

SystemC Conventions

Enterprise Architect supports round-trip engineering of SystemC, where these conventions are used.

Stereotypes

Stereotype	Applies To
delegate	Method Corresponds To: A delegate.
enumeration	Inner Class Corresponds To: An enum type.
friend	Method Corresponds To: A friend method.
property	Method Corresponds To: A property definition.
sc_ctor	Method Corresponds To: A SystemC constructor.
sc_module	Class Corresponds To: A SystemC module.

sc_port	Attribute Corresponds To: A Port.
sc_signal	Attribute Corresponds To: A signal.
struct	Inner Class Corresponds To: A struct or union.

Tagged Values

Tag	Applies To
kind	Attribute (Port) Corresponds To: Port kind (clocked, fifo, master, slave, resolved, vector).
mode	Attribute (Port) Corresponds To: Port mode (in, out, inout).
overrides	Method Corresponds To: The Inheritance list of a method declaration.

throw	Method Corresponds To: The exception specification of a method.

Other Conventions

- SystemC also inherits most of the stereotypes and Tagged Values of C++

SystemC Toolbox Pages

To model a SystemC design, drag these icons onto a diagram from the 'SystemC Constructs' page of the Diagram Toolbox.

Page	Icon
SystemC	Module Action: Defines a SystemC Module. An sc_module -stereotyped Class element.
SystemC Features	Port Action: Defines a SystemC Port. An sc_port- stereotyped attribute.

Access

Ribbon	Design > Diagram > Toolbox : > Specify 'SystemC Constructs' in the 'Find Toolbox Item' dialogs
Keyboard Shortcuts	Ctrl+Shift+3 : > Specify 'SystemC Constructs' in the 'Find Toolbox Item' dialog
Other	You can display or hide the Diagram Toolbox by clicking on the or icons at the left-hand end of the Caption Bar at the top of the Diagram View.

VB.NET Conventions

Enterprise Architect supports round-trip engineering of Visual Basic.NET, where these conventions are used. Earlier versions of Visual Basic are supported as a different language.

Stereotypes

Stereotype	Applies To
event	Operation Corresponds To: An event declaration.
import	Operation Corresponds To: An operation to be imported from another library.
module	Class Corresponds To: A module.
operator	Operation Corresponds To: An operator overload definition.
partial	Operation Corresponds To: The 'partial' keyword on

	an operation.
property	Operation Corresponds To: A property possibly containing both read and write code.

Tagged Values

Tag	Applies To
Alias	Operation with stereotype import Corresponds To: The alias for this imported operation.
attribute_name	Operation with stereotype property Corresponds To: The name of the variable behind this property.
Charset	Operation with stereotype import Corresponds To: The character set clause for this import - one of the values 'Ansi', 'Unicode' or 'Auto'.
delegate	Operation Corresponds To: The 'delegate' keyword.

enumTag	Operation with stereotype property Corresponds To: The datatype that this property is represented as.
Handles	Operation Corresponds To: The 'handles' clause on this operation.
Implements	Operation Corresponds To: The 'implements' clause on this operation.
Lib	Operation with stereotype import Corresponds To: The library this import comes from.
MustOverrid e	Operation Corresponds To: The 'MustOverride' keyword.
Narrowing	Operation with stereotype operator Corresponds To: The 'Narrowing' keyword.
NotOverridea ble	Operation Corresponds To: The 'NotOverrideable' keyword.

Overloads	Operation Corresponds To: The 'overloads' keyword.
Overrides	Operation Corresponds To: The 'overrides' keyword.
parameterArray	Parameter Corresponds To: A parameter list using the 'ParamArray' keyword.
partial	Class, Interface Corresponds To: The 'partial' keyword.
readonly	Operation with stereotype property Corresponds To: This property only defining 'read' code.
shadows	Class, Interface, Operation Corresponds To: The 'Shadows' keyword.
Shared	Attribute Corresponds To: The 'Shared' keyword.
Widening	Operation with stereotype operator Corresponds To: The 'Widening' keyword.

writeonly	Operation with stereotype property Corresponds To: This property only defining 'write' code.

Other Conventions

- Namespaces are generated for each Package below a namespace root
- The Is Leaf property of a Class corresponds to the NotInheritable keyword
- The Abstract property of a Class corresponds to the MustInherit keyword
- The Static property of an attribute or operation corresponds to the Shared keyword
- The Abstract property of an operation corresponds to the MustOverride keyword
- The value of in for the Kind property of a parameter corresponds to the ByVal keyword
- The value of inout or out for the Kind property of a parameter corresponds to the ByRef keyword

Verilog Conventions

Enterprise Architect supports round-trip engineering of Verilog, where these conventions are used.

Stereotypes

Stereotype	Applies To
asynchronous	Method Corresponds To: A concurrent process.
enumeration	Inner Class Corresponds To: An enum type.
initializer	Method Corresponds To: An initializer process.
module	Class Corresponds To: A module.
part	Attribute Corresponds To: A component instantiation.
port	Attribute Corresponds To: A Port.

synchronous	Method Corresponds To: A sequential process.

Tagged Values

Tag	Applies To
kind	Attribute (signal) Corresponds To: The signal kind (such as register, bus).
mode	Attribute (Port) Corresponds To: The Port mode (in, out, inout).
Portmap	Attribute (part) Corresponds To: The generic/Port map of the component instantiated.
sensitivity	Method Corresponds To: The sensitivity list of a sequential process.
type	Attribute Corresponds To: The range or type value

	of an attribute.

Verilog Toolbox Pages

Access: 'Design > Diagram > Toolbox : 'Hamburger' icon > HDL | Verilog Constructs'

Drag these icons onto a diagram to model a Verilog design.

Item	Action
Module	Defines a Verilog Module. A module-stereotyped Class element.
Enumeration	Defines an Enumerated Type. An enumeration element.
Port	Defines a Verilog Port. A port-stereotyped attribute.
Part	Defines a Verilog component instantiation. A part-stereotyped attribute.
Attribute	Defines an attribute.
Procedure	Defines a Verilog process: • Concurrent - An asynchronous-stereotyped method • Sequential - A

	synchronous-stereotyped method • Initializer - An initializer-stereotyped method

VHDL Conventions

Enterprise Architect supports round-trip engineering of VHDL, where these conventions are used.

Stereotypes

Stereotype	Applies To
architecture	Class Corresponds To: An architecture.
asynchronous	Method Corresponds To: An asynchronous process.
configuration	Method Corresponds To: A configuration.
enumeration	Inner Class Corresponds To: An enumerated type.
entity	Interface Corresponds To: An entity.
part	Attribute Corresponds To: A component

	instantiation.
port	Attribute Corresponds To: A Port.
signal	Attribute Corresponds To: A signal declaration.
struct	Inner Class Corresponds To: A record definition.
synchronous	Method Corresponds To: A synchronous process.
typedef	Inner Class Corresponds To: A type or subtype definition.

Tagged Values

Tag	Applies To
isGeneric	Attribute (port) Corresponds To: The 'port' declaration in a generic interface.

isSubType	Inner Class (typedef) Corresponds To: A subtype definition.
kind	Attribute (signal) Corresponds To: The signal kind (such as 'register', 'bus').
mode	Attribute (Port) Corresponds To: The Port mode ('in', 'out', 'inout', 'buffer', 'linkage').
portmap	Attribute (part) Corresponds To: The generic/Port map of the component instantiated.
sensitivity	Method (synchronous) Corresponds To: The 'sensitivity' list of a synchronous process.
type	Inner Class (typedef) Corresponds To: The 'type' indication of a 'type' declaration.
typeNameSpace	Attribute (part) Corresponds To: The 'type' namespace of the instantiated component.

VHDL Toolbox Pages

Access

To model a VHDL design, drag icons from the VHDL toolbox pages and drop them on your diagram.

Ribbon	Design > Diagram > Toolbox : > Specify 'VHDL Constructs' in the 'Find Toolbox Item' dialog
Keyboard Shortcuts	Ctrl+Shift+3 : > Specify 'VHDL Constructs' in the 'Find Toolbox Item' dialog
Other	You can display or hide the Diagram Toolbox by clicking on the or icons at the left-hand end of the Caption Bar at the top of the Diagram View.

VHDL Toolbox Page

Item	Action

Architecture	Defines an architecture to be associated with a VHDL entity. An architecture-stereotyped Class element.
Entity	Defines a VHDL entity to contain the Port definitions. An entity-stereotyped interface element.
Enumeration	Defines an Enumerated Type. An Enumeration element.
Struct	Defines a VHDL record. A struct-stereotyped Class element.
Typedef	Defines a VHDL type or subtype. A typedef-stereotyped Class element.

VHDL Features Toolbox Page

Item	Action
Part	Defines a VHDL component instantiation. A part-stereotyped attribute.

Port	Defines a VHDL Port. A port-stereotyped attribute.
Signal	Defines a VHDL signal. A signal-stereotyped attribute.
Procedure	Defines a VHDL process: • Concurrent - An asynchronous-stereotyped method • Sequential - A synchronous-stereotyped method • Configuration - An configuration-stereotyped method

Visual Basic Conventions

Enterprise Architect supports the round trip engineering of Visual Basic 5 and 6, where these conventions are used.

Visual Basic .NET is supported as a different language.

Stereotypes

Stereotype	Applies To
global	Attribute Corresponds To: The 'Global' keyword.
import	Operation Corresponds To: An operation to be imported from another library.
property get	Operation Corresponds To: A property 'get'.
property set	Operation Corresponds To: A property 'set'.
property let	Operation Corresponds To: A property 'let'.
with events	Attribute

	Corresponds To: The 'WithEvents' keyword.

Tagged Values

Tag	Applies To
Alias	Operation with stereotype import Corresponds To: The alias for this imported operation.
attribute_name	Operation with stereotype property get, property set or property let Corresponds To: The name of the variable behind this property.
Lib	Operation with stereotype import Corresponds To: The library this import comes from.
New	Attribute Corresponds To: The 'new' keyword.

Other Conventions

- The value of in for the Kind property of a parameter corresponds to the ByVal keyword
- The value of inout or out for the Kind property of a parameter corresponds to the ByRef keyword

Language Options

You can set up various options for how Enterprise Architect handles a particular language when generating and reverse-engineering code. These options are either specific to:

- Your user ID, for all models or
- The model in which they are defined, for all users

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > <language name> Settings > Model > Options > Source Code Engineering > <language name>
Keyboard Shortcuts	Ctrl+F9 ('Preferences' dialog)

Languages Supported

Language

Action Script
Ada 2012 (in the Unified and Ultimate Editions of Enterprise Architect)
ArcGIS
ANSI C
C#
C++
Delphi
Java
PHP
Python
SystemC
Verilog (Unified and Ultimate Editions)
VHDL (Unified and Ultimate Editions)
Visual Basic

Visual Basic .NET

ActionScript Options - User

If you intend to generate ActionScript code from your model, you can configure the code generation options using the 'ActionScript Specifications' page of the 'Preferences' dialog to:

- Specify the default source directory
- Specify the editor for ActionScript code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > ActionScript
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support ActionScript code generation. Select this checkbox to disable

	ActionScript code support.
Options for the current user	In the 'Default Source Directory' and 'Editor' fields, click on the ... button and browse for the source directory and external file editor that you will use.

Notes

- These options apply to all models that you access

ActionScript Options - Model

If you intend to generate ActionScript code from your model, you can configure the model-specific code generation options using the 'ActionScript Specifications' page of the 'Manage Model Options' dialog to:

- Specify default ActionScript version to generate (AS2.0 or AS3.0)
- Specify default file extensions
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > ActionScript

Options

Option	Action
Options for the current model	Type in the default ActionScript version and default file extension to apply when generating ActionScript source code.

Collection Classes	Click on this button to open the 'Collection Classes for Association Roles' dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Ada 2012 Options - User

If you intend to generate Ada 2012 code from your model, you can configure the code generation options using the 'Ada' page of the 'Preferences' dialog to:

- Inform the reverse engineering process whether the name of the Tagged Record is the same as the Package name
- Advise the engine of the alternative Tagged Record name to locate
- Specify whether the engine should create a reference type for the Tagged Record (if one is not defined)
- Supply the name of the reference type to be created (default is Ref)
- Specify the reference parameter of a Reference / Access type
- Tell the engine to ignore the name of the reference parameter
- Indicate the name of the reference parameter to locate

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Ada
Keyboard	Ctrl+F9

Shortcuts	

Options

Option	Action
Disable Language	Leave this checkbox unselected to support Ada 2012 code generation. Select this checkbox to disable Ada 2012 code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

Notes

- Ada 2012 support is available in the Unified and Ultimate Editions of Enterprise Architect

Ada 2012 Options - Model

If you intend to generate Ada 2012 code from your model, you can configure the model-specific code generation options using the 'Ada' page of the 'Manage Model Options' dialog to:

- Specify the default file extension and
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Ada

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Ada source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models
- Ada 2012 support is available in the Unified and Ultimate Editions of Enterprise Architect

ArcGIS Options - User

If you intend to generate ArcGIS code from your model, you can configure the code generation options using the 'ArcGIS' page of the 'Preferences' dialog to:

- Specify the default source directory
- Specify the editor for ArcGIS code

ArcGIS must be enabled in the 'MDG Technologies' dialog ('Specialize > Technologies > Manage Technology') in order for the 'ArcGIS' page to be available.

Access

Ribbon	Start >Appearance > Preferences > Preferences > Source Code Engineering > ArcGIS
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable	Leave this checkbox unselected to

Language	support ArcGIS code generation. Select this checkbox to disable ArcGIS code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

ArcGIS Options - Model

If you intend to generate ArcGIS code from your model, you can configure the model-specific code generation options using the 'ArcGIS' page of the 'Manage Model Options' dialog to:

- Specify default file extensions
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > ArcGIS

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating ArcGIS source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

C Options - User

If you intend to generate C code from your model, you can configure the code generation options using the 'C Specifications' page of the 'Preferences' dialog.

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > C
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support C code generation. Select this option to disable C code support.
Options for	In the value fields, specify the options

the current user	that apply under your own user ID in all models that you access: • The default attribute type to create (fixed as int) • Whether a #define constant is imported as an attribute in imported C code (if 'Object Oriented programming' is set to True on the 'C Specifications' page of the 'Manage Model Options' dialog) • Whether to generate comments for C methods to the declaration, and to reverse engineer comments from the declaration • Whether to generate comments for C methods to the implementation, and to reverse engineer comments from the implementation • Whether to update comments in regenerating code from the model • Whether to update the implementation file in re-generating code from the model • The default source code directory location (click on the ... button) • The default file extensions to read when importing a directory of C code • The Code Editor to use (click on the ... button)

	• The search path for the implementation file relative to the header file path

C Options - Model

If you intend to generate C code from your model, you can configure the model-specific code generation options using the 'C Specifications' page of the 'Manage Model Options' dialog to:

- Specify default file extensions (header and source)
- Define support for Object Oriented programming
- Set the StateMachine engineering options
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > C

Options

Option	Action
Options for the current model	In the value fields, specify these options: • The default header and source file extensions for the code files

	• Support for Object Oriented programming; if this is True, then set: - The Namespace delimiter character - Whether the first parameter of an operation is a Class reference - The parameter reference style in generated C code - The reference parameter name in generated code - The default Constructor name in generated code - The default Destructor name in generated code
StateMachine Engineering	In the value fields, use the drop-down arrows to set the options to True or False; these options apply to generating code from StateMachine models in the current model only: • 'Use the new StateMachine Template' - set to True to use the code generation templates from Enterprise Architect Release 11 and later, set to False to apply the EASL Legacy templates • Generate Trace Code - set to True to generate Trace code, False to omit it
Collection	Click on this button to open the

Classes	'Collection Classes for Association Roles' dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

C# Options - User

If you intend to generate C# code from your model, you can configure the code generation options using the 'C# Specifications' page of the 'Preferences' dialog

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > C#
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support C# code generation. Select this checkbox to disable C# code support.
Options for	In the value fields, specify the options

the current user	that apply under your own user ID in all models that you access: • The default attribute type to create • Whether Namespaces should be generated when generating C# Classes • Whether to remove new lines (hard carriage returns) from the summary tag when importing XML.NET style comments • Whether to generate a Finalizer method when generating code for a C# Class • Whether to generate a Dispose method when generating code for a C# Class • The default source code directory location (click on the ... button) • The Code Editor to use (click on the ... button)

C# Options - Model

If you intend to generate C# code from your model, you can configure the model-specific code generation options using the 'C# Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Indicate additional Collection Classes - to define custom Collection Classes, which can be simple substitutions (such as CArray<#TYPE#>) or a mix of other strings and substitutions (such as Cmap<CString,LPCTSTR,#TYPE#*,#TYPE#*>); these Collection Classes are defined by default:
  - List<#TYPE#>;Stack<#TYPE#>;Queue<#TYPE#>;
- Set the StateMachine Engineering options
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > C#

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating C# source code, and a list of any additional Collection Classes you want to define.
StateMachine Engineering	In the value fields, use the drop-down arrows to set the options to True or False; these options apply to generating code from StateMachine models in the current model only: • 'Use the new StateMachine Template' - set to True to use the code generation templates from Enterprise Architect Release 11 and later, set to False to apply the EASL Legacy templates • 'Generate Trace Code' - set to True to generate Trace code, False to omit it
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles' dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

C++ Options - User

If you intend to generate C++ code from your model, you can configure the code generation options using the 'C++ Specifications' page of the 'Preferences' dialog.

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > C++
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support C++ code generation. Select this option to disable C++ code support.
Options for	In the value fields, specify the options

the current user	that apply under your own user ID in all models that you access: • The default attribute type to create • Whether Namespaces should be generated when generating C++ Classes • What style to apply when generating and processing comments for C++ • Whether to generate comments for C++ methods to the declaration, or reverse engineer comments from the declaration • Whether to generate comments for C++ methods to the implementation, or reverse engineer comments from the implementation • Whether to update comments in re-generating code from the model • Whether to update the implementation file in re-generating code from the model • The default source code directory location (click on the ... button) • The default file extensions to read when importing a directory of C++ code • The Code Editor to use (click on the ... button)

	• The search path for the implementation file relative to the header file path

C++ Options - Model

If you intend to generate C++ code from your model, you can configure the model-specific code generation options using the 'C++ Specifications' page of the 'Manage Model Options' dialog to:

- Indicate the version of C++ to generate; this controls the set of templates used and how properties are created
- Specify the default reference type used when a type is specified by reference
- Specify the default file extensions
- Specify default Get/Set prefixes
- Specify the Collection Class definitions for Association connectors
- Define additional Collection Classes - to define custom Collection Classes, which can be simple substitutions (such as CArray<#TYPE#>) or a mix of other strings and substitutions (such as Cmap<CString,LPCTSTR,#TYPE#*,#TYPE#*>); these Collection Classes are defined by default:
  - CArray<#TYPE#>;CMap<CString,LPCTSTR,#TYPE#*, #TYPE#*>;
- Set the StateMachine Engineering options

Access

Ribbon	Settings > Model > Options > Source Code Engineering > C++

Options

Option	Action
Options for the current model	In the value fields, specify the options that affect all users of the current model: • The version of C++ you are using (which determines which templates to use when generating code) • The default reference type to use when creating properties for C++ attributes by reference • The default header and source file extensions for the code files • The default 'Get' prefix • The default 'Set' prefix • The additional Collection Classes
StateMachine Engineering Options	In the value fields, use the drop-down arrows to set the options to True or False; these options apply to generating code from StateMachine models in the current

	model only: • 'Use the new StateMachine Template' - set to True to use the code generation templates from Enterprise Architect Release 11 and later, set to False to apply the EASL Legacy templates • 'Generate Trace Code' - set to True to generate Trace code, False to omit it
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles' dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Delphi Options - User

If you intend to generate Delphi code from your model, you can configure the code generation options using the 'Delphi Specifications' page of the 'Preferences' dialog to:

- Set the default attribute type
- Indicate a default source directory
- Set the default code editor to use to edit Delphi source code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Delphi
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable	Leave this checkbox unselected to

Language	support Delphi code generation. Select this option to disable Delphi code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

Delphi Options - Model

If you intend to generate Delphi code from your model, you can configure the model-specific code generation options using the 'Delphi Specifications' page of the 'Manage Model Options' dialog to:

- Specify default file extensions (header and source)
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Delphi

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Delphi source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Delphi Properties

Enterprise Architect has comprehensive support for Delphi properties. These are implemented as Tagged Values, with a specialized property editor to help create and modify Class properties. By using the 'Feature Visibility' element context menu option, you can display the 'tags' compartment that contains the properties. Imported Delphi Classes with properties have this feature automatically made visible for your convenience.

Manually activate the property editor

- In the selected Class set the code generation language to 'Delphi'
- Right-click on the Class and select 'Delphi Properties' to open the editor

Using the Delphi Properties editor, you can build properties quickly and simply; from here you can:

- Change the name and scope (only Public and Published are currently supported)
- Change the property type (the drop-down list includes all defined Classes in the project)
- Set the Read and Write information (the drop-down lists have all the attributes and operations from the current Class; you can also enter free text)
- Set 'Stored' to True or False
- Set the Implements information

- Set the default value, if one exists

Notes

- When you use the 'Create Property' dialog from the 'Attribute' screen, the system generates a pair of Get and Set functions together with the required property definition as Tagged Values; you can manually edit these Tagged Values if required
- Public properties are displayed with a '+' symbol prefix and published with a '^'
- When creating a property in the 'Create Property Implementation' dialog (accessed through the 'Attributes' dialog), you can set the scope to 'Published' if the property type is Delphi
- Only 'Public' and 'Published' are supported
- If you change the name of a property and forward engineer, a new property is added, but you must manually delete the old one from the source file

Java Options - User

If you intend to generate Java code from your model, you can configure the code generation options using the 'Java Specifications' page of the 'Preferences' dialog.

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Java
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support Java code generation. Select this checkbox to disable Java code support.
Options for	In the value fields, specify the options

the current user	that apply under your own user ID in all models that you access; the: • Default attribute type to create (select from the drop-down list) • Default source code directory location (click on the ... button) • Code Editor to use (click on the ... button)

Java Options - Model

If you intend to generate Java code from your model, you can configure the model-specific code generation options using the 'Java Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify a default 'Get' prefix
- Specify a default 'Set' prefix
- Set the StateMachine Engineering options
- Specify the Collection Class definitions for Association connectors
- Define additional Collection Classes - to define custom Collection Classes, which can be simple substitutions (such as CArray<#TYPE#>) or a mix of other strings and substitutions (such as Cmap<CString,LPCTSTR,#TYPE#*,#TYPE#*>); these Collection Classes are defined by default:
    - HashSet<#TYPE#>;Map<String,#TYPE#>;

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Java

Options

Option	Action
Options for the current model	In the value fields, specify the options that affect all users of the current model; the: • Default file extension for the code files • The default Get and Set prefixes • The default and additional Collection Classes
StateMachine Engineering	In the value fields, use the drop-down arrows to set the options to True or False; these options apply to generating code from StateMachine models in the current model only: • 'Use the new StateMachine Template' - set to True to use the code generation templates from Enterprise Architect Release 11 and later, set to False to apply the EASL Legacy templates • 'Generate Trace Code' - set to True to generate Trace code, False to omit it
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles' dialog, through which you specify the

	Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

MySQL Options - User

If you intend to generate MySQL code from your model, you can configure the code generation options using the 'MySQL' page of the 'Preferences' dialog to:

- Specify a default attribute type
- Specify a default source directory
- Specify file name extensions for files to import
- Specify an editor for changing code
- Specify a default owner

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > MySQL
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action

Disable Language	Leave this checkbox unselected to support MySQL code generation. Select this option to disable MySQL code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

MySQL Options - Model

If you intend to generate MySQL code from your model, you can configure the model-specific code generation options using the 'MySQL' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > MySQL

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating MySQL source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

PHP Options - User

If you intend to generate PHP code from your model, you can configure the code generation options using the 'PHP Specifications' page of the 'Preferences' dialog to:

- Define a semi-colon separated list of extensions to look at when doing a directory code import for PHP
- Set a default directory for opening and saving PHP source code
- Specify the default editor to use when editing PHP code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > PHP
Keyboard Shortcuts	Ctrl+F9 | Source Code Engineering | PHP

Options

Option	Action
Disable	Leave this checkbox unselected to

Language	support PHP code generation. Select this option to disable PHP code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

PHP Options - Model

If you intend to generate PHP code from your model, you can configure the model-specific code generation options using the 'PHP Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default PHP version to generate
- Define the default file extension
- Specify a default 'Get' prefix
- Specify a default 'Set' prefix

Access

Ribbon	Settings > Model > Options > Source Code Engineering > PHP

Options

Option	Action
Options for the current model	Type in the default PHP version, the default file extension to apply when generating PHP source code, and the default 'Get' and 'Set' prefixes.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Python Options - User

If you intend to generate Python code from your model, you can configure the code generation options using the 'Python Specifications' page of the 'Preferences' dialog to:

- Specify the default source directory to be used
- Specify the default editor used to write and edit Python code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Python
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support Python code generation. Select this option to disable Python code

	support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

Python Options - Model

If you intend to generate Python code from your model, you can configure the model-specific code generation options using the 'Python Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default file extension

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Python

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Python source code.

Notes

- These options affect all users of the current model;

however, they do not apply to other models

SystemC Options - User

If you intend to generate SystemC code from your model, you can configure the code generation options using the 'SystemC' page of the 'Preferences' dialog to:

- Specify a default source directory
- Specify an editor for changing code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > SystemC
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support SystemC code generation. Select this option to disable SystemC

	code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

SystemC Options - Model

If you intend to generate SystemC code from your model, you can configure the model-specific code generation options using the 'SystemC' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > SystemC

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating SystemC source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Teradata Options - User

If you intend to generate Teradata code from your model, you can configure the code generation options using the 'Teradata' page of the 'Preferences' dialog to:

- Specify a default attribute type
- Specify a default source directory
- Specify an editor for changing code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Teradata
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support Teradata code generation.

	Select this option to disable Teradata code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

Teradata Options - Model

If you intend to generate Teradata code from your model, you can configure the model-specific code generation options using the 'Teradata' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Teradata

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Teradata source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

VB.NET Options - User

If you intend to generate VB.NET code from your model, you can configure the code generation options using the 'VB.NET Specifications' page of the 'Preferences' dialog to:

- Specify the default attribute type
- Indicate whether to generate namespaces
- Specify a default source directory
- Specify an editor for changing code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > VB.Net
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable	Leave this checkbox unselected to

Language	support VB.NET code generation. Select this option to disable VB.NET code support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

VB.NET Options - Model

If you intend to generate VB.NET code from your model, you can configure the model-specific code generation options using the 'VB.Net Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > VB.Net

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating VB.Net source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Verilog Options - User

If you intend to generate Verilog code from your model, you can configure the code generation options using the 'Verilog' page of the 'Preferences' dialog to:

- Specify a default source directory
- Specify an editor for changing code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Verilog
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support Verilog code generation. Select this option to disable Verilog code

	support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

Verilog Options - Model

If you intend to generate Verilog code from your model, you can configure the model-specific code generation options using the 'Verilog' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Verilog

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Verilog source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

VHDL Options - User

If you intend to generate VHDL code from your model, you can configure the code generation options using the 'VHDL' page of the 'Preferences' dialog to:

- Specify a default source directory
- Specify an editor for changing code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > VHDL
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action
Disable Language	Leave this checkbox unselected to support VHDL code generation. Select this option to disable VHDL code

	support.
Options for the current user	Specifies the options used for the current user; these options apply to all models that are accessed by the user.

VHDL Options - Model

If you intend to generate VHDL code from your model, you can configure the model-specific code generation options using the 'VHDL' page of the 'Manage Model Options' dialog to:

- Specify the default file extension
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > VHDL

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating VHDL source code.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles'

	dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

Visual Basic Options - User

If you intend to generate Visual Basic code from your model, you can configure the code generation options using the 'VB Specifications' page of the 'Preferences' dialog to:

- Specify the default attribute type
- Define the default source directory
- Define the file extensions to search for code files to import
- Define the default editor to use for editing source code

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > Visual Basic
Keyboard Shortcuts	Ctrl+F9

Options

Option	Action

Disable Language	Leave this checkbox unselected to support Visual Basic code generation. Select this option to disable Visual Basic code support.
Options for the current user	Specifies the options used for the current use; these options apply to all models that are accessed by the user.

Visual Basic Options - Model

If you intend to generate Visual Basic code from your model, you can configure the model-specific code generation options using the 'VB Specifications' page of the 'Manage Model Options' dialog to:

- Specify the default Visual Basic version to generate
- Indicate the default file extension when reading/writing
- Indicate the Microsoft Transaction Server (MTS) transaction mode for MTS objects
- Specify if a Class uses Multi use (True or False)
- Specify if a Class uses the Persistable property
- Indicate data binding and data source behaviors
- Set the global namespace
- Set the Exposed attribute
- Indicate if the Creatable attribute is True or False
- Specify the Collection Class definitions for Association connectors

Access

Ribbon	Settings > Model > Options > Source Code Engineering > Visual Basic

Options

Option	Action
Options for the current model	Type in the default file extension to apply when generating Visual Basic source code, and click on the drop-down arrow in each of the other fields and select the appropriate value.
Collection Classes	Click on this button to open the 'Collection Classes for Association Roles' dialog, through which you specify the Collection Class definitions for Association connectors.

Notes

- These options affect all users of the current model; however, they do not apply to other models

MDG Technology Language Options

If you have loaded an MDG Technology that specifies a code module into your *Sparx Systems > EA > MDG Technologies* folder, the language is included in the 'Source Code Engineering' list on the 'Preferences' dialog. The language is only listed on the 'Preferences' dialog if an MDG Technology file actually uses it in your model.

Access

Ribbon	Start > Appearance > Preferences > Preferences > Source Code Engineering > MDG
Keyboard Shortcuts	Ctrl+F9

Options

Field	Action
Default Extension	Default extension for generated source files; shown if the option is in the technology.

	This is saved per project.
Import File Extensions	Default folder to import source files from; shown if the technology supports namespaces. This is saved once for all projects.
Generate Namespaces	Indicates if namespaces are generated or not.
Default Source Directory	The default directory to save generated source files. This is always shown.
Editor	Indicates the editor that is used to edit source files.
Att Type	Indicates the default attribute type.

Notes

- These options are set in the technology inside the <CodeOptions> tag of a code module, as shown:
  <CodeOption name="DefaultExtension">.rb</CodeOption>

Reset Options

Enterprise Architect stores some of the options for a Class when it is first created. Some are global; for example, $LinkClass is stored when you first create the Class, so in existing Classes the global change in the 'Preferences' dialog will not automatically be picked up. You must modify the options for the existing Class.

Modify options for a single Class

Step	Action
1	Click on the Class to change, and select the 'Develop > Source Code > Generate > Generate Single Element' ribbon option. The 'Generate Code' dialog displays.
2	Click on the Advanced button. The 'Object Options' dialog displays.
3	Click on the 'Attributes/Operations' option.
4	Change the options, and click on the Close button to apply the changes.

Modify options for all Classes within a Package

Step	Action
1	Click on the Package in the Browser window, and select the 'Develop > Preferences > Options > Reset Source Language' ribbon option. The 'Manage Code Generation' dialog displays.
2	In the 'Where language is:' field, click on the drop-down arrow and select the language that you want to change from.
3	In the 'Convert to:' field, click on the drop-down arrow and select the language that you want to change to.
4	Select the checkbox against each option to apply to the changed Class elements in the Package: • Clear Filenames of the files to generate code to • Reset Default options on each Class • Process Child Packages under the selected Package
5	Click on the OK button to apply the changes.

Set Collection Classes

Using Enterprise Architect, you can define Collection Classes for generating code from Association connectors where the target role has a multiplicity setting greater than 1.

Tasks

Task	Detail
Defining Collection Classes	On the 'Source Code Engineering' section of the 'Manage Model Options' dialog (select the 'Settings > Model > Options > Source Code Engineering' ribbon option), on each language page click on the Collection Classes button. The 'Collection Classes for Association Roles' dialog displays. On this dialog, you can define: • The default Collection Class for 1..* roles • The ordered Collection Class to use for 1..* roles • The qualified Collection Class to use for 1..* roles
Defining	Class-specific Collection Classes can be

Collection Classes for a specific Class	defined by clicking the Collection Classes button in the Class 'Properties' dialog of the element.
Code Generation Precedence	When Enterprise Architect generates code for a connector that has a multiplicity role >1: 1. If the Qualifier is set, use the qualified collection: - for the Class if set - else use the code language qualified collection 2. If the 'Order' option is set, use the ordered collection: - for the Class if set - else use the code language ordered collection 3. Else use the default collection: - for the Class if set - else use the code language default collection
Using Markers	You can include the marker #TYPE# in the collection name; Enterprise Architect replaces this with the name of the Class being collected at source generation time (for example, Vector<#TYPE#> would become Vector<foo>).

	Conversely, when reverse engineering, an Association connector is also created if a matching entry (for example, foo if foo is found in the model) is defined as a Collection Class.
Additional Collection Classes	Additional Collection Classes can be defined within the model-specific language options pages for C#, C++ and Java.
Member Type	On the 'Role(s)' tab of the Association 'Properties' dialog (accessible from the right-click context menu of any Association) there is a 'Member Type' field for each of the Source and Target Roles. If you set this, the value you enter overrides all the listed options.

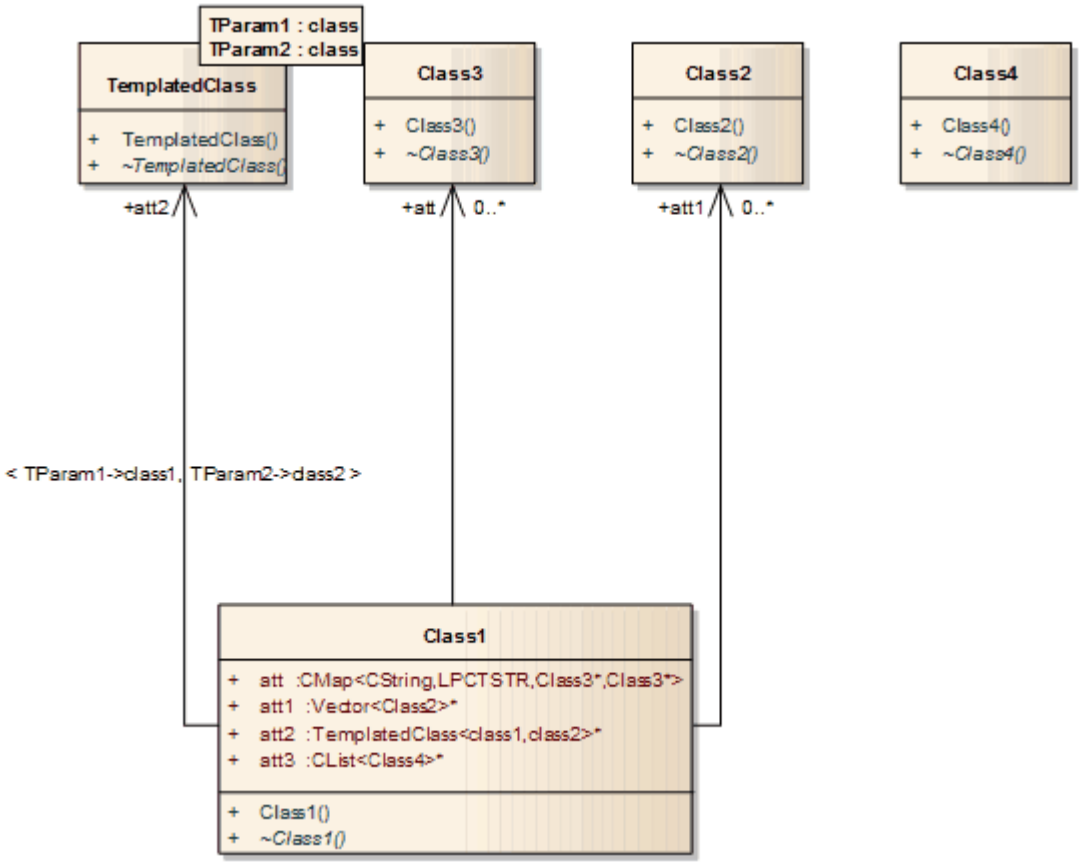
Example Use of Collection Classes

Consider this source code:

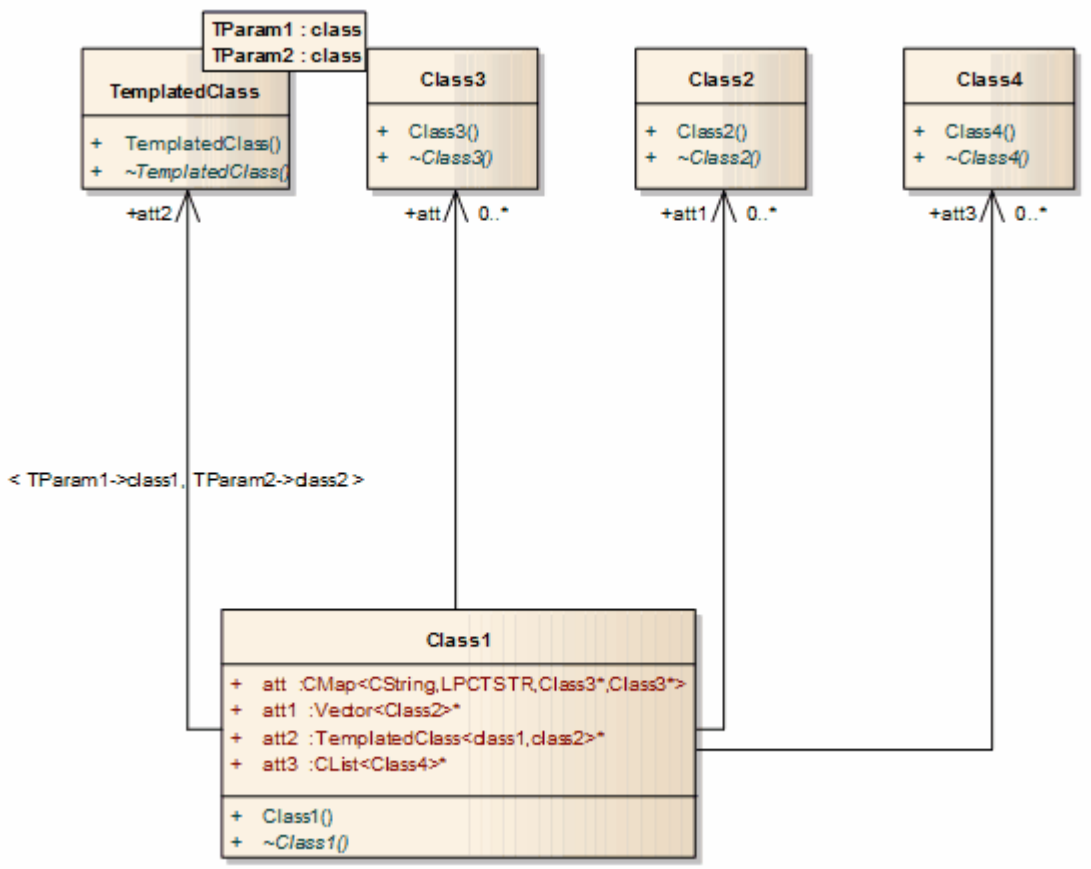
```
class Class1
{
public:
    Class1();
    virtual ~Class1();
    CMap<CString,LPCTSTR,Class3*,Class3*> att;
    Vector<Class2> *att1;
    TemplatedClass<class1,class2> *att2;
    CList<Class4> *att3;
};
class Class2
{
public:
    Class2();
    virtual ~Class2();
};
class Class3
{
public:
    Class3();
    virtual ~Class3();
```

```
};
class Class4
{
public:
    Class4();
    virtual ~Class4();
};
template<class TParam1, class TParam2>
class TemplatedClass
{
public:
    TemplatedClass() {
    }
    virtual ~TemplatedClass() {
    }
};
```

If this code is imported into the system with default import options, this diagram is generated:



If, however, you enter the value 'CList<#Type#>' in the 'Additional Collection Classes' field in the model-specific language options page (C#, Java, C++), an Association connector is also created to Class 4:


TParam1 : class
TParam2 : class
TemplatedClass
+ TemplatedClass()
+ ~TemplatedClass()
Class3
+ Class3()
+ ~Class3()
Class2
+ Class2()
+ ~Class2()
Class4
+ Class4()
+ ~Class4()
+att2
+att
0..*
+att1
0..*
+att3
0..*
< TParam1->class1, TParam2->class2 >
Class1
+ att :CMap<CString,LPCTSTR,Class3*,Class3*>
+ att1 :Vector<Class2>*
+ att2 :TemplatedClass<class1,class2>*
+ att3 :CList<Class4>*
+ Class1()
+ ~Class1()

Local Paths

When a team of developers are working on the same Enterprise Architect model, each developer might store their version of the source code in their local file system, but not always at the same location as their fellow developers. To manage this scenario in Enterprise Architect, you can define local paths for each user, on the 'Local Paths' dialog.

You can use local paths in generating code and reverse engineering, and in Version Control, developing XML schemas and generating document and web reports.

Local paths might take a little time to set up, but if you want to work collaboratively on source and model concurrently, the effort is well worth while.

For example, if:

- Developer A stores her .java files in a C:\Java\Source directory, while developer B stores his in D:\Source, and
- Both developers want to generate and reverse engineer into the same Enterprise Architect model located on a shared (or replicated) network drive

Developer A might define a local path of:

JAVA_SOURCE = "C:\Java\Source"

All Classes generated and stored in the Enterprise Architect project are stored as:

%JAVA_SOURCE%\<xxx.java>

Developer B defines a local path as:

JAVA_SOURCE ="D:\Source"

Now, Enterprise Architect stores all java files in these directories as:

%JAVA_SOURCE%\<filename>

On each developer's machine, the filename is expanded to the correct local version.

Access

Ribbon	Develop > Source Code > Options > Configure Local Paths

Local Paths Dialog

Using the 'Local Paths' dialog, you can set up local paths for a single user on a particular machine. For a description of the use of local paths, see the *Local Paths* topic.

Access

Ribbon	Develop > Source Code > Options > Configure Local Paths

Options

Option	Action
Path	Type in or browser for the path of the local directory in the file system (for example, d:\java\source).
ID	Type in the shared ID that is substituted for the Local Path (for example, JAVA_SRC).
Type	Click on the drop-down arrow and select the type of path to apply to (for example,

	Java).
Relative Paths	Lists the paths currently defined for the model, defaulting to most recent at the top. If you want to change the sequence of paths in the list, click on a path and use the buttons to move the path up or down one position in the list.
Apply Path	Click on a path in the 'Relative Paths' list and click on this button to update any existing full path names in the model to the shared relative path name. For example: d:\java\source\main.java might become %JAVA_SRC%\main.java
Expand Path	Click on a path in the 'Relative Paths' list and click on this button to remove the relative path and substitute the full path name (the opposite effect of the Apply Path button).
New	Click on this button to clear the data fields so that you can define another local path.
Save	When you have defined a local path, click

	on this button to save it and add it to the 'Relative Paths' list.
Delete	Click on a path in the 'Relative Paths' list and click on this button to remove the path from the list altogether.
Close	Click on this button to close the dialog, saving any changes to the list.

Notes

- You can also set up a hyperlink (for an Enterprise Architect command) on a diagram to access the 'Local Paths' dialog, to switch, update or expand your current local path
- If the act of expanding or applying a path for a linked file will create a duplicate record, the process will skip that record and display a message at the end of the process

Language Macros

When reverse engineering a language such as C++, you might find preprocessor directives scattered throughout the code. This can make code management easier, but can hamper parsing of the underlying C++ language.

To help remedy this, you can include any number of macro definitions, which are ignored during the parsing phase of the reverse engineering. It is still preferable, if you have the facility, to preprocess the code using the appropriate compiler first; this way, complex macro definitions and defines are expanded out and can be readily parsed. If you don't have this facility, then this option provides a convenient substitute.

Access

Ribbon	Settings > Reference Data > Settings > Preprocessor Macros or Develop > Source Code > Options > Configure > Define Preprocessor Macros

Define a macro

Ste	Action

p	
1	Select the 'Preprocessor Macros' menu option. The 'Language Macros' dialog displays.
2	Click on the Add New button.
3	Enter details for your macro.
4	Click on the OK button.

Macros Embedded Within Declarations

Macros are sometimes used within the declaration of Classes and operations, as in these examples:

```
class __declspec Foo
{
int __declspec Bar(int p);
};
```

If declspec is defined as a C++ macro, as outlined, the imported Class and operation contain a Tagged Value called DeclMacro1 with value __declspec (subsequent macros would be defined as DeclMacro2, DeclMacro3 and so on).

During forward engineering, these Tagged Values are used to regenerate the macros in code.

Define Complex Macros

It is sometimes useful to define rules for complex macros that can span multiple lines; Enterprise Architect ignores the entire code section defined by the rule.

Such macros can be defined in Enterprise Architect as in these two examples; both types can be combined in one definition.

Block Macros

BEGIN_INTERFACE_PART ^
END_INTERFACE_PART

The ^ symbol represents the body of the macro - this enables skipping from one macro to another; the spaces surrounding the ^ symbol are required.

Function Macros

RTTI_EMULATION()

Enterprise Architect skips over the token including everything inside the parentheses.

Function Macros can also include the function body:

RTTI_EMULATION() {}

In this case, Enterprise Architect skips over the token including everything inside the parentheses and inside the braces. Note that if the Function Macro includes the function body, it cannot be combined with a Block Macro.

Notes

- You can transport these language macro (or preprocessor macro) definitions between models, using the 'Settings > Model > Transfer > Export Reference Data' and 'Import Reference Data' options; the macros are exported as a Macro List

Developing Programming Languages

You can make use of a range of established programming languages in Enterprise Architect, but if these are not suitable to your needs you can develop your own. You would then apply it to your models through an MDG Technology that you might develop just for this purpose, or for broader purposes. After developing the language, you could also write MDA Transformation templates to convert a Platform Independent Model or a model in another language into a model for your new language, or vice-versa.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates
Keyboard Shortcuts	Ctrl+Shift+P

Develop a Programming Language

Step	Description

1	In the Code Template Editor, click on the New Language button and, on the 'Programming Languages Datatypes' dialog, click on the Add Product button. Enter your new programming language name and define the datatypes for it. You cannot access the new language in the Code Template Editor until at least one datatype has been added to the language.
2	After you have defined all the datatypes you need, click on the Close button, select the language in the 'Language' field of the Code Template Editor, and start to edit or create the code templates for the new language. The code templates define how the system should perform: • Forward code engineering of your models in the new language • Behavioral Code generation (if this is appropriate)
3	If you prefer, you can also define source code options for your new language. These are additional settings for the language that are not provided by the data types or code templates, and that help define how the system handles that language when generating and reverse-engineering code. The code options are made available to your models only through an MDG Technology.

4	Defining a grammar for your language is an optional step that provides two primary benefits: • Reverse engineering of existing code into your model • Synchronization during code generation so that changes made to the file since it was last generated are not lost. To access the grammar editor select the 'Develop > Source Code > Grammar Editor' ribbon option.
5	If you intend MDA transformations to be made to (or from) your new programming language, you can also edit and create transformation templates for it. The process of creating transformation templates is very similar to that for creating code templates.
6	Having created the datatypes, code templates, code options, grammar and transformation templates for your new language, you can incorporate and distribute them in an MDG Technology.

Code Template Framework

When you use Enterprise Architect to generate code from a model, or transform the model, the system refers to the Code Template Framework (CTF) for the parameters that define how it should:

- Forward engineer a UML model
- Generate Behavioral Code
- Perform a Model Driven Architecture (MDA) Transformation
- Generate DDL in database modeling

A range of standard templates is available for the direct generation of code and for transformation; if you do not want to use the standard CTF configurations, you can customize them to meet your needs.

CTF Templates

Template Type	Detail
Code Templates	When you forward engineer a Class model, the code templates define how the skeletal code is to be generated for a given programming language. The templates for a language are automatically associated with the

	language. The templates are written as plain text with a syntax that shares some aspects of both mark-up languages and scripting languages.
Model Transformation Templates	Model Transformation Templates provide a fully configurable method of defining how Model Driven Architecture (MDA) Transformations convert model elements and model fragments from one domain to another. This process is two-tiered. It creates an intermediary language (which can be viewed for debugging) which is then processed to create the objects.
Behavioral Code Generation Templates	Enterprise Architect supports user-definable code generation of the UML Behavioral models. This applies the standard Code Template Framework but includes specific Enterprise Architect Simulation Library (EASL) code generation macros.
DDL Templates	DDL Templates are very similar to Code generation templates, but they have been extended to support DDL generation with their own set of base templates, macros,

	function macros and template options.

Code Template Customization

Enterprise Architect helps you to generate source code from UML models for a wide range of programming languages. Standard templates (mappings) are provided out-of-the-box but you can customize the way that code is generated by using the practical and flexible Code Template Framework (CTF). This sophisticated framework allows you to customize every detail of the way code is generated, including the facility to create new templates for languages not supported in the base product. For example, JavaScript is not one of the supported languages but a series of templates can be written quickly to generate JavaScript from UML models. In these cases existing templates act as a useful starting point and reference for new languages.

The code template framework also provides the mechanism for generation of behavioral models and is used for the transformation templates.

Features

Feature	Detail
Default Templates	Default Code Templates are built into Enterprise Architect for forward engineering supported languages.
Code	A Code Template Editor is provided for

Template Editor	creating and maintaining user-defined Code Templates.
Customizing Code Templates	Descriptions of the template syntax and the macros and functions you can use to control the effects of the templates.
Synchronize Code	A subset of the default Code Templates to synchronize code.

Code and Transform Templates

Code templates and transform (Model Transformation) templates define how the system should generate or transform code in one or other of the programming languages that Enterprise Architect supports. Each language has a wide range of base templates, each of which defines how a particular code structure is generated. You can use these base templates as they are, or you can customize and add to the templates to better support your use of the standard languages, or of other languages that you might define to the system. You review, update and create templates through the Code Template editor or Transformation Template editor.

The order in which the base templates are listed in the two editors relates to the hierarchical order of the objects and their parts that are to be processed. Calls are made from certain base templates to others, and you can add further calls to both base templates and to your own custom templates. By default, the File template is the starting point of a code generation process through the templates; a File consists of Classes that can contain Attributes and Operations.

Access

Ribbon	Develop >Source Code > Options > Edit

	Code Templates Design > Package > Transform > Transform Templates
Keyboard Shortcuts	Ctrl+Shift+P (Code Generation Templates) Ctrl+Alt+H (MDA Transformation Templates)

Application of Templates

Action	Detail
Calling Templates	Within any template, you can call other templates using %TemplateName%. The enclosing percent (%) signs indicate a macro. You would use this for a single call to the ClassBody template, %ClassBody%, as shown: % list = "TemplateName" @separator= "\n" @indent= " " % The %list macro performs an iterative pass on all the objects in the scope of the current template and calls the TemplateName for each of them:

	% list = "ClassBody" @separator= "\n" @indent= " " %
	After generation or transformation, each macro is substituted to produce the generated output; for a language such as C++, the result of processing this template might be:
	/**
	* This is an example Class note generated using code templates
	* @author Sparx Systems
	*/
	class ClassA: public ClassB
	{
	...
	}

Execution of Code Templates	Each template might act only on a particular element type; for example, the ClassNotes template only acts on UML Class and Interface elements. The element from which code is currently being generated is said to be in scope; if the element in scope is stereotyped, the system searches for a template that has been defined for that stereotype. If a specialized template is found, it is executed; otherwise the default

	implementation of the base template is used. Templates are processed sequentially, line by line, replacing each macro with its underlying text value from the model.
Transfer Templates Between Projects	If you edit a base Code Generation or Transformation template, or create a customized template, you can copy them from one project to another as Reference Data.

Base Templates

The Code Template Framework consists of a number of base templates. Each base template transforms particular aspects of the UML to corresponding parts of object-oriented languages.

The base templates form a hierarchy, which varies slightly across different programming languages. In a typical template hierarchy relevant to a language such as C# or Java (which do not have header files) the templates can be modeled as Classes, but usually are just plain text. This hierarchy would be slightly more complicated for languages such as C++ and Delphi, which have separate implementation templates.

Each of the base templates must be specialized to be of use in code engineering; in particular, each template is specialized for the supported languages (or 'products'). For example, there is a ClassBody template defined for C++, another for C#, another for Java, and so on; by specializing the templates, you can tailor the code generated for the corresponding UML entity.

Once the base templates are specialized for a given language, they can be further specialized based on:

- A Class's stereotype, or
- A feature's stereotype (where the feature can be an operation or attribute)

This type of specialization enables, for example, a C# operation that is stereotyped as «property» to have a

different Operation Body template from an ordinary operation; the Operation Body template can then be specialized further, based on the Class stereotype.

Base templates used in the CTF

Template	Description
Attribute	A top-level template to generate member variables from UML attributes.
Attribute Declaration	Used by the Attribute template to generate a member variable declaration.
Attribute Notes	Used by the Attribute template to generate member variable notes.
Class	A top-level template for generating Classes from UML Classes.
Class Base	Used by the Class template to generate a base Class name in the inheritance list of a derived Class, where the base Class doesn't exist in the model.
Class Body	Used by the Class template to generate the body of a Class.

Class Declaration	Used by the Class template to generate the declaration of a Class.
Class Interface	Used by the Class template to generate an interface name in the inheritance list of a derived Class, where the interface doesn't exist in the model.
Class Notes	Used by the Class template to generate the Class notes.
File	A top-level template for generating the source file. For languages such as C++, this corresponds to the header file.
Import Section	Used in the File template to generate external dependencies.
Linked Attribute	A top-level template for generating attributes derived from UML Associations.
Linked Attribute Notes	Used by the Linked Attribute template to generate the attribute notes.
Linked Attribute	Used by the Linked Attribute template to generate the attribute declaration.

Declaration	
Linked Class Base	Used by the Class template to generate a base Class name in the inheritance list of a derived Class, for a Class element in the model that is a parent of the current Class.
Linked Class Interface	Used by the Class template to generate an Interface name in the inheritance list of a derived Class, for an Interface element in the model that is a parent of the current Class.
Namespace	A top-level template for generating namespaces from UML Packages (although not all languages have namespaces, this template can be used to generate an equivalent construct, such as Packages in Java).
Namespace Body	Used by the Namespace template to generate the body of a namespace.
Namespace Declaration	Used by the Namespace template to generate the namespace declaration.
Operation	A top-level template for generating operations from a UML Class's

	operations.
Operation Body	Used by the Operation template to generate the body of a UML operation.
Operation Declaration	Used by the Operation template to generate the operation declaration.
Operation Notes	Used by the Operation template to generate documentation for an operation.
Parameter	Used by the Operation Declaration template to generate parameters.

Templates for generating code for languages with separate interface and implementation sections

Template	Description
Class Impl	A top-level template for generating the implementation of a Class.
Class Body Impl	Used by the Class Impl template to generate the implementation of Class members.

File Impl	A top-level template for generating the implementation file.
File Notes Impl	Used by the File Impl template to generate notes in the source file.
Import Section Impl	Used by the File Impl template to generate external dependencies.
Operation Impl	A top-level template for generating operations from a UML Class's operations.
Operation Body Impl	Used by the Operation template to generate the body of a UML operation.
Operation Declaration Impl	Used by the Operation template to generate the operation declaration.
Operation Notes Impl	Used by the Operation template to generate documentation for an operation.

Export Code Generation and Transformation Templates

It is possible to export Code Generation and Transformation templates from your model to a .xml file. You can then import that file - and hence the templates - into other models, as reference data. You can export customized templates, which includes those that you or other users have created and updated, and base (standard) templates that have been tailored. You do not need to export base templates that have not been changed, as these are available in every installation of Enterprise Architect.

Access

Ribbon	Settings > Model > Transfer > Export Reference Data

Export a Code Generation template or Transformation template

Step	Action

1	On the 'Export Reference Data' dialog, in the 'Name' list, select the templates to export. The list includes any standard Code Generation or Transformation templates that have been changed, and any customized templates that you have created or changed. You can select one or more templates to be exported to a single XML file, by pressing Ctrl or Shift as you click on the template names.
2	Click on the Export button.
3	When prompted to do so, enter a valid file name with a .xml extension.
4	Click on the Save button and on the OK button. This exports the template(s) to the file; you can use any text or XML viewer to examine the file.

Import Code Generation and Transformation Templates

If you have exported Code Generation and/or Transformation templates from an Enterprise Architect model, you can import them into other Enterprise Architect models as reference data.

Access

Ribbon	Settings > Model > Transfer > Import Reference Data

Import Code Generation and/or Transformation Templates

Step	Action
1	On the 'Import Reference Data' dialog, click on the Select File button and browse to the .xml file containing the required Code Generation or Transformation templates.

2	Select the name of one or more template datasets and click on the Import button.

Synchronize Code

Enterprise Architect uses code templates during the forward synchronization of these programming languages:

- ActionScript
- C
- C++
- C#
- Delphi
- Java
- PHP
- Python
- VB
- VB.Net

Three types of change can occur in the source when it is synchronized with the UML model:

- Existing sections are synchronized: for example, the return type in an operation declaration is updated
- New sections are added to existing features: for example, Notes are added to a Class declaration where there were previously none
- New features and elements are added: for example, a new operation is added to a Class

Each of these changes has a different effect on the CTF and must be handled differently by Enterprise Architect, as described in these topics:

- *Synchronize Existing Sections*
- *Add New Sections to Existing Features*
- *Add New Features and Elements*

Code sections that can be synchronized

Only a subset of the CTF base templates is used during synchronization. This subset corresponds to the distinct sections that Enterprise Architect recognizes in the source code.

Code Template	Code Section
Class Notes	Comments preceding the Class declaration.
Class Declaration	Up to and including the Class parents.
Attribute Notes	Comments preceding an Attribute declaration.
Attribute Declaration	Up to and including the terminating character.
Operation Notes	Comments preceding an operation declaration.
Operation	

Notes Impl	As for Operation Notes.
Operation Declaration	Up to and including the terminating character.
Operation Declaration Impl	Up to and including the terminating character.
Operation Body	Everything between and including the braces.
Operation Body Impl	As for Operation Body.

Synchronize Existing Sections

When an existing section in the source code differs from the result generated by the corresponding template, that section is replaced.

Consider, for example, this C++ Class declaration:

```
(asm) class A: public B
```

Now assume that you add an inheritance relationship from Class A to Class C; the entire Class declaration would be replaced with something resembling this:

```
(asm) class A: public B, public C
```

Add New Sections

These sections can be added to existing features in the source code, as new sections:

- Class Notes
- Attribute Notes
- Operation Notes
- Operation Notes Impl
- Operation Body
- Operation Body Impl

Assume that, in this example, Class A had no note when you originally generated the code:

(asm) class A: public B, public C

If you now specify a note in the model for Class A, Enterprise Architect attempts to add the new note from the model during synchronization, by executing the Class Notes template.

To make room for the new section to be inserted, you can specify how much white space to append to the section via synchronization macros.

Add New Features and Elements

These features and elements can be added to the source code during synchronization:

- Attributes
- Inner Classes
- Operations

They are added by executing the relevant templates for each new element or feature in the model.

Enterprise Architect attempts to preserve the appropriate indenting of new features in the code, by finding the indents specified in list macros of the Class; for languages that make use of namespaces, the 'synchNamespaceBodyIndent' macro is available.

Classes defined within a (non-global) namespace are indented according to the value set for this macro, during synchronization.

The value is ignored:

- For Classes defined within a Package set up as a root namespace, or
- If the 'Generate Namespaces' option is set to False in the appropriate language page (C#, C++ or VB.Net) on the 'Preferences' dialog ('Start > Appearance > Preferences > Preferences > Source Code Engineering > <language>')

The Code Template Editor

The Code Template Editor provides the facilities of the Common Code Editor, including Intelli-sense for the various macros. For more information on Intelli-sense and the Common Code Editor, see the *Editing Source Code* topic.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates
Keyboard Shortcuts	Ctrl+Shift+P

Options

Option	Action
Language	Select the programming language.
New Language	Display the 'Programming Languages Datatypes' dialog, which enables you to include programming languages other than those supported for Enterprise

	Architect, for which to create or edit code templates.
Template	Display the contents of the active template, and open the editor for modifying templates.
Templates	List the base code templates; the active template is highlighted. The 'Modified' field indicates whether you have changed the default template for the current language.
Stereotype Overrides	List the stereotyped templates, for the active base template. The 'Modified' field indicates whether you have modified a default stereotyped template.
Add New Custom Template	Invoke a dialog for creating a custom stereotyped template.
Add New Stereotyped Override	Invoke a dialog for adding a stereotyped template, for the currently selected base template.
Get Default Template	Update the editor display with the default version of the active template.

Save	Overwrite the active templates with the contents of the editor.
Delete	If you have overridden the active template, the override is deleted and replaced by the corresponding default code template.

Notes

- User-modified and user-defined Code Templates can be imported and exported as reference data (see the *Sharing Reference Data* topic); the templates defined for each language are indicated in the 'Export Reference Data' dialog by the language name with the suffix _Code_Templates - if no templates exist for a language, there is no entry for the language in the dialog

Create New Custom Template

The Create New Custom Template dialog provides the ability to create a custom template for the current Programming or Database Management System (DBMS) Language, depending on what information is being edited with the Code Template Editor.

When this dialog is loaded you will be prompted to enter a value for Template Type and Template Name. In order to save a new template both Type and Name are required.

Options

Option	Action
Template Type	Choose the type of Template for the new Custom Template.
Template Name	Enter a Name for the new Custom Template.
OK	Save the details of the new Custom Template.
Cancel	Close the Create New Custom Template dialog and loose any unsaved changes.

Note:

All templates of type "<none>" are treated as functions, therefore Enterprise Architect will automatically remove all space characters entered into the Name.

Code Template Syntax

Code Templates are written using Enterprise Architect's Code Template Editor. The Code Template Editor supports syntax highlighting of the Code Template Framework language.

Syntax Elements

Elements	Detail
Basic Constructs	Templates can contain: • Literal Text • Variables • Macros • Calls to other templates
Comments	If you want to add comments to the templates, use the command: $COMMENT="text" where "text" is the text of the comment; this must be enclosed in quotes. The command is case-sensitive, and must be typed in upper case.

Literal Text

All text within a given template that is not part of a macro or a variable definition/reference, is considered literal text. With the exception of blank lines, which are ignored, literal text is directly substituted from the template into the generated code.

Consider this excerpt from the Java Class Declaration template:

```
$bases = "Base"
class % className % $bases
```

On the final line, the word 'class ', including the subsequent space, would be treated as literal text and thus for a Class named 'foo' would return the output:

```
class fooBase
```

A blank line following the variable $bases would have no effect on the output.

Inserting System Characters:

The %, $, " and \ characters have special meaning in the template syntax and cannot always be used as literal text. If these characters must be generated from within the templates, they can be safely reproduced using these direct substitution macros:

Macro	Action
%dl%	Produce a literal $ character.

%pc%	Produce a literal % character.
%qt%	Produce a literal " character.
%sl%	Produce a literal \ character

Notes

String conjunction operators (“+”, “+=”) are not required but can be used

Variables

Template variables provide a convenient way of storing and retrieving data within a template. This section explains how variables are defined and referenced.

Variable Definitions

Variable definitions take the basic form:

$<name> = <value>

where <name> can be any alpha-numeric sequence and <value> is derived from a macro or another variable.

A simple example definition would be:

$foo = %className%

Variables can be defined using values from:

- Substitution, function or list macros
- String literals, enclosed within double quotation marks
- Variable references

Definition Rules

These rules apply to variable definitions:

- Variables have global scope within the template in which they are defined and are not accessible to other templates
- Each variable must be defined at the start of a line, without any intervening white space
- Variables are denoted by prefixing the name with $, as in

$foo

- Variables do not have to be declared, prior to being defined
- Variables must be defined using either the assignment operator (=), or the addition-assignment operator (+=)
- Multiple terms can be combined in a single definition using the addition operator (+)

Examples

Using a substitution macro:

```
$foo = %opTag:"bar"%
```

Using a literal string:

```
$foo = "bar"
```

Using another variable:

```
$foo = $bar
```

Using a list macro:

```
$ops = %list="Operation" @separator="\n\n" @indent="\t"%
```

Using the addition-assignment operator (+=):

```
$body += %list="Operation" @separator="\n\n" @indent="\t"%
```

That definition is equivalent to:

```
$body = $body + %list="Operation" @separator="\n\n" @indent="\t"%
```

Using multiple terms:

$templateArgs = %list="ClassParameter" @separator=", "%

$template ="template<" + $templateArgs + ">"

Variable References

Variable values can be retrieved by using a reference of the form:

$<name>

where <name> can be a previously defined variable.

Variable references can be used:

- As part of a macro, such as the argument to a function macro
- As a term in a variable definition
- As a direct substitution of the variable value into the output

It is legal to reference a variable before it is defined. In this case, the variable is assumed to contain an empty string value: ""

Variable References - Example 1

Using variables as part of a macro. This is an excerpt from the default C++ ClassNotes template.

$wrapLen = %genOptWrapComment%

$style = %genOptCPPCommentStyle% (Define variables to store the style and wrap length options)

%if $style == "XML.NET"% (Reference to $style as part of a condition)

%XML_COMMENT($wrapLen)%

%else%

%CSTYLE_COMMENT($wrapLen)% (Reference to $wrapLen as an argument to function macro)

%endIf%

Variable References - Example 2

Using variable references as part of a variable definition.

$foo = "foo" (Define our variables)

$bar = "bar"

$foobar = $foo + $bar ($foobar now contains the value foobar)

Variable References - Example 3

Substituting variable values into the output.

$bases=%classInherits% (Store the result of the ClassInherits template in $bases)

Class %className%$bases (Now output the value of $bases after the Class name)

Macros

Macros provide access to element fields within the UML model and are also used to structure the generated output. All macros are enclosed within percent (%) signs, as shown:

%<macroname>%

In general, macros (including the % delimiters) are substituted for literal text in the output. For example, consider this item from the Class Declaration template:

... class %className% ...

The field substitution macro, %className%, would result in the current Class name being substituted in the output. So if the Class being generated was named Foo, the output would be:

... class Foo ...

The CTF contains a number of types of macro:

- Template Substitution Macros
- Field Substitution Macros
- Substitution Examples
- Attribute Field Substitution Macros
- Class Field Substitution Macros
- Code Generation Option Field Substitution Macros
- Connector Field Substitution Macros
- Constraint Field Substitution Macros
- Effort Field Substitution Macros
- File Field Substitution Macros

- File Import Field Substitution Macros
- Link Field Substitution Macros
- Linked File Field Substitution Macros
- Metric Field Substitution Macros
- Operation Field Substitution Macros
- Package Field Substitution Macros
- Parameter Field Substitution Macros
- Problem Field Substitution Macros
- Requirement Field Substitution Macros
- Resource Field Substitution Macros
- Risk Field Substitution Macros
- Scenario Field Substitution Macros
- Tagged Value Substitution Macros
- Template Parameter Substitution Macros
- Test Field Substitution Macros
- Function Macros
- Control Macros
- List Macro
- Branching Macros
- Synchronization Macros
- The Processing Instruction (PI) Macro
- EASL Code Generation Macros

Template Substitution Macros

Template substitution macros correspond to Base templates, and result in the execution of the named template. By convention, template macros are named according to Pascal casing.

Structure: %<TemplateName>%

where <TemplateName> can be one of the templates listed in this topic.

When a template is referenced from within another template, it is generated with respect to the elements currently in scope. The specific template is selected based on the stereotypes of the elements in scope.

As noted previously, there is an implicit hierarchy among the various templates. Some care should be taken in order to preserve a sensible hierarchy of template references. For example, it does not make sense to use the %ClassInherits% macro within any of the Attribute or Operation templates. Conversely, the Operation and Attribute templates are designed for use within the ClassBody template.

Template substitution macros in the CTF

- Attribute
- AttributeDeclaration
- AttributeDeclarationImpl
- AttributeNotes
- Class

- ClassBase
- ClassBody
- ClassBodyImpl
- ClassDeclaration
- ClassDeclarationImpl
- ClassImpl
- ClassInherits
- ClassInterface
- ClassNotes
- ClassParameter
- File
- FileImpl
- ImportSection
- ImportSectionImpl
- InnerClass
- InnerClassImpl
- LinkedAttribute
- LinkedAttributeDeclaration
- LinkedAttributeNotes
- LinkedClassBase
- LinkedClassInterface
- Namespace
- NamespaceBody
- NamespaceDeclaration
- NamespaceImpl

- Operation
- OperationBody
- OperationBodyImpl
- OperationDeclaration
- OperationDeclarationImpl
- OperationImpl
- OperationNotes
- Parameter

Field Substitution Macros

The field substitution macros provide access to data in your model. In particular, they are used to access data fields from:

- Packages
- Classes
- Attributes
- Operations, and
- Parameters

Field substitution macros are named according to Camel casing. By convention, the macro is prefixed with an abbreviated form of the corresponding model element. For example, attribute-related macros begin with att, as in the %attName% macro, to access the name of the attribute in scope.

Macros that represent checkboxes return a value of T if the box is selected. Otherwise the value is empty.

This table lists a small number of project field substitution macros. Type-specific macros are listed in the subtopics of this *Field Substitution Macros* section.

Project Macros

Macro Name	Description
eaDateTime	The current time with format:

	DD-MMM-YYYY HH:MM:SS AM/PM.
eaGUID	A unique GUID for this generation.
eaVersion	Program Version (located in the 'About Enterprise Architect' dialog by selecting 'Start > Help > Help > About EA').

Substitution Examples

Field substitution macros can be used in one of two ways:

- Direct Substitution or
- Conditional Substitution

Direct Substitution

This form directly substitutes the corresponding value of the element in scope into the output.

Structure: %<macroName>%

Where <macroName> can be any of the macros listed in the Field Substitution Macros tables.

Examples

- %className%
- %opName%
- %attName%

Conditional Substitution

This form of the macro enables alternative substitutions to be made depending on the macro's value.

Structure: %<macroName> (== "<text>") ? <subTrue> (: <subFalse>) %

Where:

- () denotes that values between the parentheses are optional
- <text> is a string representing a possible value for the macro
- <subTrue> and <subFalse> can be a combination of quoted strings and the keyword value; where the value is used, it is replaced with the macro's value in the output

Examples

- %classAbstract=="T" ? "pure" :""%
- %opStereotype=="operator" ? "operator" :""%
- %paramDefault != "" ? " = " value : ""%

These three examples output nothing if the condition fails. In this case the False condition can be omitted, resulting in this usage:

- %classAbstract=="T" ? "pure"%
- %opStereotype=="operator" ? "operator"%
- %paramDefault != "" ? " = "value%

The third example of both blocks shows a comparison checking for a non-empty value or existence. This test can also be omitted.

- %paramDefault ? " = " value : ""%
- %paramDefault ? " = " value%

All of these examples containing paramDefault are equivalent. If the parameter in scope had a default value of 10, the output from each of them would normally be:

= 10

Notes

- In a conditional substitution macro, any white space following <macroName> is ignored; if white space is required in the output, it should be included within the quoted substitution strings

Attribute Field Substitution Macros

This table lists each of the attribute field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Attribute Macros

Macro Name	Description
attAlias	'Attributes' dialog: Alias.
attAllowDuplicates	'Attributes Detail' dialog: 'Allow Duplicates' checkbox.
attClassifierGUID	The unique GUID for the classifier of the current attribute.
attCollection	'Attributes Detail' dialog: 'Attribute is a Collection' checkbox.
attConst	'Attributes' dialog: 'Const' checkbox.
attContainerType	'Attributes Detail' dialog: Container Type.

attContainment	'Attributes' dialog: Containment.
attDerived	'Attributes' dialog: 'Derived' checkbox.
attGUID	The unique GUID for the current attribute.
attInitial	'Attributes' dialog: Initial.
attIsEnumLiteral	'Attributes' dialog: 'Is Literal' checkbox.
attIsID	'Attributes Detail' dialog: 'isID' checkbox.
attLength	'Column' dialog: Length.
attLowerBound	'Attributes Detail' dialog: Lower Bound.
attName	'Attributes' dialog: Name.
attNotes	'Attributes' dialog: Notes.
attOrderedMultiplicity	'Attributes Detail' dialog: 'Ordered Multiplicity' checkbox.
attProperty	'Attributes' dialog: 'Property' checkbox.

attQualType	The attribute type qualified by the namespace path (if generating namespaces) and the classifier path (dot delimited). If the attribute classifier has not been set, is equivalent to the attType macro.
attScope	'Attributes' dialog: Scope.
attStatic	'Attributes' dialog: 'Static' checkbox.
attStereotype	'Attributes' dialog: Stereotype.
attType	'Attributes' dialog: Type.
attUpperBound	'Attributes Detail' dialog: Upper Bound.
attVolatile	'Attributes Detail' dialog: 'Transient' checkbox.

Class Field Substitution Macros

This table provides a list of methods for accessing each available Class property in the Code Generation and Transformation templates.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Class Macros

Macro Name	Description
elemType	The element type: Interface or Class.
classAbstract	Class 'Properties' dialog: 'Abstract' checkbox ('Details' tab).
classAlias	Class 'Properties' dialog: 'Alias' field.
classArguments	Class 'Detail' dialog: C++ Templates: Arguments.
classAuthor	Class 'Properties' dialog: 'Author' field.
classBaseName	'Type Hierarchy' dialog: Class Name (for use where no connector exists between child and base Classes).

classBaseScope	The scope of the inheritance as reverse engineered. (For use where no connector exists between child and base Classes.)
classBaseVirtual	The virtual property of the inheritance as reverse engineered. (For use where no connector exists between child and base Classes.)
classComplexity	Class 'Properties' dialog: 'Complexity' field.
classCreated	The date and time the Class was created.
classGUID	The unique GUID for the current Class.
classHasConstructor	Looks at the list of methods in the current object and, depending on the conventions of the current language, returns T if one is a default constructor. Typically used with the genOptGenConstructor macro.
classHasCopyConstructor	Looks at the list of methods in the current object and, depending on the conventions of the current language, returns T if one is a copy constructor. Typically used with the genOptGenCopyConstructor macro.

classHasDestructor	Looks at the list of methods in the current object and, depending on the conventions of the current language, returns T if one is a destructor. Typically used with the genOptGenDestructor macro.
classHasParent	True, if the Class in scope has one or more base Classes.
classHasStereotype	True, if the Class in scope has a stereotype that matches a stereotype name (which you can optionally specify as fully qualified). It therefore checks all stereotypes that a Class has and returns 'T' if any of them is the specified stereotype or a specialization of it. For example: • %classHasStereotype:"block"% will return 'T' for any block-stereotyped Class from any SysML version, including associationBlock • %classHasStereotype:"SysML1.4::block"% will specifically match the SysML 1.4 versions Compare this with classStereotype, later.
classImports	'Code Gen' dialog: Imports.
classIsActive	Class 'Advanced' dialog: 'Is Active'

	checkbox.
classIsAssociationClass	True, if the Association is an AssociationClass connector.
classIsInstantiated	True, if the Class is an instantiated template Class.
classIsLeaf	Class 'Advanced' dialog: 'Is Leaf' checkbox.
classIsRoot	Class 'Advanced' dialog: 'Is Root' checkbox.
classIsSpecification	Class 'Advanced' dialog: 'Is Specification' checkbox.
classKeywords	Class 'Properties' dialog: 'Keywords' field.
classLanguage	Class 'Properties' dialog: 'Language' field.
classMacros	A space separated list of macros defined for the Class.
classModified	The date and time the Class was last modified.

classMultiplicity	Class 'Advanced' dialog: Multiplicity.
className	Class 'Properties' dialog: 'Name' field.
classNotes	Class 'Properties' dialog: 'Note' field.
classParamDefault	Class 'Detail' dialog.
classParamName	Class 'Detail' dialog.
classParamType	Class 'Detail' dialog.
classPersistence	Class 'Properties' dialog: 'Persistence' field ('Details' tab)
classPhase	Class 'Properties' dialog: 'Phase' field.
classQualName	The Class name prefixed by its outer Classes. Class names are separated by double colons (::).
classScope	Class 'Properties' dialog: 'Scope' field.
classStereoty	Class 'Properties' dialog: 'Stereotype' field. Retrieves the name of the first

pe	stereotype applied to the Class. When used in a comparison, it checks whether that first stereotype exactly matches a string. For example: %classStereotype=="enumeration" ? "enum" : "class"% Compare this with classHasStereotype, earlier.
classStatus	Class 'Properties' dialog: 'Status' field.
classVersion	Class 'Properties' dialog: 'Version' field.

Code Generation Option Field Substitution Macros

Code generation option field substitution macros operate on the source code generation options defined in the 'Source Code Engineering' pages of either the:

- 'Preferences' dialog ('Start > Appearance > Preferences > Preferences > Source Code Engineering') for user-specific options, or
- 'Manage Model Options' dialog ('Settings > Model > Options') for model-specific options

For more information on the division of the options, see the *Source Code Engineering Options* topic.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty. This table lists each of the code generation option field substitution macros.

Code Generation Option Macros

Macro Name	Description
genOptActionScriptVersion	ActionScript Specifications page: Default Version.

genOptCDefaultAttributeType	C Specifications page: Default Attribute Type.
genOptCGenMethodNotesInBody	C Specifications page: Method Notes In Implementation.
genOptCGenMethodNotesInHeader	C Specifications page: Method Notes In Header.
genOptCSynchNotes	C Specifications page: Synchronize Notes in Generation.
genOptCSynchCFile	C Specifications page: Synchronise Implementation file in Generation.
genOptCDefaultSourceDirectory	C Specifications page: Default Source Directory.
genOptCNamespaceDelimiter	C Specifications page: Namespace Delimiter.
genOptCOperationRefParam	C Specifications page: Reference as Operation Parameter.

genOptCOperationRefParamStyle	C Specifications page: Reference Parameter Style.
genOptCOperationRefParamName	C Specifications page: Reference Parameter Name.
genOptCConstructorName	C Specifications page: Default Constructor Name.
genOptCDestructorName	C Specifications page: Default Destructor Name.
genOptCPPCommentStyle	C++ Specifications page: Comment Style.
genOptCPPDefaultAttributeType	C++ Specifications page: Default Attribute Type.
genOptCPPDefaultReferenceType	C++ Specifications page: Default Reference Type.
genOptCPPDefaultSourceDirectory	C++ Specifications page: Default Source Directory.

genOptCPPGenMethodNotesInHeader	C++ Specifications page: 'Method Notes In Header' checkbox.
genOptCPPGenMethodNotesInBody	C++ Specifications page: Method Notes In Body checkbox.
genOptCPPGetPrefix	C++ Specifications page: Get Prefix.
genOptCPPHeaderExtension	C++ Specifications page: Header Extension.
genOptCPPSetPrefix	C++ Specifications page: Set Prefix.
genOptCPPSourceExtension	C++ Specifications page: Source Extension.
genOptCPPSynchNotes	C++ Specifications page: Synchronize Notes.
genOptCPPSynchCPPFile	C++ Specifications page: Synchronize CPP File.

genOptCSDefaultAttributeType	C# Specifications page: Default Attribute Type.
genOptCSSourceExtension	C# Specifications page: Default file extension.
genOptCSGenDispose	C# Specifications page: Generate Dispose.
genOptCSGenFinalizer	C# Specifications page: Generate Finalizer.
genOptCSGenNamespace	C# Specifications page: Generate Namespace.
genOptCSDefaultSourceDirectory	C# Specifications page: Default Source Directory.
genOptDefaultAssocAttName	Source Code Engineering page: Default name for associated attribute.
genOptDefaultConstructorScope	Object Lifetimes page: Default Constructor Visibility.

genOptDefaultCopyConstructorScope	Object Lifetimes page: Default Copy Constructor Visibility.
genOptDefaultDatabase	Code Editors page: Default Database.
genOptDefaultDestructorScope	Object Lifetimes page: Default Destructor Constructor Visibility.
genOptGenCapitalisedProperties	'Source Code Engineering' page: 'Capitalize Attribute Names for Properties' checkbox.
genOptGenComments	'Source Code Engineering' page: 'Comments - Generate' checkbox.
genOptGenConstructor	Object Lifetimes page: 'Generate Constructor' checkbox.
genOptGenConstructorInline	Object Lifetimes page: 'Constructor Inline' checkbox.
genOptGenCopyConstructor	Object Lifetimes page: 'Generate Copy Constructor' checkbox.

genOptGenCopyConstructorInline	Object Lifetimes page: 'Copy Constructor Inline' checkbox.
genOptGenDestructor	Object Lifetimes page: 'Generate Destructor' checkbox.
genOptGenDestructorInline	Object Lifetimes page: 'Destructor Inline' checkbox.
genOptGenDestructorVirtual	Object Lifetimes page: 'Virtual Destructor' checkbox.
genOptGenImplementedInterfaceOps	'Code Generation' page: 'Generate methods for implemented interfaces' checkbox.
genOptGenPrefixBoolProperties	'Source Code Engineering' page: 'Use 'Is' for Boolean property Get()' checkbox.
genOptGenRoleNames	'Source Code Engineering' page: 'Autogenerate role names when creating code' checkbox.
genOptGenUnspecAssocD	'Source Code Engineering' page: 'Do not generate members where Association

ir	direction is unspecified' checkbox.
genOptJavaDefaultAttributeType	Java Specifications page: Default attribute type.
genOptJavaGetPrefix	Java Specifications page: Get Prefix.
genOptJavaDefaultSourceDirectory	Java Specifications page: Default Source Directory.
genOptJavaSetPrefix	Java Specifications page: Set Prefix.
genOptJavaSourceExtension	Java Specifications page: Source code extension.
genOptPHPDefaultSourceDirectory	PHP Specifications page: Default Source Directory.
genOptPHPGetPrefix	PHP Specifications page: Get Prefix.
genOptPHPSetPrefix	PHP Specifications page: Set Prefix.

genOptPHPSourceExtension	PHP Specifications page: Default file extension.
genOptPHPVersion	PHP Specifications page: PHP Version.
genOptPropertyPrefix	'Source Code Engineering' page: Remove prefixes when generating Get/Set properties.
genOptVBMultiUse	VB Specifications page: 'Multiuse' checkbox.
genOptVBPersistable	VB Specifications page: 'Persistable' checkbox.
genOptVBDataBindingBehavior	VB Specifications page: 'Data binding behavior' checkbox.
genOptVBDataSourceBehavior	VB Specifications page: 'Data source behavior' checkbox.
genOptVBGlobal	VB Specifications page: 'Global namespace' checkbox.

genOptVBCreatable	VB Specifications page: 'Creatable' checkbox.
genOptVBExposed	VB Specifications page: 'Exposed' checkbox.
genOptVBMTS	VB Specifications page: MTS Transaction Mode.
genOptVBNetGenNamespace	VB.Net Specifications page: Generate Namespace.
genOptVBVersion	VB Specifications page: Default Version.
genOptWrapComment	'Source Code Engineering' page: Wrap length for comment lines.

Connector Field Substitution Macros

This table lists each of the connector field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Connector Macros

Macro Name	Description
connectorAlias	Connector 'Properties' dialog: 'Alias' field.
connectorAssociationClassElemGUID	The GUID of the connector's Association Class element.
connectorAssociationClassElemName	The name of the connector's Association Class element.
connectorDestAccess	Connector 'Properties' dialog, 'Target Role' tab: Access.
connectorDestAggregation	Connector 'Properties' dialog, 'Target Role' tab: Aggregation.

connectorDestAlias	Connector 'Properties' dialog, 'Target Role' tab: Alias.
connectorDestAllowDuplicates	Connector 'Properties' dialog, 'Target Role' tab: 'Allow Duplicates' checkbox.
connectorDestChangeable	Connector 'Properties' dialog, 'Target Role' tab: Changeable.
connectorDestConstraint	Connector 'Properties' dialog, 'Target Role' tab: Constraint(s).
connectorDestContainment	Connector 'Properties' dialog, 'Target Role' tab: Containment.
connectorDestDerived	Connector 'Properties' dialog, 'Target Role' tab: 'Derived' checkbox.
connectorDestDerivedUnion	Connector 'Properties' dialog, 'Target Role' tab: 'DerivedUnion' checkbox.
connectorDestElem*	A set of macros that access a property of the element at the target end of a connector. The * (asterisk) is a wildcard that corresponds to any Class substitution macro in the Class macro list. For

	example: • connectorDestElemAlias (classAlias) • connectorDestElemAuthor (classAuthor)
connectorDestElemType	The element type of the connector destination element. (Separate from the connectorDestElem* macros because there is no classType substitution macro.)
connectorDestFeature*	A set of macros that access a property of the feature at the target end of a connector. The * (asterisk) is a wildcard that corresponds to any attribute or operation substitution macro in the Attribute macro or Operation macro list, depending on the connectorDestFeatureType. For example: • connectorDestFeatureReturnClassifier GUID - an operation's return classifier GUID • connectorDestFeatureContainment - an attribute's containment
connectorDestFeatureType	The type of the connector destination feature. • connectorDestFeatureType="Attribute"

	or "Operation"
connectorDestMemberType	Connector 'Properties' dialog, 'Target Role' tab: Member Type.
connectorDestMultiplicity	Connector 'Properties' dialog, 'Target Role' tab: Multiplicity.
connectorDestNavigability	Connector 'Properties' dialog, 'Target Role' tab: Navigability.
connectorDestNotes	Connector 'Properties' dialog, 'Target Role' tab: Role Notes.
connectorDestOrdered	Connector 'Properties' dialog, 'Target Role' tab: 'Ordered' checkbox.
connectorDestOwned	Connector 'Properties' dialog, 'Target Role' tab: 'Owned' checkbox.
connectorDestQualifier	Connector 'Properties' dialog, 'Target Role' tab: Qualifier(s).
connectorDestRole	Connector 'Properties' dialog, 'Target Role' tab: Role.
connectorDestScope	Connector 'Properties' dialog, 'Target Role' tab: Target Scope.

connectorDestStereotype	Connector 'Properties' dialog, 'Target Role' tab: Stereotype.
connectorDirection	Connector Properties: Direction.
connectorEffect	'Transition Constraints' dialog: 'Effect' field.
connectorGuard	'Object Flow' and 'Transition Constraints' dialogs: 'Guard' field.
connectorGUID	The unique GUID for the current connector.
connectorIsAssociationClass	True, if the connector is an AssociationClass connector.
connectorName	Connector Properties: Name.
connectorNotes	Connector Properties: Notes.
connectorSourceAccess	Connector 'Properties' dialog, 'Source Role' tab: Access.

connectorSourceAggregation	Connector 'Properties' dialog, 'Source Role' tab: Aggregation.
connectorSourceAlias	Connector 'Properties' dialog, 'Source Role' tab: Alias.
connectorSourceAllowDuplicates	Connector 'Properties' dialog, 'Source Role' tab: Allow Duplicates checkbox.
connectorSourceChangeable	Connector 'Properties' dialog, 'Source Role' tab: Changeable.
connectorSourceConstraint	Connector 'Properties' dialog, 'Source Role' tab: Constraint(s).
connectorSourceContainment	Connector 'Properties' dialog, 'Source Role' tab: Containment.
connectorSourceDerived	Connector 'Properties' dialog, 'Source Role' tab: 'Derived' checkbox.
connectorSourceDerivedUnion	Connector 'Properties' dialog, 'Source Role' tab: 'DerivedUnion' checkbox.

connectorSourceElem*	A set of macros that access a property of the element at the source end of a connector. The * (asterisk) is a wildcard that corresponds to any Class substitution macro in the Class macro list. For example: • connectorSourceElemAlias (classAlias) • connectorSourceElemAuthor (classAuthor)
connectorSourceElemType	The element type of the connector source element. (Separate from the connectorSourceElem* macros because there is no classType substitution macro.)
connectorSourceFeature*	A set of macros that access a property of the feature at the source end of a connector. The * (asterisk) is a wildcard that corresponds to any attribute or operation substitution macro in the Attribute macro or Operation macro list, depending on the connectorSourceFeatureType. For example: • connectorSourceFeatureCode - Operation's Code • connectorSourceFeatureInitial - Attribute's Initial

connectorSourceFeatureType	The type of the connector source feature. • connectorSourceFeatureType="Attribute" or "Operation"
connectorSourceMemberType	Connector 'Properties' dialog, 'Source Role' tab: Member Type.
connectorSourceMultiplicity	Connector 'Properties' dialog, 'Source Role' tab: Multiplicity.
connectorSourceNavigability	Connector 'Properties' dialog, 'Source Role' tab: Navigability.
connectorSourceNotes	Connector 'Properties' dialog, 'Source Role' tab: Role Notes.
connectorSourceOrdered	Connector 'Properties' dialog, 'Source Role' tab: 'Ordered' checkbox.
connectorSourceOwned	Connector 'Properties' dialog, 'Source Role' tab: 'Owned' checkbox.
connectorSourceQualifier	Connector 'Properties' dialog, 'Source Role' tab: Qualifier(s).
connectorSou	Connector 'Properties' dialog, 'Source

rceRole	Role' tab: Role.
connectorSourceScope	Connector 'Properties' dialog, 'Source Role' tab: Target Scope.
connectorSourceStereotype	Connector 'Properties' dialog, 'Source Role' tab: Stereotype.
connectorStereotype	Connector 'Properties' dialog: 'Stereotype' field.
connectorTrigger	'Transition Constraints' dialog: 'Trigger' field.
connectorType	The connector type; f or example, Association or Generalization.
connectorWeight	'Object Flow Constraints' dialog: 'Weight' field.

Constraint Field Substitution Macros

This table lists each of the 'Constraint' field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Constraint Macros

Macro Name	Description
constraintName	'Class' dialog, 'Constraints' tab: Name.
constraintNotes	'Class' dialog, 'Constraints' tab: Notes.
constraintStatus	'Class' dialog, 'Constraints' tab: Status.
constraintType	'Class' dialog, 'Constraints' tab: Type.
constraintWeight	'Class' dialog, 'Constraints' tab: ordering (hand up/down) keys.

Effort Field Substitution Macros

This table lists each of the 'Effort' field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Effort Macros

Macro Name	Description
effortName	Effort window: Effort.
effortNotes	Effort window: Notes (unlabelled).
effortTime	Effort window: Time.
effortType	Effort window: Type.

File Field Substitution Macros

This table lists each of the file field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

File Macros

Macro Name	Description
fileExtension	The file type extension of the file being generated.
fileName	The name of the file being generated.
fileNameImpl	The filename of the implementation file for this generation, if applicable.
fileHeaders	'Code Gen' dialog: Headers.
fileImports	'Code Gen' dialog: Imports. For supported languages this also includes dependencies derived from these types of relationship: • Aggregation • Association • Attribute classifier

	• Method return type • Method parameter classifier • Generalization • Realization (to interface) • Template Binding (C++) • Dependency
filePath	The full path of the file being generated.
filePathImpl	The full path of the implementation file for this generation, if applicable.

File Import Field Substitution Macros

This table lists each of the file import field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of T if the box is selected. Otherwise the value is empty.

File Import Macros

Macro Name	Description
importClassName	The name of the Class being imported.
importFileName	The filename of the Class being imported.
importFilePath	The full path of the Class being imported.
importFromAggregation	T if the Class has an Aggregation connector to a Class in this file, F otherwise.
importFromAssociation	T if the Class has an Association connector to a Class in this file, F otherwise.

importFromAtt	T if an attribute of a Class in the current file is of the type of this Class, F otherwise.
importFromDependency	T if the Class has a Dependency connector to a Class in this file, F otherwise.
importFromGeneralization	T if the Class has a Generalization connector to a Class in this file, F otherwise.
importFromMeth	T if a method return type of a Class in the current file is the type of this Class, F otherwise.
importFromParam	T if a method parameter of a Class in the current file is of the type of this Class; otherwise F.
importFromPropertyType	T if the Class has a property (Part/Port) typing to another Class, F otherwise.
importFromRealization	T if the Class has a Realization connector to a Class in this file, F otherwise.
importFromTemplateBindi	T if the Class has a TemplateBinding connector to a Class in this file, F

ng	otherwise.
importInFile	T if the Class is in the current file, F otherwise.
importPackagePath	The Package path with a '.' separator of the Class being imported.
ImportRelativeFilePath	The relative file path of the Class being imported from the file path of the file being generated.

Link Field Substitution Macros

If you want to provide access to data concerning connectors in the model, particularly Associations and Generalizations, you can use the 'Link field substitution' macros. The macro names are in Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected; otherwise the value is empty.

Link Macros

Macro Name	Description/Result
linkAttAccess	Association 'Properties' dialog, Target Role: 'Access' field.
linkAttAggregation	Association 'Properties' dialog, Source or Target Role: Aggregation.
linkAttCollectionClass	The collection appropriate for the linked attribute in scope.
linkAttContainment	Association 'Properties' dialog, Target Role: Containment.
linkAttName	'Association Properties' dialog: Target.
linkAttNotes	Association 'Properties' dialog, Target

	Role: Role Notes.
linkAttOwnedByAssociation	True, if the 'Owned' checkbox on the 'Role(s)' page of the Association 'Properties' dialog is not selected.
linkAttOwnedByClass	True, if the 'Owned' checkbox on the 'Role(s)' page of the Association 'Properties' dialog is selected.
linkAttQualName	The Association target qualified by the namespace path (if generating namespaces) and the classifier path (dot delimited).
linkAttRole	Association 'Properties' dialog, Target Role: Role.
linkAttRoleAlias	'Association Properties Target Role' dialog: Alias
linkAttStereotype	Association 'Properties' dialog, Target Role: Stereotype.
linkAttTargetScope	Association 'Properties' dialog, Target Role: Target Scope.
linkCard	Link 'Properties' dialog, Target Role: Multiplicity.

linkGUID	The unique GUID for the current connector.
linkIsAssociationClass	True, if the Association is an AssociationClass connector.
linkIsBound	Returns T if any TemplateBindings are specified on the connector.
linkParamSubs	Returns a comma-separated list of the arguments specified.
linkParentName	Generalization 'Properties' dialog: 'Target' field.
linkParentQualName	The Generalization target qualified by the namespace path (if generating namespaces) and the classifier path (dot delimited).
linkStereotype	The stereotype of the current connector.
linkVirtualInheritance	Generalization 'Properties' dialog: 'Virtual Inheritance' field.

Linked File Field Substitution Macros

This table lists each of the 'Linked File' field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Linked File Macros

Macro Name	Description
linkedFileLastWrite	Class 'Properties' dialog: 'Files' tab, 'Last Write' field.
linkedFileNotes	Class 'Properties' dialog: 'Files' tab, 'Notes' field.
linkedFilePath	Class 'Properties' dialog: 'Files' tab, 'File Path' field.
linkedFileSize	Class 'Properties' dialog: 'Files' tab, 'Size' field.
linkedFileType	Class 'Properties' dialog: 'Files' tab, 'Type' field.

Metric Field Substitution Macros

This table lists each of the Metric field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Metric Macros

Macro Name	Description
metricName	Metrics screen: 'Metric' field.
metricNotes	Metrics screen: (Notes) field.
metricType	Metrics screen: 'Type' field.
metricWeight	Metrics screen: 'Weight' field.

Operation Field Substitution Macros

The 'Operation field substitution' macros provide access to data concerning operations in the model. The macro names are in Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected; otherwise the value is empty.

Operation field substitution macros

Macro Name	Description/Result
opAbstract	'Operation' dialog: 'Virtual' checkbox.
opAlias	'Operation' dialog: Alias.
opBehavior	'Operation Behavior' dialog: Behavior.
opCode	'Operation Behavior' dialog: Behavior Code.
opConcurrency	'Operation' dialog: Concurrency.
opConst	'Operation' dialog: 'Const' checkbox.
opGUID	The unique GUID for the current operation.

opHasSelfRefParam	Scans the list of parameters in the current Operation, returning 'T' if one type is the Class reference (this could be ClassA* or ClassA&, depending on the value of the genOptCOperationRefParamStyle code generation option field substitution macro).
opImplMacros	A space-separated list of macros defined in the implementation of this operation.
opIsQuery	'Operation' dialog: 'IsQuery' checkbox.
opMacros	A space-separated list of macros defined in the declaration for this operation.
opName	'Operation' dialog: Name.
opNotes	'Operation' dialog: Notes.
opPure	'Operation' dialog: 'Pure' checkbox.
opReturnArray	'Operation' dialog: 'Return Array' checkbox.
opReturnClassifierGUID	The unique GUID for the classifier of the current operation.

opReturnQualType	The operation return type qualified by the namespace path (if generating namespaces) and the classifier path (dot delimited). If the return type classifier has not been set, it is equivalent to the opReturnType macro.
opReturnType	'Operation' dialog: Return Type.
opScope	'Operation' dialog: Scope.
opStatic	'Operation' dialog: 'Static' checkbox.
opStereotype	'Operation' dialog: Stereotype.
opSynchronized	'Operation' dialog: 'Synchronized' checkbox.

Package Field Substitution Macros

This table lists the Package Field Substitution macros.

Field Substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Package Macros

Macro Name	Description
packageAbstract	'Package' dialog: Abstract.
packageAlias	'Package' dialog: Alias.
packageAuthor	'Package' dialog: Author.
packageComplexity	'Package' dialog: Complexity.
packageGUID	The unique GUID for the current Package.
packageKeywords	'Package' dialog: Keywords.

packageLanguage	'Package' dialog: Language.
packageName	'Package' dialog: Name.
packagePath	The string representing the hierarchy of Packages, for the Class in scope. Each Package name is separated by a dot (.).
packagePhase	'Package' dialog: Phase.
packageScope	'Package' dialog: Scope.
packageStatus	'Package' dialog: Status.
packageStereotype	'Package' dialog: Stereotype.
packageVersion	'Package' dialog: Version.

Parameter Field Substitution Macros

This table lists each of the Parameter field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Parameter Macros

Macro Name	Description
paramClassifierGUID	The unique GUID for the classifier of the current parameter.
paramDefault	Operation 'Parameters' dialog: 'Default' field.
paramFixed	Operation 'Parameters' dialog: 'Fixed' checkbox.
paramGUID	The unique GUID for the current parameter.
paramIsEnum	True, if the parameter uses the enum keyword (C++).
paramKind	Operation 'Parameters' dialog: 'Kind'

	field.
paramName	Operation 'Parameters' dialog: 'Name' field.
paramNotes	Operation 'Parameters' dialog: 'Notes' field.
paramQualType	The parameter type qualified by the namespace path (if generating namespaces) and the classifier path (dot delimited). If the parameter classifier has not been set, is equivalent to the paramType macro.
paramType	Operation 'Parameters' dialog: 'Type' field.

Problem Field Substitution Macros

This table lists each of the Problem field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Problem Macros

Macro Name	Description
problemCompletedBy	'Maintenance' dialog, 'Element Issues' tab: Completed by.
problemCompletedDate	'Maintenance' dialog, 'Element Issues' tab: Completed.
problemHistory	'Maintenance' dialog, 'Element Issues' tab: History.
problemName	'Maintenance' dialog, 'Element Issues' tab: Name.
problemNotes	'Maintenance' dialog, 'Element Issues' tab: Description.
problemPrior	'Maintenance' dialog, 'Element Issues'

ity	tab: Priority.
problemRaisedBy	'Maintenance' dialog, 'Element Issues' tab: Raised by.
problemRaisedDate	'Maintenance' dialog, 'Element Issues' tab: Raised.
problemStatus	'Maintenance' dialog, 'Element Issues' tab: Status.
problemVersion	'Maintenance' dialog, 'Element Issues' tab: Version.

Requirement Field Substitution Macros

This table lists each of the Requirement field substitution macros with a description of the result.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Requirement Macros

Macro Name	Description
requirementDifficulty	'Properties' dialog: 'Require' tab: Difficulty.
requirementLastUpdated	'Properties' dialog: 'Require' tab: Last Update.
requirementName	'Properties' dialog: 'Require' tab: Short Description.
requirementNotes	'Properties' dialog: 'Require' tab: Notes.
requirementPriority	'Properties' dialog: 'Require' tab: Priority.

requirementStatus	'Properties' dialog: 'Require' tab: Status.
requirementType	'Properties' dialog: 'Require' tab: Type.

Resource Field Substitution Macros

This table lists each of the Resource field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Resource Macros

Macro Name	Description
resourceAllocatedTime	Resource Allocation window: Allocated Time.
resourceEndDate	Resource Allocation window: End Date.
resourceExpectedTime	Resource Allocation window: Expected Time.
resourceExpendedTime	Resource Allocation window: Time Expended.
resourceHistory	Resource Allocation window: History.
resourceNam	Resource Allocation window: Resource.

e	
resourceNotes	Resource Allocation window: Description.
resourcePercentCompleted	Resource Allocation window: Completed(%).
resourceRole	Resource Allocation window: Role.
resourceStartDate	Resource Allocation window: Start Date.

Risk Field Substitution Macros

This table lists each of the Risk field substitution macros.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Risk Macros

Macro Name	Description
riskName	Risks window: Risk.
riskNotes	Risks window: (Notes).
riskType	Risks window: Type.
riskWeight	Risks window: Weight.

Scenario Field Substitution Macros

This table lists each of the Scenario field substitution macros with a description of the result.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Scenario Macros

Macro Name	Description
scenarioGUID	The unique ID for a scenario. Identifies the scenario unambiguously within a model.
scenarioName	'Properties' dialog, 'Scenario' tab: Scenario.
scenarioNotes	'Properties' dialog, 'Scenario' tab: (Notes).
scenarioType	'Properties' dialog, 'Scenario' tab: Type.

Tagged Value Substitution Macros

Tagged Value macros are a special form of field substitution macros, which provide access to element tags and the corresponding Tagged Values. They can be used in one of two ways:

- Direct Substitution
- Conditional Substitution

Direct Substitution

This form of the macro directly substitutes the value of the named tag into the output.

Structure: %<macroName>:"<tagName>"%

<macroName> can be one of:

- attTag
- classTag
- connectorDestElemTag
- connectorDestTag
- connectorSourceElemTag
- connectorSourceTag
- connectorTag
- linkAttTag
- linkTag
- opTag
- packageTag

- paramTag

This corresponds to the tags for attributes, Classes, operations, Packages, parameters, connectors with both ends, elements at both ends of connectors and connectors including the attribute end.

<tagName> is a string representing the specific tag name.

Example

%opTag:"attribute"%

Conditional Substitution

This form of the macro mimics the conditional substitution defined for field substitution macros.

Structure: %<macroName>:"<tagName>" (== "<test>") ? <subTrue> (: <subFalse>) %

Note:

- <macroName> and <tagName> are as defined here
- (<text>) denotes that <text> is optional
- <test> is a string representing a possible value for the macro
- <subTrue> and <subFalse> can be a combination of quoted strings and the keyword value; where the value is used, it gets replaced with the macro's value in the output

Examples

%opTag:"opInline" ? "inline" : ""%

%opTag:"opInline" ? "inline"%

%classTag:"unsafe" == "true" ? "unsafe" : ""%

%classTag:"unsafe" == "true" ? "unsafe"%

Tagged Value macros use the same naming convention as field substitution macros.

Template Parameter Substitution Macros

If you want to provide access in a transformation template to data concerning the transformation of a Template Binding connector's binding parameter substitution in the model, you can use the Template Parameter substitution macros. The macro names are in Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected; otherwise the value is empty.

Template Parameter substitution macros

Macro Name	Description
parameterSubstitutionFormal	'Template Binding Properties' dialog, 'Binding Parameter' tab, 'Parameter Substitution(s)' panel: Formal Template Parameter name.
parameterSubstitutionActual	'Template Binding Properties' dialog, 'Binding Parameter' tab, 'Parameter Substitution(s)' panel: Actual parameter name/expression.
parameterSubstitutionActualClassifier	'Template Binding Properties' dialog, 'Binding Parameter' tab, 'Parameter Substitution(s)' panel: Actual parameter

	classifier.

Test Field Substitution Macros

This table lists each of the Test field substitution macros with a description of the result.

Field substitution macros are named according to Camel casing. Macros that represent checkboxes return a value of 'T' if the box is selected. Otherwise the value is empty.

Test Macros

Macro Name	Description
testAcceptanceCriteria	Testing dialog: Acceptance Criteria.
testCheckedBy	Test Cases window: Checked By.
testDateRun	Test Cases window: Last Run.
testClass	Test Cases window: Test Class (the type of test defined: Unit, Integration, System, Acceptance, Inspection, Scenario)
testInput	Testing dialog: Input.
testName	Test Cases window: Test.

testNotes	Test Cases window: Description.
testResults	Testing dialog: Results.
testRunBy	Test Cases window: Run By. (Values are derived from the Project Author definitions in the 'People' dialog - 'Settings > Reference Data > Model Types > People > Project Authors'.)
testStatus	Test Cases window: Status.
testType	Test Cases window: Type.

Function Macros

Function macros are a convenient way of manipulating and formatting various element data items. Each function macro returns a result string. There are two primary ways to use the results of function macros:

- Direct substitution of the returned string into the output, such as: %TO_LOWER(attName)%
- Storing the returned string as part of a variable definition such as: $name = %TO_LOWER(attName)%

Function macros can take parameters, which can be passed to the macros as:

- String literals, enclosed within double quotation marks
- Direct substitution macros without the enclosing percent signs
- Variable references
- Numeric literals

Multiple parameters are passed using a comma-separated list.

Function macros are named according to the All-Caps style, as in:

%CONVERT_SCOPE(opScope)%

The available function macros are described here. Parameters are denoted by square brackets, as in:

FUNCTION_NAME([param]).

CONVERT_SCOPE([umlScope])

For use with supported languages, to convert [umlScope] to the appropriate scope keyword for the language being generated. This table shows the conversion of [umlScope] with respect to the given language.

Language	Conversions
C++	Package ==> public Public ==> public Private ==> private Protected ==> protected
C#	Package ==> internal Public ==> public Private ==> private Protected ==> protected
Delphi	Package ==> protected Public ==> public Private ==> private Protected ==> protected
Java	Package ==> {blank} Public ==> public Private ==> private Protected ==> protected

PHP	Package ==> public Public ==> public Private ==> private Protected ==> protected
VB	Package ==> Protected Public ==> Public Private ==> Private Protected ==> Protected
VB .Net	Package ==> Friend Public ==> Public Private ==> Private Protected ==> Protected

COLLECTION_CLASS([language])

Gives the appropriate collection Class for the language specified for the current linked attribute.

CSTYLE_COMMENT([wrap_length])

Converts the notes for the element currently in scope to plain C-style comments, using /* and */.

DELPHI_PROPERTIES([scope], [separator], [indent])

Generates a Delphi property.

DELPHI_COMMENT([wrap_length])

Converts the notes for the element currently in scope to Delphi comments.

EXEC_ADD_IN(, [function_name],, ...,)

Invokes an Enterprise Architect Add-In function, which can return a result string.

[addin_name] and [function_name] specify the names of the Add-In and function to be invoked.

Parameters to the Add-In function can be specified via parameters [prm_1] to [prm_n].

```
$result = %EXEC_ADD_IN("MyAddin",
"ProcessOperation", classGUID, opGUID)%
```

Any function that is to be called by the EXEC_ADD_IN macro must have two parameters: an EA.Repository object, and a Variant array that contains any additional parameters from the EXEC_ADD_IN call. Return type should be Variant.

```
Public Function ProcessOperation(Repository As
EA.Repository, args As Variant) As Variant
```

FIND([src], [subString])

Position of the first instance of [subString] in [src]; -1 if none.

GET_ALIGNMENT()

Returns a string where all of the text on the current line of output is converted into spaces and tabs.

JAVADOC_COMMENT([wrap_length])

Converts the notes for the element currently in scope to javadoc -style comments.

LEFT([src], [count])

The first [count] characters of [src].

LENGTH([src])

Length of [src]. Returns a string.

MATH_ADD(x,y) MATH_MULT(x,y) and MATH_SUB(x,y)

In a code template or DDL template, these three macros

perform, respectively, the mathematical functions of:

- Addition (x+y)
- Multiplication (x*y) and
- Subtraction (x-y)

The arguments x and y can be integers or variables, or a combination of the two. Consider these examples, as used in a 'Class' template for C++ code generation:

- $a = %MATH_ADD(3,4)%
- $b = %MATH_SUB(10,3)%
- $c = %MATH_MULT(2,3)%
- $d = %MATH_ADD($a,$b)%
- $e = %MATH_SUB($b,$c)%
- $f = %MATH_MULT($a,$b)%
- $g = %MATH_MULT($a,10)%
- $h = %MATH_MULT(10,$b)%

These compute, in the same sequence, to:

- a = 3 + 4 = $a
- b = 10 - 3 = $b
- c = 2 * 3 = $c
- d = a + b = $d
- e = b - c = $e
- f = a * b = $f
- g = a * 10 = $g
- h = 10 * b = $h

When the code is generated, the .h file (for C++) contains

these corresponding strings:

- a = 3 + 4 = 7
- b = 10 - 3 = 7
- c = 2 * 3 = 6
- d = a + b = 14
- e = b - c = 1
- f = a * b = 49
- g = a * 10 = 70
- h = 10 * b = 70

MID([src], [start]) MID([src], [start], [count])

Substring of [src] starting at [start] and including [count] characters. Where [count] is omitted the rest of the string is included.

PI([option], [value], {[option], [value]})

Sets the PI for the current template to [value]. Valid values for [value] are:

- "\n"
- "\t "
- “ “
- “”

<option> controls when the new PI takes effect. Valid values for <option> are:

- I, Immediate: the new PI is generated before the next non-empty template line
- N, Next: the new PI is generated after the next non-empty template line

Multiple pairs of options are allowed in one call. An example of the situation where this would used is where one keyword is always on a new line, as illustrated here:

%PI=" "%

%classAbstract ? "abstract"%

%if classTag:"macro" != ""%

%PI("I", "\n", "N", " ")%

%classTag:"macro"%

%endIf%

class

%className%

For more details, see *The Processing Instruction (PI) Macro*.

PROCESS_END_OBJECT([template_name])

Enables the Classes that are one Class further away from the base Class, to be transformed into objects (such as attributes, operations, Packages, parameters and columns) of the base Class. [template_name] refers to the working template that temporarily stores the data.

REMOVE_DUPLICATES([source], [separator])

Where [source] is a [separator] separated list; this removes any duplicate or empty strings.

REPLACE([string], [old], [new])

Replaces all occurrences of [old] with [new] in the given string <string>.

RESOLVE_OP_NAME()

Resolves clashes in interface names where two method-from interfaces have the same name.

RESOLVE_QUALIFIED_TYPE()
RESOLVE_QUALIFIED_TYPE([separator])
RESOLVE_QUALIFIED_TYPE([separator], [default])

Generates a qualified type for the current attribute, linked attribute, linked parent, operation, or parameter. Enables the specification of a separator other than. and a default value for when some value is required.

RIGHT([src], [count])

The last [count] characters of [src].

TO_LOWER([string])

Converts [string] to lower case.

TO_UPPER([string])

Converts [string] to upper case.

TRIM([string]) TRIM([string], [trimChars])

Removes trailing and leading white spaces from [string]. If [trimChars] is specified, all leading and trailing characters in the set of <trimChars> are removed.

TRIM_LEFT([string]) TRIM_LEFT([string], [trimChars])

Removes the specified leading characters from <string>.

TRIM_RIGHT([string]) TRIM_RIGHT([string], [trimChars])

Removes the specified trailing characters from <string>.

VB_COMMENT([wrap_length])

Converts the notes for the element currently in scope to Visual Basic style comments.

WRAP_COMMENT([comment], [wrap_length], [indent], [start_string])

Wraps the text [comment] at width [wrap_length] putting [indent] and [start_string] at the beginning of each line.

```
$behavior = %WRAP_COMMENT(opBehavior, "40", "
", "//")%
```

<wrap_length> must still be passed as a string, even though WRAP_COMMENT treats this parameter as an integer.

WRAP_LINES([text], [wrap_length], [start_string] {, [end_string] })

Wraps [text] as designated to be [wrap_length], adding [start_string] to the beginning of every line and [end_string] to the end of the line if it is specified.

XML_COMMENT([wrap_length])

Converts the notes for the element currently in scope to XML-style comments.

Control Macros

Control macros are used to control the processing and formatting of the templates. The basic types of control macro include:

- The list macro, for generating multiple element features, such as attributes and operations
- The branching macros, which form if-then-else constructs to conditionally execute parts of a template
- The PI macro for formatting new lines in the output, which takes effect from the next non-empty line
- A PI function macro that enables setting PI to a variable and adds the ability to set the PI that is generated before the next line
- The synchronization macros

In general, control macros are named according to Camel casing.

List Macro

If you need to loop or iterate through a set of Objects that are contained within or are under the current object, you can do so using the *%list* macro. This macro performs an iterative pass on all the objects in the scope of the current template, and calls another template to process each one.

The basic structure is:

%list=<TemplateName> @separator=<string> @indent=<string> (<conditions>) %

where <string> is a double-quoted literal string and <TemplateName> can be one of these template names:

- Attribute
- AttributeImpl
- Class
- ClassBase
- ClassImpl
- ClassInitializer
- ClassInterface
- Constraint
- Custom Template (custom templates enable you to define your own templates)
- Effort
- InnerClass
- InnerClassImpl
- LinkedFile

- Metric
- Namespace
- Operation
- OperationImpl
- Parameter
- Problem
- Requirement
- Resource
- Risk
- Scenario
- Test

<conditions> is optional and looks the same as the conditions for 'if' and 'elseIf' statements.

Example

In a Class transform, the Class might contain multiple attributes; this example calls the Attribute transform and outputs the result of processing the transform for each attribute of the Class in scope. The resultant list separates its items with a single new line and indents them two spaces respectively. If the Class in scope had any stereotyped attributes, they would be generated using the appropriately specialized template.

%list="Attribute" @separator="\n" @indent=" "%

The separator attribute, denoted by @separator, specifies the space that should be used between the list items, excluding

the last item in the list.

The indent attribute, denoted by @indent, specifies the space by which each line in the generated output should be indented.

Special Cases

There are some special cases to consider when using the %list macro:

- If the Attribute template is used as an argument to the %list macro, this also generates attributes derived from Associations by executing the appropriate LinkedAttribute template
- If the ClassBase template is used as an argument to the %list macro, this also generates Class bases derived from links in the model by executing the appropriate LinkedClassBase template
- If the ClassInterface template is used as an argument to the %list macro, this also generates Class bases derived from links in the model by executing the appropriate LinkedClassInterface template
- If InnerClass or InnerClassImpl is used as an argument to the %list macro, these Classes are generated using the Class and ClassImpl templates respectively; these arguments direct that the templates should be processed based on the inner Classes of the Class in scope

Branching Macros

Branching macros provide if-then-else constructs. The CTF supports a limited form of branching through these macros:

- if
- elseIf
- else
- endIf
- endTemplate (which exits the current template)

The basic structure of the if and elseIf macros is:

%if <test> <operator> <test>%

where <operator> can be one of:

- ==
- !=
- < (mathematics comparison, less than)
- > (mathematics comparison, greater than)
- <= (mathematics comparison, less than or equal to)
- >= (mathematics comparison, greater than or equal to)

and <test> can be one of:

- a string literal, enclosed within double quotation marks
- a direct substitution macro, without the enclosing percent signs
- a variable reference

Note that if you are using one of the mathematics comparison operators, <test> must be a decimal number in

string format.

Branches can be nested, and multiple conditions can be specified using one of:

- and, or
- or

When specifying multiple conditions, 'and' and 'or' have the same order of precedence, and conditions are processed left to right.

If conditional statements on strings are case sensitive, 'a String' does not equal 'A STRING'. Hence in some situations it is better to set the variable $str=TO_LOWER(variable) or TO_UPPER(variable) and then compare to a specific case.

Macros are not supported in the conditional statements. It is best to assign the results of a macro (string) to a variable, and then use the variable in the comparison.

```
$fldType = % TO_LOWER ($parameter1)%
$COMMENT = "Use the first 4 characters for Date and Time field types"
$fldType4 = % LEFT ($fldType, 4)%
%if $fldType4 == "date"%
Datetime
%endif%
```

This takes a parameter of value “Datetime”, “DATETIME” or “Date”, and returns “Datetime”.

The endif or endTemplate macros must be used to signify the end of a branch. In addition, the endTemplate macro

causes the template to return immediately, if the corresponding branch is being executed.

Example 1

```
%if elemType == "Interface"%
;
%else%
%OperationBody%
%endIf%
```

In this case:

- If the elemType is "Interface" a semi-colon is returned
- If the elemType is not "Interface", a template called Operation Body is called

Example 2

```
$bases="ClassBase"
$interfaces=""%
%if $bases !="" and $interfaces !=""%
: $bases, $interfaces
%elseIf $bases !=""%
: $bases
%elseIf $interfaces !=""%
: $interfaces
%endIf%
```

In this case the text returned is ':ClassBase'.

Conditions using Boolean Value

When setting up branching using conditions that involve a system checkbox (Boolean fields), such as Attribute.Static (attStatic) the conditional statement would be written as:

```
%if attStatic == "T"%
```

For example:

```
% if attCollection == "T" or attOrderedMultiplicity == "T" %
% endTemplate %
```

Synchronization Macros

The synchronization macros are used to provide formatting hints to Enterprise Architect when inserting new sections into the source code, during forward synchronization. The values for synchronization macros must be set in the File templates.

The structure for setting synchronization macros is:

%<name>=<value>%

where <name> can be one of the macros listed here and <value> is a literal string enclosed by double quotes.

Synchronization Macros

Macro Name	Description
synchNewClassNotesSpace	Space to append to a new Class note. Default value: \n.
synchNewAttributeNotesSpace	Space to append to a new attribute note. Default value: \n.
synchNewOperationNotesSpace	Space to append to a new operation note. Default value: \n.

synchNewOperationBodySpace	Space to append to a new operation body. Default value: \n.
synchNamespaceBodyIndent	Indent applied to Classes within non-global namespaces. Default value: \t.

The Processing Instruction (PI) Macro

The PI (Processing Instruction) macro provides a means of defining the separator text to be inserted between the code pieces (which represent entities) that are generated using a template.

The structure for setting the Processing Instruction is:

%PI=<value>%

In this structure, <value> is a literal string enclosed by double quotes, with these options:

- "\n" - New line (the default)
- " " - Space
- "\t" - Tab
- "" - Null

By default, the PI is set to generate a new line (\n) for each non-empty substitution, which behavior can be changed by resetting the PI macro. For instance, a Class's Attribute declaration in simple VB code would be generated to a single line statement (with no new lines). These properties are derived from the Class-Attribute properties in the model to generate, for example:

Private Const PrintFormat As String = "Portrait"

The template for generating this starts with the PI being set to a space rather than a new line:

% PI = " " %

% CONVERT_SCOPE (attScope)%

% endIf %

% if attConst == "T" %

Const

% endIf %

On transforming this, attscope returns the VB keyword 'Private' and attConst returns 'Const' on the same line spaced by a single space (fitting the earlier VB Class.Attribute definition example).

Alternatively, when generating a Class you might want the Class declaration, the notes and Class body all separated by double lines. In this case the %PI is set to '/n/n' to return double line spacing:

% PI = "\n\n" %

% ClassDeclaration %

% ClassNotes %

% ClassBody %

PI Characteristics

- Blank lines have no effect on the output
- Any line that has a macro that produces an empty result does not result in a PI separator (space/new line)
- The last entry does not return a PI; for example, %Classbody% does not have a double line added after the body

Code Generation Macros for Executable StateMachines

The templates listed here are available through the Code Template Editor (the 'Develop > Source Code > Options > Edit Code Templates' ribbon option); select 'STM_C++_Structured' in the 'Language' field.

The templates are structured as shown:

StmContextStateMachineEnum
    StmStateMachineEnum

StmContextStateEnum
    StmAllStateEnum

StmContextTransitionEnum
    StmTransitionEnum

StmContextEntryEnum
    StmAllEntryEnum

StmContextStateMachineStringToEnum
    StmStateMachineStringToEnum

StmContextStateEnumToString

StmStateEnumToString

StmContextTransitionEnumToString

StmTransitionEnumToString

StmContextStateNameToGuid

StmStateNameToGuid

StmContextTransitionNameToGuid

StmTransitionNameToGuid

StmContextDefinition

StmStateMachineEnum

StmAllStateEnum

StmTransitionEnum

StmAllEntryEnum

StmAllRegionVariableInitialize

StmStateWithDeferredEvent

StmDeferredEvent

StmTransitionProcMapping

StmTransitionProc

StmTransitionExit

StmTransitionEntry

StmTargetOutgoingTransition

StmTargetParentSubmachineState

StmStateProcMapping
StmStateProc
StmStateEntry
StmOutgoingTransition
StmConnectionPointReferenceEntry
StmParameterizedInitial
StmSubMachineInitial
StmRegionInitial
StmRegionDeactive
StmStateExitProc
StmStateTransition
StmStateEvent
StmStateTriggeredTransition
StmStateCompletionTransition
StmStateIncomingTransition
StmStateOutgoingTransition
StmSubmachineStateExitEvent
StmVertexOutgoingTransition
StmConnectionPointReferenceExitEvent
StmStateExitEvent
StmVertexOutgoingTransition
StmAllRegionVariable
StmStateMachineStringToEnum
StmStateMachineRun
StmStateInitialData

StmStateMachineEntry

StmOutgoingTransition

StmStateMachineRunInitial

StmStateMachineInitial

StmStateMachineRuns

StmContextManager

StmSimulationManager

StmContextInstanceDeclaration

StmContextInstance

StmContextVariableRunstate

StmContextInstanceAssociation

StmContextInstanceClear

StmEventProxy

StmSignalEnum

StmContextJoinEventEnum

StmJoinEventEnum

StmEventEnum

StmSignalDefinition

StmSignalAttributeAssignment

StmSignalAttribute

StmSignalInitialize

StmEventStringToEnum

StmEventEnumToString
StmEventNameToGuid

StmConsoleManager
StmContextInstanceDeclaration
StmContextInstance
StmContextVariableRunstate
StmContextInstanceAssociation
StmContextInstanceClear

StmStateMachineStrongToEnum

StmInitialForTransition

StmVertextOutgoingTransition

StmSendEvent

StmBroadcastEvent

StmContextRef

Signal & Event

Macro name	Description

stmEventEnum	The name of the Event with the prefix 'ENUM_', all upper case.
StmEventGuid	The GUID of the Event.
stmEventName	The name of the Event with spaces and asterisks removed.
stmEventVariable	The name of the Event with the prefix 'm_' in lower case.
stmIsSignalEvent	Is 'T' if the element is a SignalEvent.
stmSignalEnum	The name of the Signal with the prefix 'ENUM_', all upper case.
stmSignalFirstEvent	The name of the Event with the prefix 'ENUM_', all upper case.
stmSignalGuid	The GUID of the Signal.
stmSignalName	The name of the Signal with spaces and asterisks removed.
stmSignalVar	The name of the Signal with the prefix

iable	'm_' in lower case.
stmTriggerName	Transition Properties: The name of the Trigger.
stmTriggerSpecification	Transition Properties: The specification of the Trigger.
stmTriggerType	Transition Properties: The type of the Trigger.

Context

Macro name	Description
stmContextName	The name of the Class with spaces and asterisks removed.
stmContextQualName	The qualified name of the Class for which code is being generated.
stmContextVariableName	
stmContextFileName	The output file name for the Class for which code is being generated.

Writing Object Runstate to StateMachine Initialization

Macro name	Description
stmContextVariableRunstateName	
stmContextVariableRunstateValue	
stmContextHasStatemachine	Is 'T' if the current context has one or more StateMachines.
stmHasHistoryPattern	Is 'T' if the StateMachine has a History Pattern.
stmHasTerminatePattern	Is 'T' if the StateMachine has a Terminate Pattern.
stmHasDeferredEventPatt	Is 'T' if the StateMachine has a Deferred Event Pattern.

ern	
stmHasSubmachinePattern	Is 'T' if the StateMachine has a Submachine Pattern.
stmHasOrthogonalPattern	Is 'T' if the StateMachine has an Orthogonal Pattern.

StateMachine

Macro name	Description
stmStatemachineName	The name of the StateMachine with asterisks and spaces removed.
stmStatemachineEnum	The name of the StateMachine plus 'ENUM_' plus the name of the StateMachine in upper case.
stmStatemachineGuid	The GUID of the StateMachine element.
stmStateCount	The number of State elements in the StateMachine.
stmSubmachineInitialCoun	The number of Initial elements in the Sub

t	Machine State element.
stmStatemachineHasSubmachineState	Is 'T' if the StateMachine has at least one SubMachine State.
stmStatemachineInitialCount	The number of Initial elements in the StateMachine.

Region

Macro name	Description
stmRegionEnum	The name of the State Region plus 'ENUM_' plus the name of the State Region in upper case.
stmRegionFQName	The fully qualified name of the State Region.
stmRegionName	The name of the State Region with spaces and asterisks removed.
stmRegionVariable	The name of the State Region with the prefix 'm_' in lower case.

stmRegionFQVariable	The fully qualified name of the State Region with the prefix 'm_' in lower case.
stmRegionGuid	The GUID of the Region.
stmRegionInitial	
stmRegionIsOwnedByStateMachine	Is 'T' if the Region is owned by a StateMachine.

Transition

Macro name	Description
stmTransitionEnum	The name of the Transition with the prefix 'ENUM_', plus the name of the Transition in upper case.
stmTransitionGuid	The GUID of the Transition.
stmTransitionName	The name of the Transition with spaces and asterisks removed.

stmTransitionSourceGuid	The GUID of the Source element in the Transition.
stmTransitionTargetGuid	The GUID of the Target element in the Transition.
stmTransitionVariable	The name of the Transition with the prefix 'm_' in lower case.
stmTransitionSourceVariable	
stmTransitionTargetVariable	
stmTransitionFQVariable	
stmSourceVertexEnum	The name of the Transition's source vertex plus '_ENUM' plus the name of the Transition's source vertex in upper case.
stmTargetVertexEnum	The name of the Transition's target vertex plus '_ENUM' plus the name of the Transition's target vertex in upper case.
stmSourceIsI	Is 'T' if the Transition's source is an

nitial	Initial.
stmSourceIsState	Is 'T' if the Transition's source is a State.
stmSourceIsEntryPoint	Is 'T' if the Transition's source is an Entry Point.
stmSourceIsExitPoint	Is 'T' if the Transition's source is an Exit Point.
stmSourceIsFork	Is 'T' if the Transition's source is a Fork.
stmSourceIsJoin	Is 'T' if the Transition's source is a Join element.
stmTargetIsFinalState	Is 'T' if the Transition's target is a Final State element.
stmTargetIsExitPoint	Is 'T' if the Transition's target is an Exit Point element.
stmTargetIsState	Is 'T' if the Transition's target is a State element.
stmTargetIsChoice	Is 'T' if the Transition's target is a Choice element.

stmTargetIsJunction	Is 'T' if the Transition's target is a Junction element.
stmTargetIsEntryPoint	Is 'T' if the Transition's target is an Entry Point element.
stmTargetIsConnectionPointReference	Is 'T' if the Transition's target is a Connection Point Reference element.
stmTargetIsFork	Is 'T' if the Transition's target is a Fork element.
stmTargetIsJoin	Is 'T' if the Transition's target is a Join element.
stmTransitionEffect	The Effect of the Transition.
stmTransitionGuard	The Guard of the Transition.
stmTransitionKind	The type or kind of the Transition.
stmTargetInitialTransition	
stmTargetIsS	Is 'T' if the Transition's target is a

ubmachineState	Submachine State.
stmSourceStateEnum	The name of the Transition's source state with the prefix '_ENUM' in upper case.
stmTargetStateEnum	The name of the Transition's target state, with the prefix '_ENUM' in upper case.
stmTargetVertexFQName	The fully qualified name of the Transition's target vertex.
stmTargetIsDeepHistory	Is 'T' if the Transition's target is a Deep History State.
stmTargetIsShallowHistory	Is 'T' if the Transition's target is a Shallow History State.
stmTargetIsTerminate	Is 'T' if the Transition's target is a Terminate element.
stmParentIsStateMachine	Is 'T' if the vertex is an Entry Point or Exit Point, or if the container is a StateMachine.
stmSourceParentStateEnum	

stmTargetParentStateEnum	
stmTargetSubmachineEnum	
stmTargetRegionIndex	
stmIsSelfTransition	Is 'T' if the Transition's source is the same as its target.
stmHistoryOwningRegionInitialTransition	
stmDefaultHistoryTransition	

Vertex and State

Macro name	Description

stmVertexName	The name of the Vertex.
stmStateName	The name of the State.
stmVertexGuid	The GUID of the Vertex.
stmVertexFQName	The fully qualified name of the Vertex.
stmStateFQName	The fully qualified name of the State.
stmVertexType	The type of the vertex; one of 'State', 'FinalState', 'Pseudostate', 'ConnectionPointReference' or ' ' (empty).
stmPseudostateKind	The kind of the Pseudostate; one of 'initial', 'deepHistory', 'shallowHistory', 'join', 'fork', 'junction', 'choice', 'entryPoint', 'exitPoint' or 'terminate'.
stmPseudostateName	The name of the Pseudostate.
stmPseudosta	The name of the Pseudostate with the

teVariable	prefix 'm_' in lower case.
stmPseudostateStateMachineName	The name of the Pseudostate StateMachine.
stmPseudostateStateMachineVariable	The name of the Pseudostate StateMachine with the prefix 'm_' in lower case.
stmVertexVariable	The name of the Vertex with the prefix 'm_' in lower case.
stmVertexEnum	The name of the Vertex plus '_ENUM' plus the name of the Vertex in upper case.
stmStateEnum	The name of the State plus '_ENUM' plus the name of the State in upper case.
stmConnectionPointReferenceStateName	The name of the Connection Point Reference.
stmConnectionPointReferenceStateVariable	The name of the Connection Point Reference with the prefix 'm_' in lower case.

stmConnectionPointReferenceEntryCount	
stmParameterizedInitialCount	
stmInitialCountForTransition	
stmStateVariable	The name of the State with the prefix 'm_' in lower case.
stmStateEntryBehavior	The behavior defined for an 'entry' Action operation for a State (the text on the 'Behavior' tab for the 'entry' Action operation on the Features window for the element).
stmStateEntryCode	The initial code defined for an 'entry' Action operation for a State (the text for the 'entry' Action operation on the Behavior's 'Code' tab).
stmStateDoB	The behavior defined for a 'do' Action

ehavior	operation for a State (the text on the 'Behavior' tab for the 'do' Action operation on the Features window for the element).
stmStateDoCode	The initial code defined for a 'do' Action operation for a State (the text for the 'do' Action operation on the Behavior's 'Code' tab).
stmStateExitBehavior	The behavior defined for an 'exit' Action operation for a State (the text on the 'Behavior' tab for the 'exit' Action operation on the Features window for the element).
stmStateExitCode	The initial code defined for an 'exit' Action operation for a State (the text for the 'exit' Action operation on the Behavior's 'Code' tab).
stmStateSubmachineName	The name of the Submachine.
stmStateSubmachineVariable	The name of the Submachine with the prefix 'm_' in lower case.

stmStateIsFinal	Is 'T' if the State is a FinalState.
stmStateIsSubmachineState	Is 'T' if the State is a Submachine State ('Properties' page | Advanced | 'isSubmachineState' property).
stmSubMachineEnum	The name of the Submachine followed by '_ENUM' plus the name of Submachine in upper case.
stmStateHasChildrenToJoin	
stmStateIsTransitionTarget	
stmThisIsSource	
stmThisIsSourceState	
stmStateParentIsSubmachine	Is 'T' if the State's container is a StateMachine.
stmStateCont	

ainerMatchTransitionContainer	
stmVertexRegionIndex	
stmStateRegionCount	The number of regions in the State.
stmStateInitialCount	The number of Initial elements in the StateMachine.
stmVertexContainerVariable	
stmVertexParentEnum	
stmStateHasUnGuardedCompletionTransition	
stmStateEventHasUnGuardedTransition	

stmInitialTransition	

Instance Association

Macro name	Description
stmSourceInstanceName	
stmTargetInstanceName	
stmSourceRoleName	
stmTargetRoleName	

EASL Code Generation Macros

Enterprise Architect provides a number of Enterprise Architect Simulation Library (EASL) code generation macros to generate code from behavioral models. These are:

- EASL_INIT
- EASL_GET
- EASLList and
- EASL_END

EASL_INIT

The EASL_INIT macro is used to initialize an EASL behavior model. The behavior model code generation is dependent on this model.

Aspect	Description
Syntax	%EASL_INIT(<<GUID>>)% where: • <<GUID>> is the GUID of the Object (usually a Class element) that is the owner of the behavior model

EASL_GET

The EASL_GET macro is used to retrieve a property or a

collection of an EASL object. The EASL objects and the properties and collections for each object are identified in the *EASL Collections* and *EASL Properties* topics.

Aspect	Description
Syntax	$result = %EASL_GET(<<Property>>, <<Owner ID>>, <<Name>>)% where: • <<Property>> is one of "Property", "Collection", "At", "Count", or "IndexOf" • <<OwnerID>> is the ID of the owner object for which the property/collection is to be retrieved • <<Name>> is the name of the property or Collection being accessed • $result is the returned value; this is “” if not a valid property If <<Property>> is: • "At", then <<OwnerID>> is the ID of a collection and <<Name>> is the index into the collection for which the item is to be retrieved • "Count", then <<Owner ID>> is the ID of a collection and <<Name>> is not used; it will retrieve the item number in the collection • "IndexOf", then <<Owner ID>> is the

	ID of a collection and <<Name>> is the ID of the item in the collection; it will retrieve the index (string format) of the item within the collection
Example	$sPropName = %EASL_GET("Property", $context, "Name")%

EASLList

The EASLList macro is used to render each object in an EASL collection using the appropriate template.

Aspect	Description
Syntax	$result = %EASLList=<<TemplateName>> @separator=<<Separator>> @indent=<<indent>> @owner=<<OwnedID>> @collection=<<CollectionName>> @option1=<<OPTION1>> @option2=<<OPTION2>>..... @optionN=<<OPTIONN>>% where: • <<TemplateName>> is the name of any behavioral model template or

	custom template • <<Separator>> is a list separator (such as “\n”) • <<indent>> is any indentation to be applied to the result • <<OwnedID>> is the ID of the object that contains the required collection • <<CollectionName>> is the name of the required collection • <<OPTION1>...<<OPTION99>> are miscellaneous options that might be passed on the template; each option is given as an additional input parameter to the template • $result is the resultant value; this is “” if not a valid collection
Example	$sStates = %EASLList="State" @separator="\n" @indent="\t" @owner=$StateMachineGUID @collection="States" @option=$sOption%

EASL_END

The EASL_END macro is used to release the EASL behavior model.

Aspect	Description
Syntax	%EASL_END%

Behavioral Model Templates

- Action
- Action Assignment
- Action Break
- Action Call
- Action Create
- Action Destroy
- Action If
- Action Loop
- Action Opaque
- Action Parallel
- Action RaiseEvent
- Action RaiseException
- Action Switch
- Behavior
- Behavior Body
- Behavior Declaration
- Behavior Parameter
- Call Argument

- Decision Action
- Decision Condition
- Decision Logic
- Decision Table
- Guard
- Property Declaration
- Property Notes
- Property Object
- State
- State CallBack
- State Enumerate
- State EnumeratedName
- StateMachine
- StateMachine HistoryVar
- Transition
- Transition Effect
- Trigger

EASL Collections

This topic lists the EASL collections for each of the EASL objects, as retrieved by the EASL Code Generation Macros code generation macro.

Action

Collection Name	Description
Arguments	The Action's arguments.
SubActions	The sub-actions of the Action.

Behavior

Collection Name	Description
Actions	The Behavior's Actions.
Nodes	The Behavior's nodes.
Parameters	The Behavior's parameters.

Variables	The Behavior's variables.

Classifier

Collection Name	Description
AllStateMachines	All StateMachines for the Classifier.
AsynchProperties	The asynchronous properties of the Classifier.
AsynchTriggers	The asynchronous triggers of the Classifier.
Behaviors	The behaviors of the Classifier.
Properties	The properties of the Classifier.
TimedProperties	The timed properties of the Classifier.
TimedTriggers	The timed triggers of the Classifier.

Triggers	All triggers of the Classifier.

Construct

Collection Name	Description
AllChildren	The Construct's children.
ClientDependencies	The client dependencies on the Construct.
StereoTypes	The stereotypes of the Construct.
SupplierDependencies	The supplier dependencies on the Construct.

Node

Collection Name	Description
IncomingEdges	The Node's incoming edges.

OutgoingEdges	The Node's outgoing edges.
SubNodes	The sub-nodes of the Node.

State

Collection Name	Description
DoBehaviors	The State's Do behaviors.
EntryBehaviors	The State's Entry behaviors.
ExitBehaviors	The State's Exit behaviors.

StateMachine

Collection Name	Description
AllFinalState	

s	The StateMachine's final States.
AllStates	All States within the StateMachine, including those within Submachine States.
DerivedTransitions	The StateMachine's derived Transitions with the associated valid effect.
States	The States within the StateMachine.
Transitions	The transitions within the StateMachine.
Vertices	The StateMachine's vertices.

Transition

Collection Name	Description
Effects	The Transition's effects.
Guards	The Transition's guards.
Triggers	The Transition's triggers.

Trigger

Collection Name	Description
TriggeredTransitions	The triggered transitions associated with the Trigger.

Vertex

Collection Name	Description
DerivedOutgoingTransitions	The Vertex's derived outgoing transitions after traversing the pseudo-nodes.
IncomingTransitions	The Vertex's incoming transitions.
OutgoingTransitions	The Vertex's outgoing transitions.

EASL Properties

This topic lists the EASL properties for each of the EASL objects, as retrieved by the EASL Code Generation Macros code generation macro.

Action

Property Name	Description
Behavior	The Action's associated behavior (Call Behavior Action or Call Operation Action).
Body	The Action's body.
Context	The Action's context.
Guard	The Action's guard.
IsFinal	A check on whether the action is a final Action.
IsGuarded	A check on whether the action is a guarded Action.
IsInitial	A check on whether the action is an

	initial Action.
Kind	The Action's kind.
Next	The Action's next action.
Node	The Action's associated node in the graph.

Argument

Property Name	Description
Parameter	The ID of the Argument's associated parameter.
Value	The default value of the argument.

Behavior

Property Name	Description

InitialAction	The Behavior's initial action.
isReadOnly	The isReadOnly of the Behavior.
isSingleExecution	The isSingleExecution of the Behavior.
Kind	The kind of Behavior.
ReturnType	The return type of the Behavior.
Specification	The specification of the Behavior.

CallEvent

Property Name	Description
Operation	The operation of the CallEvent.

ChangeEvent

Property Name	Description

ChangeExpression	The change expression of the ChangeEvent.

Classifier

Property Name	Description
HasBehaviors	A check on whether the Classifier has behavioral models (Activity and Interaction).
Language	The Classifier's language.
StateMachine	The StateMachine of the Classifier.

Condition

Property Name	Description
Expression	The Condition's expression.

Lower	The Condition's lower value.
Upper	The Condition's upper value.

Construct

Property Name	Description
GetTaggedValue	The Property's Tagged Value.
IsStereotypeApplied	A check on whether a particular stereotype is applied to the Property.
Notes	Notes on the Property.
UMLType	The UML type of the Property.
Visibility	The visibility of the Property.

Edge

Property	Description

Name	
From	The ID of the node from which the Edge arises.
To	The ID of the node at which the Edge is targeted.

EventObject

Property Name	Description
EventKind	The event kind of the Event Object.

Instance

Property Name	Description
Classifier	The classifier of the Instance.
Value	The value of the Instance.

Parameter

Property Name	Description
Direction	The direction of the Parameter.
Type	The type of the Parameter.
Value	The value of the parameter.

Primitive

Property Name	Description
FQName	The FQ name of the Primitive.
ID	The ID of the Primitive.
Name	The name of the Primitive.
ObjectType	The object type of the Primitive.
Parent	The IDParent of the Primitive.

PropertyObject

Property Name	Description
BoundSize	The bound size of the PropertyObject (if it is a collection).
ClassifierStereoType	The stereotype of the PropertyObject's classifier.
IsAsynchProp	A check on whether the PropertyObject is an asynchronous property.
IsCollection	A check on whether the PropertyObject is a collection.
IsOrdered	A check on whether the PropertyObject is ordered (if it is a collection).
IsTimedProp	A check on whether the PropertyObject is a timed property.
Kind	The PropertyObject's kind.
LowerValue	The PropertyObject's lower value (if it is

	a collection).
Type	The PropertyObject's type.
UpperValue	The PropertyObject's upper value (if it is a collection).
Value	The PropertyObject's value.

SignalEvent

Property Name	Description
Signal	The signal of the SignalEvent.

State

Property Name	Description
HasSubMachine	A check on whether the State is a Submachine state.

IsFinalState	A check on whether the State is a final state.
SubMachine	Get the ID of the Submachine contained by the State (if applicable).

StateMachine

Property Name	Description
HasSubMachineState	A check on whether the StateMachine has a Submachine state.
InitialState	The StateMachine's initial state.
SubMachine State	The StateMachine's Submachine State.

TimeEvent

Property Name	Description

When	The 'when' property of the TimeEvent.

Transition

Property Name	Description
HasEffect	A check on whether the transition has a valid effect.
IsDerived	A check on whether the transition is a derived transition.
IsTranscend	A check on whether the transition transcends from one StateMachine (Submachine State) to another.
IsTriggered	A check on whether the transition is triggered.
Source	The Transition's source.
Target	The Transition's target.

Trigger

Property Name	Description
AsynchDestinationState	The asynchronous destination state of the Trigger (if it is an asynchronous trigger).
DependentProperty	The ID of the property associated with the Trigger.
Event	The Trigger's event.
Name	The Trigger's name.
Type	The Trigger's type.

Vertex

Property Name	Description
IsHistory	A check on whether the vertex is a history state.
IsPseudoState	A check on whether the vertex is a pseudo state.

PseudoState Kind	The Vertex's pseudostate kind.

Call Templates From Templates

Using function calls with parameters, you can call templates from other templates, whether standard templates or user-defined templates created within your project. Also, called templates can return a value, and can be called recursively.

Examples

A call statement returning a parameter to a variable:

```
$sSource = %StateEnumeratedName($Source)%
```

A call statement to a template that has parameters:

```
%RuleTask($GUID, $index)%
```

Using the $parameter statement in the called template:

```
$GUID = $parameter1
$index = $parameter2
```

Templates support recursive calls, such as this recursive call on the template RuleTask:

```
$GUID = $parameter1
$index = $parameter2
% PI = "" %
$nul = "Initialize condition and action object"
$count = %BR_GET("RuletCount")%
% if $count == "" or $count == $index %
%ComputeRulet($GUID)%
```

```
\n
% endTemplate %
%Rulet($index)%
\n
$index = %MATH_ADD($index, "1")%
%RuleTask($GUID, $index)%
```

The Code Template Editor in MDG Development

These topics describe how you use the Code Template Editor window to create custom templates:

- Create Custom Templates
- Customize Base Templates
- Add New Stereotyped Templates

The Code Template Editor provides the facilities of the Common Code Editor, including Intelli-sense for the code generation template macros. For more information on Intelli-sense and the Common Code Editor, see the *Editing Source Code* topic.

Create Custom Templates

Enterprise Architect provides a wide range of templates that define how code elements are generated. If these are not sufficient for your purposes - for example, if you want to generate code in a language not currently supported by Enterprise Architect - you can create completely new custom templates. You can also add stereotype overrides to your custom templates; for example, you might list all of your parameters and their notes in your method notes.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates Design > Package > Transform > Transform Templates
Keyboard Shortcuts	Ctrl+Shift+P (code generation templates) Ctrl+Alt+H (MDA transformation templates)

Create custom templates using the Code Templates Editor

Step	Description
1	In the 'Language' field, click on the drop-down arrow and select the appropriate programming language.
2	Click on the Add New Custom Template button. The 'Create New Custom Template' dialog displays.
3	In the 'Template Type' field, click on the drop-down arrow and select the appropriate modeling object. The '<None>' option requires special treatment; it enables the definition of a function macro that doesn't actually apply to any of the types, but must be called as a function to define variables $parameter1, $parameter2 and so on for each value passed in.
4	In the 'Template Name' field, type an appropriate name. Click on the OK button.
5	On the 'Code Templates Editor' tab, the new template is included in the 'Templates' list, with the value 'Yes' in the 'Modified' field. The template is called <Template Type>__<Template Name>.

	Note the double underscore character between the template type and template name.
6	Select the template from the Templates list and edit the contents in the Template field to meet your requirements.
7	Click on the Save button. This stores the new template, which is now available from the list of templates for use. You can also add a stereotype override to the template, if necessary.

Notes

- For a custom language, you must define the File template so that it can call the Import Section, Namespace and Class templates, and any other templates that you decide are applicable

Customize Base Templates

Enterprise Architect provides a wide range of templates that define how code elements are generated. If you want to change the way a code element is generated, you can customize the appropriate existing system-provided templates. Your changes might be to the effect of the template itself, or to its calls to other templates. You can also add stereotype overrides to your customized templates; for example, you might list all of your parameters and their notes in your method notes.

When you customize a system-provided (base) template, you effectively create a copy of the template that is used in preference to the original. All subsequent changes are to that copy, and the original base template is hidden. If you subsequently delete the copy it can no longer override the original, which is then brought into use again.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates
Keyboard Shortcuts	Ctrl+Shift+P

Customize a base template

Step	Description
1	On the Code Template Editor, in the 'Language' field, click on the drop-down arrow and select the programming language for which you want to customize the base templates.
2	In the Templates list, click on the base template to edit.
3	Update the template.
4	Click on the Save button to store your changes.
5	Repeat steps 2 to 4 for each of the relevant base templates you want to customize.
6	If you prefer, add one or more stereotype overrides to any of the templates.

Add New Stereotyped Templates

Sometimes it is useful to define a specific code generation template for use with elements of a given stereotype. This enables different code to be generated for elements, depending on their stereotype. Enterprise Architect provides some default templates, which have been specialized for commonly used stereotypes in supported languages. For example, the 'Operation Body' template for C# has been specialized for the property stereotype, so that it automatically generates its constituent 'get' and 'set' methods. You can override the default stereotyped templates as described in the *Override Default Templates* topic. Additionally, you can define templates for your own stereotypes, as described here.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates
Keyboard Shortcuts	Ctrl+Shift+P

Add a new stereotyped template using the Code Template Editor

Step	Description
1	Select the appropriate language, from the Language list.
2	Select one of the base templates, from the Templates list.
3	Click on the 'Add New Stereotyped Override' button. The 'New Template Override' dialog displays.
4	Select the required Feature and/or Class stereotype. Click on the OK button.
5	The new stereotyped template override displays in Stereotype Overrides list, marked as modified.
6	Make the required modifications in the Code Templates Editor.
7	Click on the Save button to store the new stereotyped template in the project file. Enterprise Architect can now use the stereotyped template, when generating code for elements of that stereotype.

Notes

- Class and feature stereotypes can be combined to provide a further level of specialization for features; for example, if properties should be generated differently when the Class has a stereotype MyStereotype, then both property and MyStereotype should be specified in the New Template Override dialog

Override Default Templates

Enterprise Architect has a set of built-in or default code generation templates. The Code Templates Editor enables you to modify these default templates, hence customizing the way in which Enterprise Architect generates code. You can choose to modify any or all of the base templates to achieve your required coding style.

Any templates that you have overridden are stored in the project file. When generating code, Enterprise Architect first checks whether a template has been modified and if so, uses that template. Otherwise the appropriate default template is used.

Access

Ribbon	Develop > Source Code > Options > Edit Code Templates
Keyboard Shortcuts	Ctrl+Shift+P

Reference

Override a default code generation template using the Code Templates Editor.

When generating code, Enterprise Architect now uses the overriding template instead of the default template.

Field/Button	Description
Language	Select the appropriate language from the list.
Templates	Select one of the base templates from the list.
Stereotype Overrides	If the base template has stereotyped overrides, you can select one of these from the list.
<Other fields>	Make any other modifications required.
Save	Click on this button to store the modified version of the template to the project file. The template is marked as modified.

Grammar Framework

Enterprise Architect provides reverse engineering support for a number of popular programming languages. However, if the language you are using is not supported, you can write your own grammar for it, using the in-built Grammar Editor. You can then incorporate the grammar into an MDG Technology to provide both reverse engineering and code synchronization support for your target language.

The framework for writing a grammar and importing it into Enterprise Architect is the direct complement to the Code Template Framework. While code templates are for converting a model to a textual form, grammars are required to convert text to a model. Both are required to synchronize changes into your source files.

An example language source file and an example Grammar for that language are provided in the Code Samples directory, which you can access from your installation directory (the default location is C:\Program Files\Sparx Systems\EA). Two other grammar files are also provided, illustrating specific aspects of developing Grammars.

Components

Component	Description
Grammar Syntax	Grammars define how a text is to be broken up into a structure, which is

	necessary when you are converting code into a UML representation. At the simplest level, a grammar is instructions for breaking up an input to form a structure. Enterprise Architect uses a variation of Backus–Naur Form (nBNF) to include processing instructions, the execution of which returns structured information from the parsed results in the form of an Abstract Syntax Tree (AST), which is used to generate a UML representation.
Grammar Editor	The Grammar Editor is an in-built editor that you can use to open, edit, validate and save grammar files.
Grammar Debugging	You can debug the grammar files you create using two facilities: • The Parser, which generates the AST for the Grammar • The Profiler, which also parses the Grammar and generates the AST but which exposes the Profiling pathway to show exactly what happened at each step of the process

Grammar Syntax

Grammars define how a text is to be broken up into a structure, which is exactly what is needed when you are converting code into a UML representation. At the simplest level, a grammar is just instructions for breaking up an input to form a structure. Enterprise Architect uses a variation of Backus–Naur Form (BNF) to express a grammar in a way that allows it to convert the text to a UML representation. What the grammar from Enterprise Architect offers over a pure BNF is the addition of processing instructions, which allow structured information to be returned from the parsed results in the form of an Abstract Syntax Tree (AST). At the completion of the AST, Enterprise Architect will process it to produce a UML model.

Syntax

Syntax	Detail
Comments	Comments have the same form as in many programming languages. // You can comment to the end of a line by adding two /s. /* You can comment multiple lines by adding a / followed by a *. The comment is ended when you add a *

	followed by a /. */
Instructions	Instructions specify the key details of how the grammar works. They are generally included at the top of the grammar, and resemble function calls in most programming languages.
Rules	Rules make up the body of a grammar. A rule can have one or more definitions separated by pipe delimiters (|). For a rule to pass, any single complete definition must pass. Rules are terminated with the semi-colon character (;).
Definitions	A definition is one of the paths a rule can take. Each definition is made up of one or more terms.
Definition Lists	A definition list corresponds to one or more sets of terms. These will be evaluated in order until one succeeds. If none succeed then the containing rule fails. Each pair of definitions is separated by a | character. This is a simple rule with three definitions: <greeting> ::= "hello" | "hi" | ["good"] "morning";

Terms	A term can be a reference to a rule, a specific value, a range of values, a sub-rule or a command.
Commands	Like instructions, commands resemble function calls. They serve two main purposes: • To process tokens in a specific way or • To provide a result to the caller

Grammar Instructions

Instructions specify the key details of how the grammar works. They are generally included at the top of the grammar, and resemble function calls in most programming languages.

Instructions

Instruction	Description
caseSensitive()	One of these two instructions is expected to specify if token matching needs to be case sensitive or not. For example, languages in the BASIC family are case insensitive while languages in the C family are case sensitive.
caseInsensitive()	
delimiters(DelimiterRule: Expression)	The delimiters instruction tells the lexical analyzer which rule to use for delimiter discovery. Delimiters are used during keyword analysis, and can be defined as the characters that can be used immediately before or after language keywords.

lex(TokenRule: Expression)	The lex instruction tells the lexical analyzer the name of the root rule to use for its analysis.
parse(RootRule: Expression) parse(RootRule: Expression, SkipRule: Expression)	The parse instruction tells the parser the name of the root rule to use for its processing. The optional second argument specifies a skip (or escape) rule, which is generally used to handle comments.

Grammar Rules

Rules are run to break up text into structure. A rule is made up of one or more definitions, each of which is made up of one or more terms.

Types of Rule

Rule	Description
Named rules	A name, followed by a definition list. For example: <rule> ::= <term1> <term2> | "-" <term1>;
Inline Rules	Inside a definition, a rule defined within parentheses. These act in exactly the same way as if they were a named rule being called by a term. For example: <rule> ::= (<inline>);
Optional Rules	Inside a definition, a rule defined within square brackets. This rule succeeds even if the contents fail. For example: <rule> ::= [<inline>];
Repeating Rules	Inside a definition, a term followed by a plus sign. This rule matches the inner rule

	once or more than once. For example: <rule> ::= <inline>+; rule ::= (<term1> <term2>)+;
Optional Repeating Rules	Inside a definition, a rule followed by a star. This rule matches the inner rule zero or more times, meaning it succeeds even if the inner rule never succeeds. For example: <rule> ::= <inline>*; rule ::= (<term1> <term2>)*;

Grammar Terms

Terms identify where tokens are consumed.

Types of Term

Type	Description
Concrete terms	Quoted strings. For example, "class"
Unicode characters	A lexer-only term, having the prefix of U+0x followed by a hexadecimal number. For example: U+0x1234
Ranges	A lexer-only term, matching any character between the two characters specified. For example, "a".."z" or U+0x1234..U+2345
References	The name of another rule, in angled brackets. The token will match if that rule succeeds. For example, <anotherRule>

Commands	A call to a specific command.

Grammar Commands

Commands, like Instructions, resemble function calls. They serve two main purposes:

- To process tokens in a specific way or
- To provide a result to the caller

Commands

Command	Description
attribute(Name: String, Value: Expression)	Creates an attribute on the current AST node. The attribute will be created with the Name specified in the grammar source, and will be given the value of all tokens consumed as a part of executing the Value expression. This command produces the AST node attributes that Enterprise Architect operates on in code engineering.
attributeEx(Name: String) attributeEx(Name: String, Value: String)	Creates an attribute on the current AST node without consuming any tokens. The attribute will be created with the same name as is specified in the grammar source, and with either an empty value or the value specified by the optional Value argument.

	This command produces the AST node attributes that Enterprise Architect operates on in code engineering.
except(Target : Expression, Exception: Expression)	Consumes input data that matches the Target expression, but fail on data that matches the Exception expression. This operates somewhat similar to, but exactly the opposite of, the skip command.
fail()	Causes the parser to fail the current rule, including any remaining definitions.
keywords()	Matches any literal string used as a grammar term; that is, if you enter an explicit string that you are searching for, it becomes a key word.
mapLeft(Target: Expression, Common: Expression)	Consumes input data that matches the Target expression, then consumes input data that matches the Common expression. The AST Nodes and Attributes generated from the Common expression are then copied into each of the top level nodes generated by the Target Expression. Both Target and Common expressions

	need to successfully parse for this command to succeed.
mapRight(Common: Expression, Target: Expression)	Consumes input data that matches the Common expression, then consumes input data that matches the Target expression. The AST Nodes and Attributes generated from the Common expression are then copied into each of the top level nodes generated by the Target Expression. Both Common and Target expressions need to successfully parse for this command to succeed.
node(Name: String, Target: Expression)	Creates an AST node under the current AST node (the nodes that Enterprise Architect operates on in code engineering). The node will be created with the Name specified in the grammar source.
preProcess(Target: Expression)	Evaluates an expression and uses that pre-processed data in multiple definitions. This is most useful within expression parsing, where the same left hand side expression will be evaluated

	against a number of operators. This command reduces the work the parser must do to make this happen.
skip(Target: Expression) skip(Target: Expression, Escape: Expression)	Consumes input data (characters when lexing, and tokens when parsing) until the 'Target' expression is matched. The optional 'Escape' expression can be used to handle instances such as escaped quotes within strings.
skipBalanced (Origin: Expression, Target: Expression) skipBalanced (Origin: Expression, Target: Expression, Escape: Expression)	Consumes input data (characters or tokens) until the 'Target' expression is matched and the nesting level reaches zero. If the 'Origin' expression is matched during this process, the nesting level is increased. If the 'Target' expression is matched, the nesting level is decreased. When the nesting level reaches zero, the command exits with success. An optional 'Escape' expression can be provided.
skipEOF()	Consumes all remaining data (characters or tokens) until the end of the file.
token(Target: Expression)	Creates a token during lexical analysis for processing during parsing. The value of

	the token will be the value of all characters consumed as a result of executing the Target expression.
warning()	Inserts a warning into the resulting AST.

AST Nodes

In defining a grammar, you would use AST nodes and AST node attributes that can be recognized in code engineering in Enterprise Architect, in the AST results that are returned by the attribute, attributeEx and node commands. The nodes and attributes are identified in these tables. Any others will be ignored in code engineering.

FILE Node

The FILE node represents a file. It isn't mapped to anything, but contains all the required information.

Multiplicity / Nodes	Description
0..* / PACKAGE	See *PACKAGE Node*.
0..* / CLASS	See *CLASS Node*.
0..* / IMPORT	The node to represent the imported namespace/Package or equivalent. The 'NAME' attribute of the node will be the name of imported namespace/Package or equivalent.
0..* /	Field labels as part of a skip rule will be

COMMENT	at the root level; the code generator looks for comments of this sort by position relative to the node.
0..1 / INSERT_POSITION	This gives the position where new Classes, Packages and method implementations can be inserted into the file. If it is not found, the code generator will automatically insert new items immediately after the last one is found in code.

PACKAGE node

The PACKAGE node corresponds to a namespace or equivalent in the file. When importing with 'package per namespace', Enterprise Architect will create a Package directly under the import for this and place all Classes within it. When not importing namespaces, Enterprise Architect will look for Classes under this point, but it will do nothing with this node.

Additionally, if you are generating with namespaces enabled (see the *Code Options* Help topics for generic languages) a generated Class will not match a Class in code unless they are under the same Package structure.

Contained in nodes: FILE

Multiplicity /	Description

Nodes	
1 / NAME	See *NAME Node*.
0..* / CLASS	See *CLASS Node*.
0..* / PACKAGE	The child Package node.
0..1 / OPEN_POSITION	Gives the position where the Package body opens. This can also be used as an insert position.
0..1 / INSERT_POSITION	Gives the position where new Classes and Packages can be inserted into the file. If it is not found, the code generator will automatically insert new items immediately after the last one is found in code.
0..1 / SUPPRESS	Prevents indenting when inserting into this Package.

CLASS/INTERFACE Node

The CLASS (or INTERFACE) node is the most important in code generation. It is brought in as Class (or Interface)

Objects.
See *Class DECLARATION* and *Class BODY*.
Contained in Nodes: FILE, PACKAGE, Class BODY

CLASS Declaration

Contained in Nodes: CLASS/INTERFACE

Multiplicity / Nodes	Description
1 / NAME	See *NAME Node*.
0..* / PARENT	See *PARENT Node*.
0..* / TAG	See *TAG Node*.
0..1 / DESCRIPTION	See *DESCRIPTION Node*.
1 / NAME	The name of the Class. If there is a node NAME, that will overwrite this attribute.
0..1 / SCOPE	The UML Scope of the Class - Public, Private, Protected or Package.
0..1 /	If present, indicates that this is an abstract

ABSTRACT	Class.
0..1 / VERSION	The version of the Class.
0..1 / STEREOTYPE	The stereotype that Enterprise Architect should assign to the Class. This does not support multiple stereotypes.
0..1 / ISLEAF	If present, indicates that this is a leaf/final/sealed Class which cannot be inherited by any sub-Class.
0..1 / MULTIPLICITY	If present, represents the multiplicity of the Class.
0..1 / LANGUAGE	Generally, you do not need to set this.
0..1 / NOTE	Generally not used as it is addressed by the comments above the Class.
0..1 / ALIAS	If present, represents the Alias of any identifier, such as a Namespace, Class or variable.
0..* / MACRO	Adds a numbered Tagged Value that Enterprise Architect can use to round trip

	macros.

Class BODY Node

Contained in Nodes: CLASS/INTERFACE

Multiplicity / Nodes	Description
0..* / METHOD	See *METHOD Node*.
0..* / ATTRIBUTE	See *ATTRIBUTE Node*.
0..* / FIELD	See *FIELD Node*.
0..* / CLASS	See *CLASS Node*.
0..* / SCOPE	See *SCOPE Node*.
0..* / PROPERTY	This node represents the Property definition within the Class Body.
0..* / TAG	See *TAG Node*.

0..* / PARENT	See *PARENT Node*.
0..1 / OPEN_POSITION	Gives the position where the Class body opens. This can also be used as an insert position.
0..1 / INSERT_POSITION	Gives the position where new Class members can be inserted into the file. If it is not found, the code generator will automatically insert new items immediately after the last one is found in code.

SCOPE Node

This is an optional feature for languages resembling C++ that have Blocks that specify the scope of elements. The language needs to have a name specified that is used for the scope of all elements in the Block. In all other respects it behaves identically to the Class BODY node.

Contained in Nodes: Class BODY

Multiplicity / Nodes	Description
1 / NAME	Used as the scope for all methods and

	attributes contained within the scope.

METHOD Node

Contained in Nodes: Class BODY, SCOPE

Multiplicity / Nodes	Description
1 / Method DECLARATION	See *Method DECLARATION Node*.

Method DECLARATION Node

Contained in Nodes: METHOD

Multiplicity / Nodes	Description
0..1 / TYPE	See *TYPE Node*.
0..* / PARAMETER	See *PARAMETER Node*.
0..* / TAG	See *TAG NODE*.

0..1 / DESCRIPTION	See *DESCRIPTION Node*.
0..1 / MULTI PARAMETER	Supports Delphi's parameter list style of declaration. This is the equivalent of FIELD.
1 / NAME	The name of the method.
0..1 / TYPE	The return type of the method.
0..1 / SCOPE	The UML Scope of the method - Public, Private, Protected or Package.
0..1 / ABSTRACT	If present, indicates that the method is Abstract.
0..1 / STEREOTYPE	The stereotype that Enterprise Architect should assign to the Method. This does not support multiple stereotypes.
0..1 / STATIC	If present, indicates that the method is static.
0..1 / CONSTor	If present, indicates that the method is constant.

CONSTANT	
0..1 / PURE	If present, indicates that the method is a Pure method.
0..1 / ISQUERY	If present, indicates that the method is query/read only.
0..1 / ARRAY	If present, indicates that the method type (return type) is an array.
0..1 / SYNCHRONIZED	If present, indicates that the method is a synchronized method.
0..* / MACRO	The Macro specified in the method declaration.
0..1 / CSHARPIMPLEMENTS	Specifies special behavior for C#.
0..1 / BEHAVIOR	Provides support for Aspect J, using behavior.
0..1 / SHOWBEHAVIOR	Provides support for Aspect J, using behavior, and shows the reverse-engineered behavior on the diagram.

ATTRIBUTE Node

Contained in Nodes: Class BODY, SCOPE

Multiplicity / Nodes	Description
1 / TYPE	See *TYPE Node*.
0..* / TAG	See *TAG Node*.
0..1 / DESCRIPTION	See *DESCRIPTION Node*.
1 / NAME	The name of the Attribute.
0..1 / TYPE	The type of the Attribute.
0..1 / SCOPE	The UML Scope of the Attribute - Public, Private, Protected or Package.
0..1 / DEFAULT	The default value of the Attribute.
0..1 / CONTAINE	If present, indicates the container for the Attribute.

R or ARRAY	
0..1 / CONTAINMENT	Reference or value.
0..1 / STEREOTYPE	The stereotype that Enterprise Architect should assign to the Attribute. This does not support multiple stereotypes.
0..1 / STATIC	If present, indicates that it is a static Attribute.
0..1 / CONST or CONSTANT	If present, indicates that it is a constant Attribute.
0..1 / ORDERED	If present, indicates that the Attribute (value) is ordered.
0..1 / LOWBOUND	If present, represents the lower boundary of the Attribute value.
0..1 / HIGHBOUND	If present, represents the higher boundary of the Attribute value.
0..1 /	If present, indicates that the Attribute is

TRANSIENT or VOLATILE	Transient or Volatile.

FIELD Node

A field corresponds to multiple attribute declarations in one. Anything not defined in the Declarators but defined in the field itself will be set for each declarator. Everything supported in an attribute is supported in the field. If no declarators are found then this works in the same way as an attribute.

Contained in Nodes: Class BODY, SCOPE

Multiplicity / Nodes	Description
0..* / DECLARATOR	See *ATTRIBUTE Node*.

PARAMETER Node

Contained in Nodes: Method DECLARATION, TEMPLATE

Multiplicity /	Description

Nodes	
1 / TYPE	See *TYPE Node*.
0..* / TAG	See *TAG Node*.
0..1 / DESCRIPTION	See *DESCRIPTION Node*.
0..1 / NAME	The name of the parameter.
0..1 / TYPE	The type of the parameter.
0..1 / KIND	Expected to be in, inout, out or return.
0..1 / DEFAULT	The default value of the parameter.
0..1 / FIXED	If present, indicates that the parameter is fixed/constant.
0..1 / ARRAY	If present, indicates that the parameter type is an array.

NAME Node

Contained in Nodes: PACKAGE, Class DECLARATION

Multiplicity / Nodes	Description
1 / NAME	The name portion.
0..* / QUALIFIER	The qualifier portion.
0..* / NAMEPART	An alternative to using NAME and QUALIFIER. A string of values, all except the last one taken as qualifiers. The last one is taken as the Name.

TYPE Node

Contained in Nodes: Method DECLARATION, ATTRIBUTE, PARAMETER

Multiplicity / Nodes	Description
0..1 / TEMPLATE	The entire text of the template is the name of the type. Only used if NAME is undefined. See *TEMPLATE Node*.

1 / NAME	The name portion.
0..* / QUALIFIER	The qualifier portion.
0..* / NAMEPART	An alternative to using NAME and QUALIFIER. A string of values, all except the last one taken as qualifiers. The last one is taken as the Name.

TEMPLATE Node

Contained in Nodes: TYPE

Multiplicity / Nodes	Description
0..* / PARAMETER	See *PARAMETER Node*.
1 / NAME	

PARENT Node

Contained in Nodes: Class DECLARATION

Multiplicity / Nodes	Description
0..1 / TYPE	Has the value Parent, Implements or VirtualP.
1 / NAME	The name portion of the Parent.
0..* / QUALIFIER	The qualifier portion of the Parent.
0..* / NAMEPART	An alternative to using NAME and QUALIFIER. A string of values, all except the last one taken as qualifiers. The last one is taken as the Name.
0..1 / INSTANTIATION	If present, indicates the instantiation of a template parameter.

TAG Node

Contained in Nodes: Class DECLARATION, Method DECLARATION, ATTRIBUTE, PARAMETER

Multiplicity / Nodes	Description

1 / NAME	The name of the Tagged Value (the Tag).
0..* / VALUE	The value of the Tagged Value.
0..1 / MEMO	If present, indicates that the type of the Tagged Value is <memo>.
0..1 / NOMEMO	If present, indicates that the type of the Tagged Value is not <memo>.
0..1 / GROUP	If present, indicates that the value is a Tagged Value group.

DESCRIPTION Node

Contained in Nodes: Class DECLARATION, Method DECLARATION, ATTRIBUTE, PARAMETER

Multiplicity / Nodes	Description
0..* / VALUE	The text that Enterprise Architect should assign to the Note.

Editing Grammars

If you need to write and edit a grammar for code imported in a new programming language, you can do so using the built-in Grammar Editor.

Access

Ribbon	Develop > Source Code > Grammar Editor

Create and Edit Grammar

Field/Button	Action
Open Grammar	Display a browser through which you can locate and open the file containing the grammar you want to edit.
Recent	Recently used grammars can be quickly accessed using this combo box.
Save	Save the current file.
Save As	Saves a copy of the current file

Validate Grammar	The grammar validation will run a series of tests on the current grammar to ensure its validity. Errors and warnings will be displayed informing you of both errors that will make the grammar unusable, and conditions where you might get unexpected results.
Help	Display this Help topic.

Context Menu Options

Field/Button	Action
Open File	Display a browser through which you can locate and open the file containing the grammar you want to edit.
Validate	The grammar validation will run a series of tests on the current grammar to ensure its validity. Errors and warnings will be displayed informing you of both errors that will make the grammar unusable, and conditions where you might get unexpected results.

Language	The Grammar Editor defaults to normal Backus–Naur Form (nBNF). The mBNF option is also available.
Line Numbers	Turn line numbers on or off in the grammar editor.

Parsing AST Results

The Abstract Syntax Tree (AST) is the code that Enterprise Architect sees as it processes a grammar.

You parse the text in the bottom half of the Grammar Editor window and review what is displayed as a result. You can either open a file or paste text in. If you have pasted text that corresponds to something that cannot appear at the file level (such as Operation Parameters) you can select an alternative rule to use as a starting point. The parse will then commence from that rule.

Access

Ribbon	Develop > Source Code > Grammar Editor > Grammar Debugger > AST Results

Toolbar Options

Option	Action
Open File	Open a sample input file to test against.
Recent	Recently opened source files can be

	selected from this combo box.
Parse	Perform the parse operation. If the parse is successful, the 'AST Results' tab will contain the resulting AST.
Select Rule	This drop down allows you to select an alternative root rule for processing your sample source.
Help	Display this Help topic.

Profiling Grammar Parsing

When you parse a grammar that you have created, it might show errors that you cannot immediately diagnose. To help you resolve such errors, you can review the process that the parser followed to generate the AST you can see, using the Grammar Profiler.

You again parse the text in the bottom half of the Grammar Editor window, but this time the tree shows each rule that the parser attempted, where it got to and if it passed or not. Rules for opening a file, pasting a file and setting the starting rule remain the same.

Access

Ribbon	Develop > Source Code > Grammar Editor > Grammar Debugger > Profiler Results

Toolbar Options

Option	Action
Open File	Display a browser through which you can locate and open the file containing the

	grammar you want to edit.
Parse	Perform the parse operation. If the parse is successful, the 'AST Results' tab will contain the resulting AST, and the 'Profile Results' tab will contain debug information regarding the path that the parser took through your grammar. The profile data is extremely useful when debugging a new grammar.
Select Rule	If you want to use a different root rule for processing your sample source, click on the drop-down arrow and select the alternative rule.
Help	Display this Help topic.

Notes

- Because profiling can take a very long time for large files, the 'Profile Results' tab is not filled if you are not displaying that tab when you begin parsing

Macro Editor

The macro editor allows a user to supplement the grammar with a list of keywords and rules to exclude macros during grammar parse operations. The macro definition list is particularly useful when developing grammars for languages that support macros such as C++. It avoids the necessity of describing these rules in the grammar itself, and can be used with multiple grammars.

This feature is available from Enterprise Architect Release 14.1.

Access

Ribbon	Develop > Source Code > Grammar Editor > Macro Editor

Editing Macros

Open File	Open an existing macro definition list
Recent	Recently opened macro definition lists can be selected from this combo box

Save	Saves changes to the opened macro definition list
Save As	Saves a copy of the existing macro definition list
Validate	Validates the grammar of the macro definition list

Example Grammars

The Code Samples directory set up by the Enterprise Architect installer contains an example Grammar that you can load into the Grammar editor to review, and into the Grammar Debugger to parse and profile.

The Grammar example consists of two files:

- test.ssl - a simple sample language source file, in the style of C, and
- ssl.nbnf - a grammar for the simple sample language

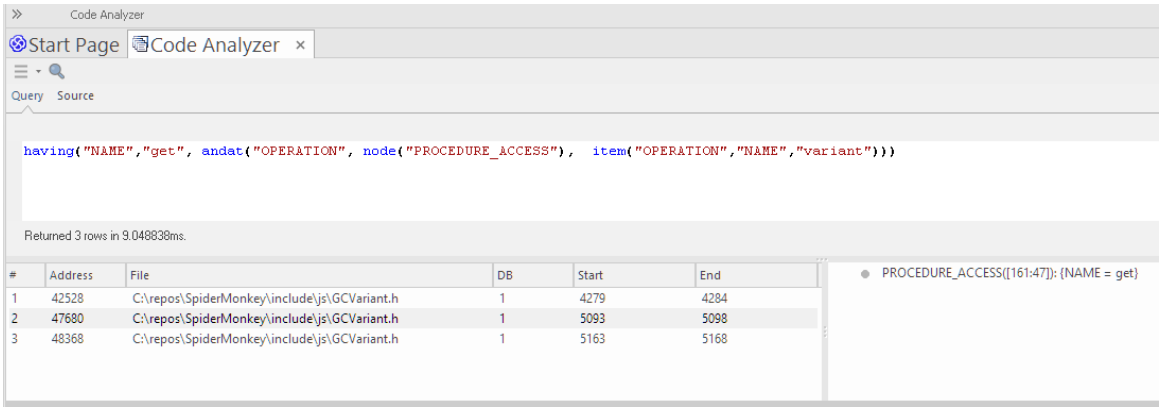
The example illustrates:

- Tokenization (using the Lexer)
- Creation of a Package
- Creation of a Class or Interface
- Creation of an attribute
- Creation of an operation (with parameters)
- Importing comments

The Code Samples directory also contains two other Grammar files that you can examine:

- Expressions Sample.nBNF - this illustrates how expression parsing is set up and processed, with detailed comment text providing explanations
- CSV Sample.nBNF - an example grammar for processing CSV files

Code Analyzer



The Code Analyzer is an essential tool for anyone who deals with source code every day.

It can perform very complex queries on source code repositories at lightning speed either locally or on a Sparx Intel cloud service. The queries are composed using a high level language developed by Sparx System. The language uses a small but expressive vocabulary that is easily learned and permits code metrics to be queried much faster than conventional methods.

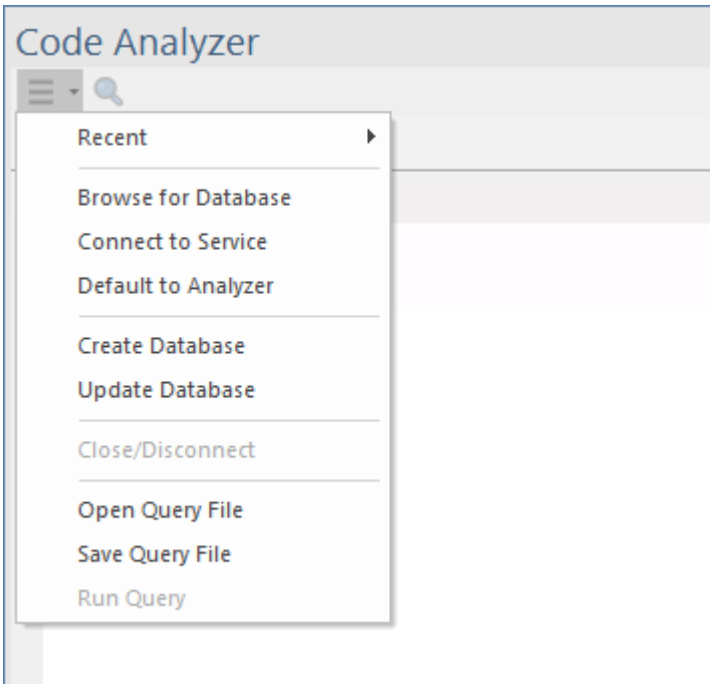
Access

Ribbon	Develop > Source Code > Code Analyzer

Code Analyzer Menu

The Code Analyzer menu is displayed when you click on

the icon in the top-left corner of the window.



The menu provides various commands for activities associated with the use of the Code Analyzer, including such things as choosing a Code Miner database to use, updating the Code Miner database and Opening a Query File for editing.

This table describes each of the menu commands.

Command	Description
Recent	Displays a sub-menu that provides a list of recent connections to services and local database files.
Browse for Database	Displays a 'file chooser' dialog, allowing you to browse for a Code Miner database on your machine.

Connect to Service	Displays the 'Code Miner Database Connection' dialog, in which you specify connection details for a (list of) Code Miner Database services.
Default to Analyzer	Selecting this option results in the Code Analyzer automatically connecting to the Code Miner service configured for the active Execution Analyzer Script, when the Code Analyzer is started.
Create Database	Displays the 'Create Code Miner Database' dialog, which allows you to create a Code Miner database from a source code repository in the file system.
Update Database	Displays the 'Code Miner Database Update' dialog, which allows you to perform an incremental update to an existing Code Miner database, to incorporate recent changes to source code files.
Close/Disconnect	Closes or disconnects from the Code Miner Database library or service.
Open Query File	Shows a 'file open' dialog allowing you to choose an mFQL query file from the file system.

Save Query File	Shows a 'file save' dialog allowing you to save the current mFQL query to a named file.
Run Query	Runs the entire query or selected contents of the query entered in the 'Query' tab editor. Shortcut F6.

Before Using the Analyzer

Before you can use the Code Analyzer, you must first create a Code Miner database or locate an existing one that the Code Analyzer can access. Creating a Code Miner database is summarized here, or you can read a detailed description in the Help topic *Creating a New Code Miner Database*.

Depending on the location of the library you will be using, you should either:

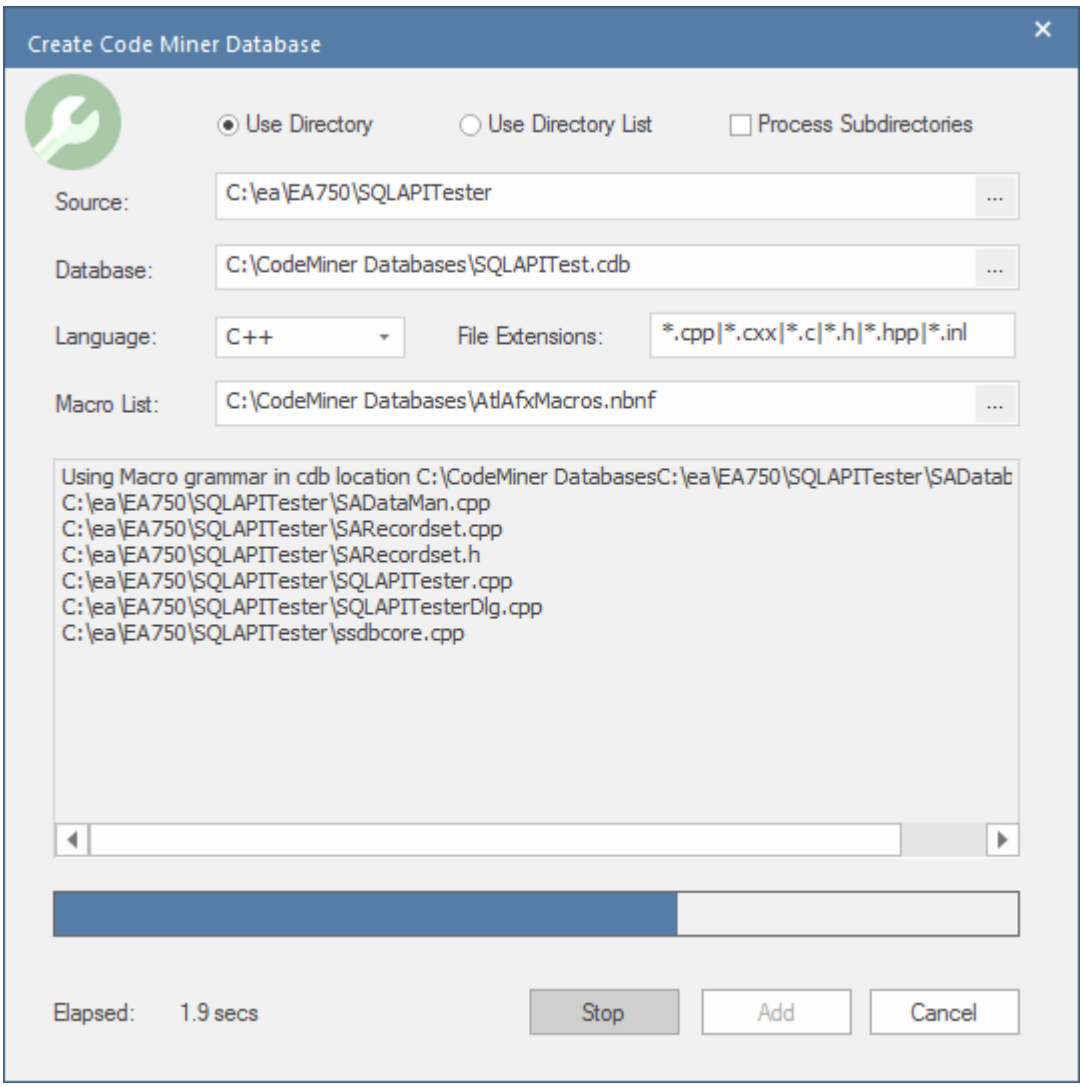
- Select a Code Miner library file to use, or
- Connect to a service that is hosting a Code Miner database.

Once you have completed these steps, you are ready to begin writing and running queries in the Code Analyzer.

Creating a Code Miner Database

Code Miner databases are built from source code repositories. The process is similar to code compilation, using the language grammar to analyze individual files.

There are two types of build - full and incremental. The initial full build might take some time, but the subsequent incremental builds are incredibly quick.



Using a Directory as input

You can select a single folder as the root of the source code you want to compile. With this option you can choose to include subdirectories

Using a Directory List

Sometimes, you want to use more than a single project, but not all the projects are under a single directory. In this case, you can create a text file that lists the full path to each folder you want to include and you specify that text file in the 'Source' field. Each directory path should be listed on a separate line.

c:\myprojects\project1\tools\scintilla

c:\myprojects\project2\src

d:\mylibs\lib1\src

If you want to recursively process the sub-directories within a directory, precede the path with an exclamation mark like this:

!d:\mylibs\lib1\src

Any line that begins with a # character is treated as a comment.

include scintilla

c:\myprojects\project1\tools\scintilla

Language

In this field, you specify the language used in the source code from which this Code Miner database is being built.

Available languages are: C++, C#, Java, XML,

MDGTechnology and Custom.

Macro List

When the language selected is 'C++', the 'Macro List' selection field is displayed . For C++, the success and depth of information compiled into the database can be inextricably linked to the use of macros. This field can be used to select an nBNF macro file that will be used as an auxiliary grammar component for the compilation.

By default the macro file will default to the macro file in the Enterprise Architect installation folder. You are free to modify or extend the content of this file to suit your requirements - for example, when you need to correct errors reported in the compilation log file.

Grammar

Sparx Systems has developed grammars for all of the languages listed in the drop-down selection list; C++, C#, Java, XML and also MDGTechnology. For these languages a built-in grammar file is used.

There is also an option to select a 'Custom' language. When 'Custom' is selected, the 'Grammar' field is displayed. This field is used to specify a file containing the grammar for your custom language. The Code Miner will then use that grammar to parse the source code written in that language.

Users that develop a Custom language, will need to specify grammar rules for that language and save them into an nBNF file. Enterprise Architect's Grammar Editor is

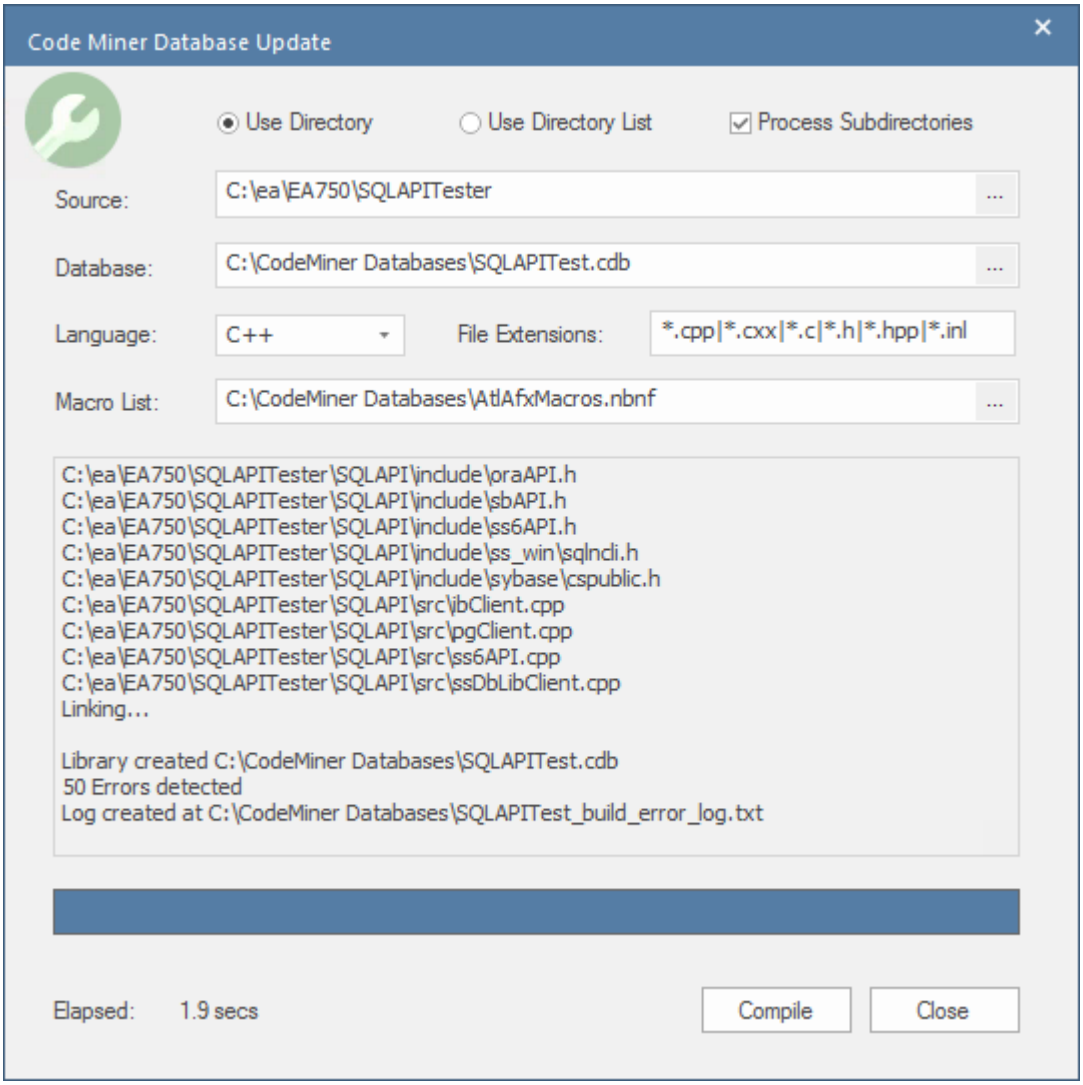
designed specifically for that purpose.

The Help Topic *Grammar Framework* provides detailed information on writing an nBNF grammar.

Updating a Code Miner Database

From time to time, you will want to update your Code Miner database. Typically, when you have made changes to your source code, but also after updating a grammar file or extending a macro file.

The process to update a database is very similar to creating a new database, but faster because you are not starting from scratch. Simply choose the menu option 'Update Database'. The 'Code Miner Database Update' dialog will display. The input fields will be populated with values from the last build. Proceed as for 'Creating a Code Miner Database'.



Selecting a Code Miner Database File

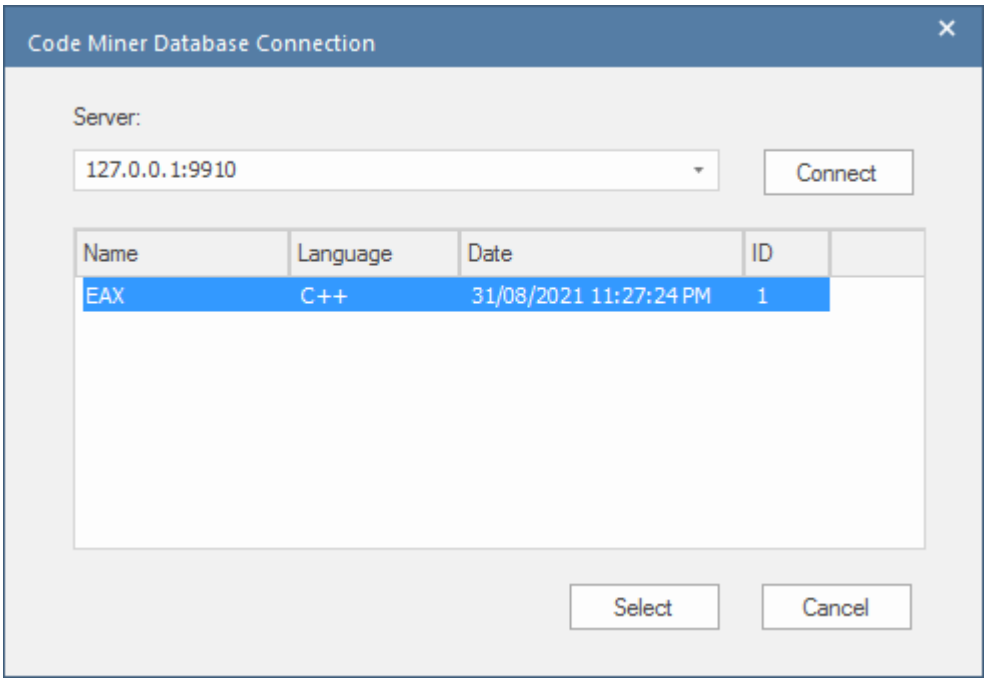
If you choose to use a library file for your Code Miner database, choose the menu option 'Browse for Database'. This will display a 'File Chooser', where you can browse for and select a *.cdb file.

Connecting to a Service

When connecting to a service, the dialog lists all databases

hosted by the service.

You can choose to select an individual database in the list, or simply click the Select button, in which case queries will be executed across all databases listed by the service.

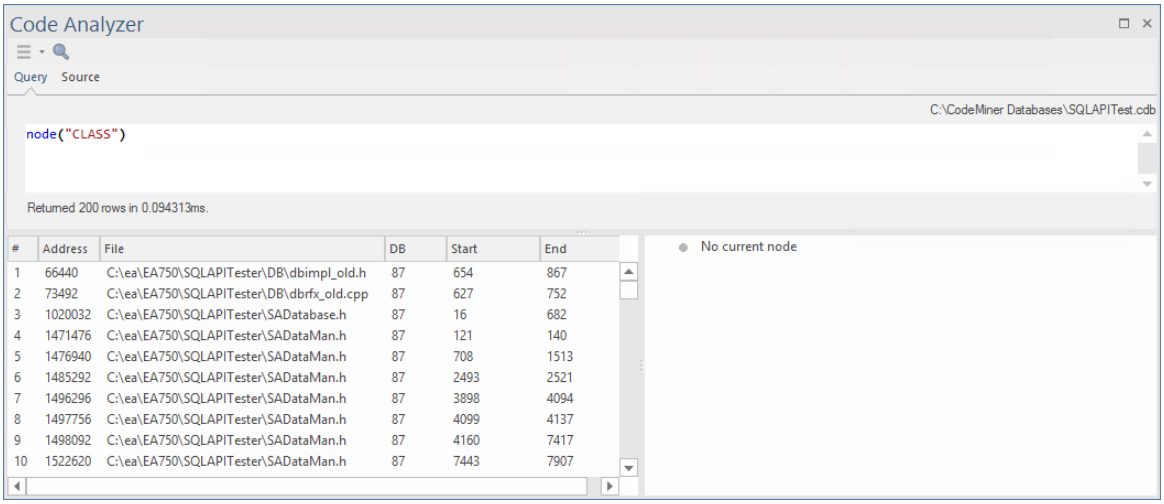


Running Queries

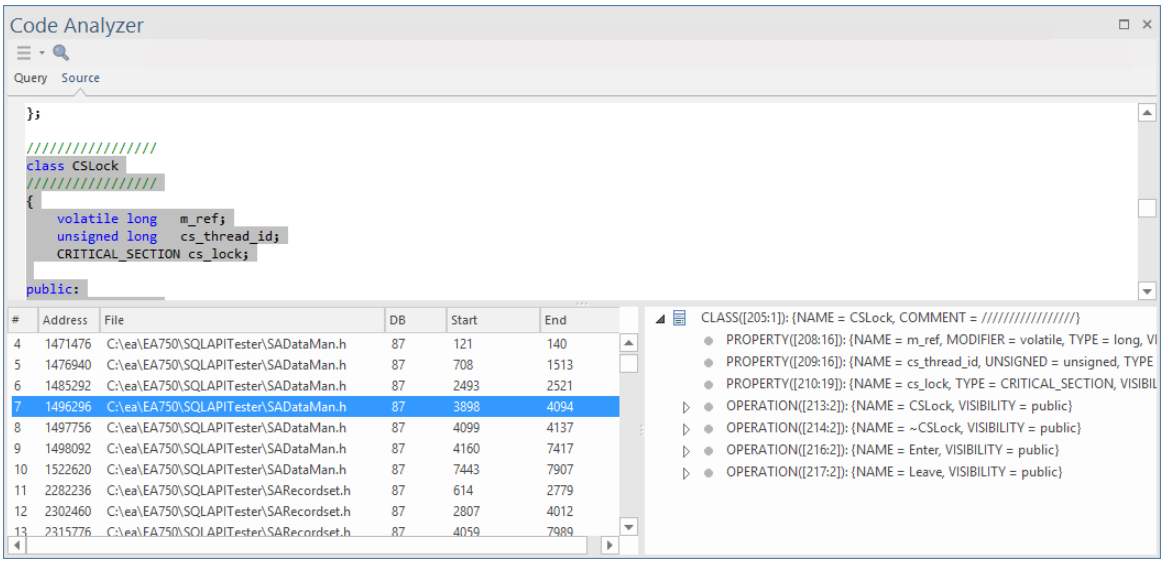
Once you have connected to a Code Miner database, you are ready to start running queries.

To run a query, select the Query tab in the Code Analyzer window, type in your query, then click on the icon to execute the query.

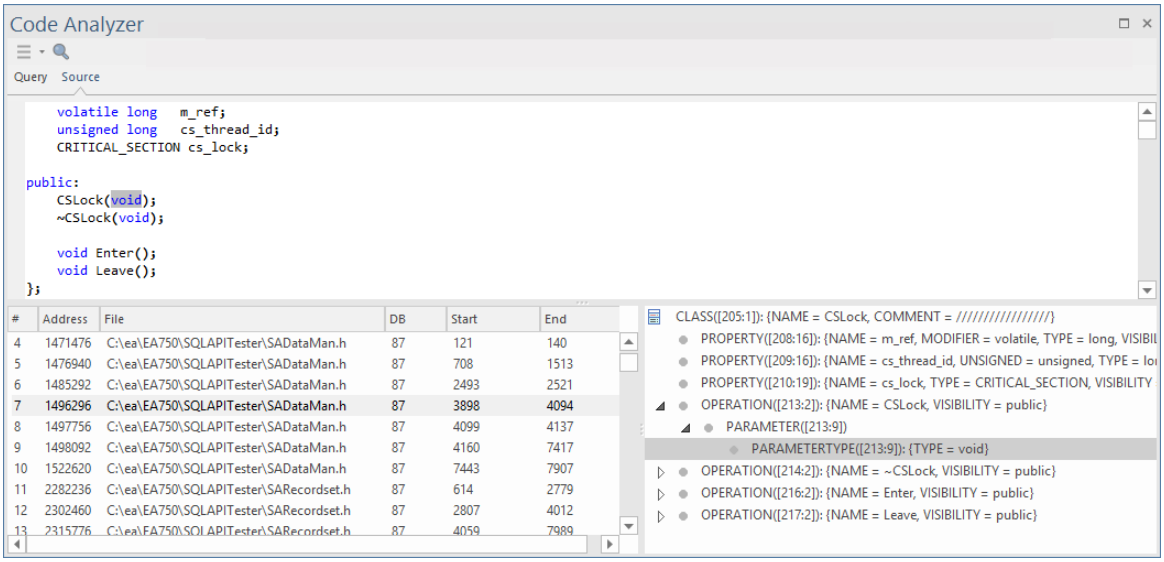
In this example, we have run a simple query node("CLASS"), which will return all 'Class' nodes found in the Code Miner database.



By selecting a result in the lower-left panel, the 'Source' tab is activated and displays the source code corresponding to the selected node. Details for that class node are displayed in the lower-right panel.



Selecting a detail item in the lower-right panel, results in narrowing the selection within the source code, as shown here.



Query Example - Intersection

As an example, this mFQL query finds all the classes that have an operation named GetOption.

```
andat( "CLASS", item("OPERATION", "NAME", "GetOption"), node("CLASS"))
```

This clause returns a set of operations for which the 'NAME' value is "GetOption":

```
item("OPERATION", "NAME", "GetOption")
```

This clause returns a set of all Class nodes:

```
node("CLASS")
```

Formal syntax:

```
andat( string:rule, set:left, set:right)
```

'andat' takes the set of operations (left), applies the rule "CLASS" (only include rows that have a CLASS parent), then intersects that set with the set of all known classes (right). If the intersection succeeds, the operation node is added to the result set, otherwise it is excluded.

The Query Language - mFQL

The query language used with the Code Analyzer is described in full, in the *Code Miner Query Language (mFQL)* Help topic.

A brief description and some examples are also presented here.

The mFQL language is based on sets. Each statement works using the various types of set operations of which there are only a few.

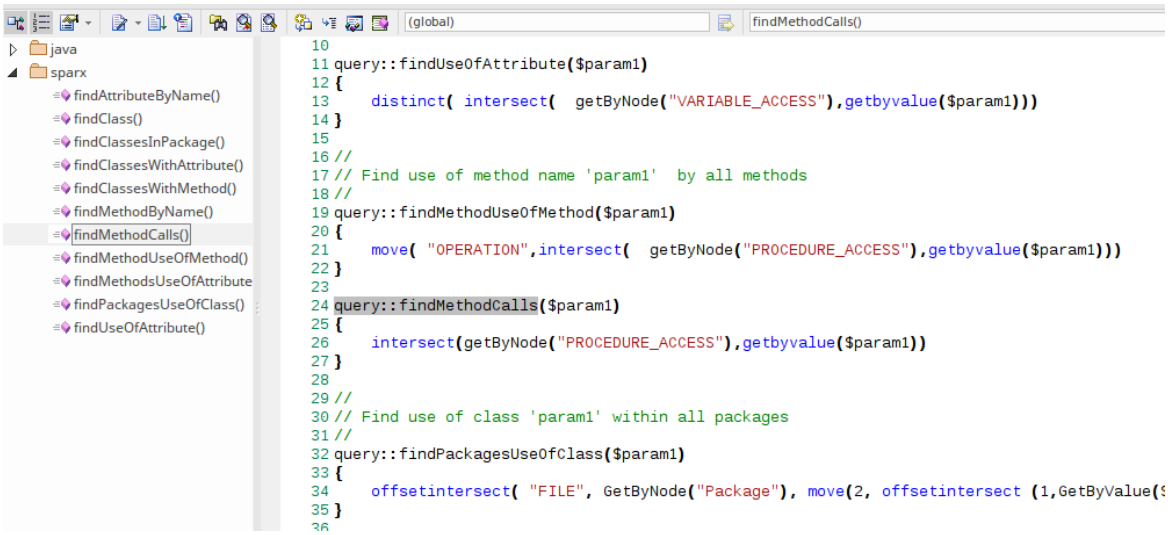
Code Miner Framework

The Code Miner system provides fast and comprehensive access to the information in existing source code. By parsing all source code and storing the resulting Abstract Syntax Tree in a read-optimized database, the system provides complete access to all aspects of the original source code, in a machine understandable format.

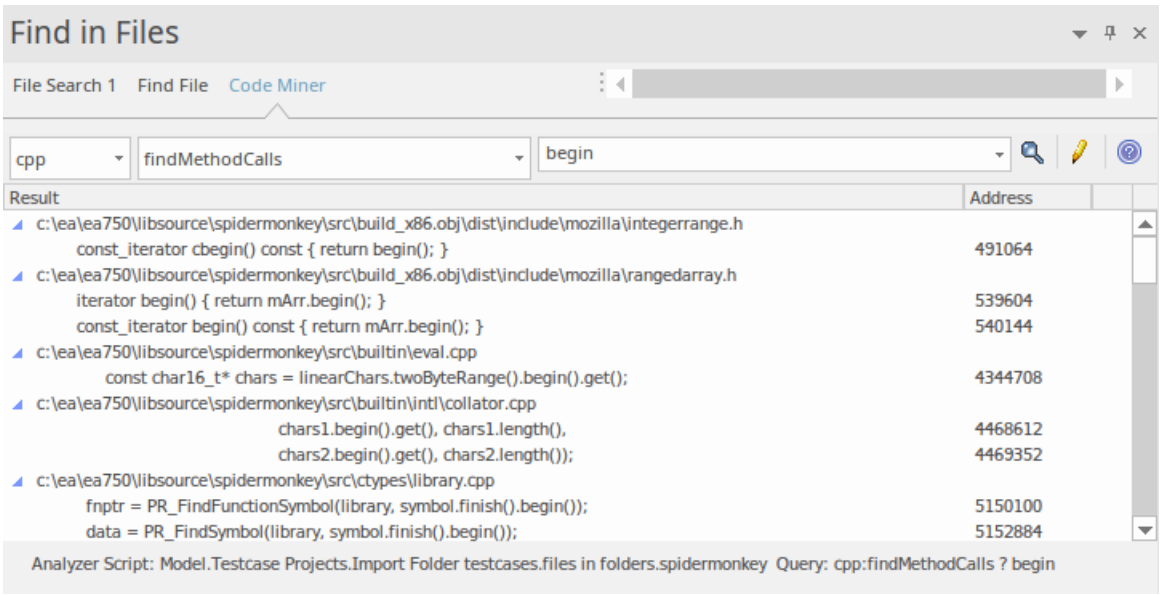
The core goal behind the system is to provide access to the data hidden within source code in a timely and effective manner. Great pains have been taken to ensure maximum performance, while providing the simplest interfaces possible. As a result the system can be used to analyze program structure, calculate metrics, trace relationships and even perform refactoring.

Information from Code Miner databases is retrieved using queries written in Code Miner NBNF Query Language (mFQL), Code Miner's own language. The language itself is reasonably simple, providing a small number of commands. Simple as the language is, it supports queries of arbitrary size and complexity. The design provides extreme performance for all queries, great and small.

This feature is available from Enterprise Architect Release 14.1.



Enterprise Architect's Code Analyzer, its search tools and the Intelli-sense features of its code editors all make use of the information mined from these databases.

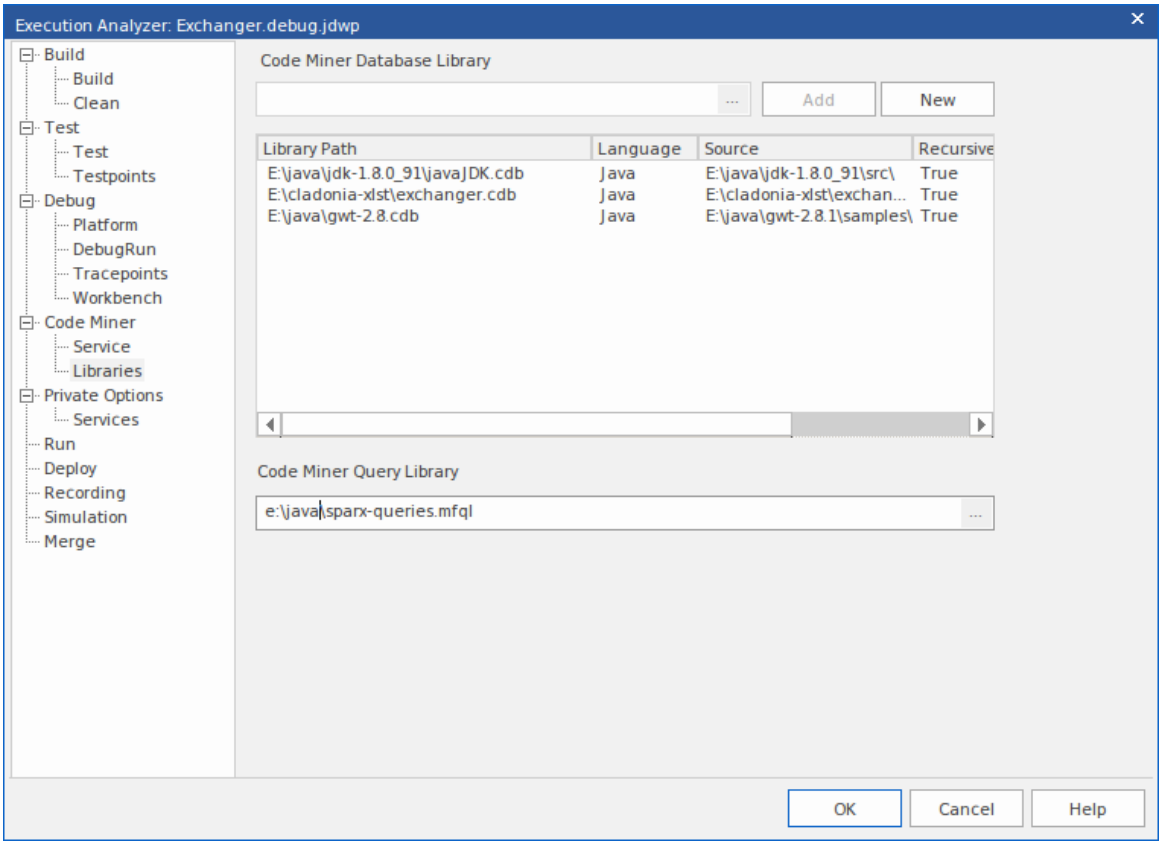


The currently active Analyzer Script, and also the query parameters, are indicated across the bottom of the 'Code Miner' page of the search tool.

Code Miner Libraries

Code Miner libraries are managed in Enterprise Architect using the Analyzer Script Editor. These Libraries are a collection of Code Miner databases, one of which would normally exist for each framework or project. The Analyzer Script Editor allows new databases to be created, and existing databases to be added, updated or removed. Together, these databases form the Code Miner Library used by the Code Analyzer and Intelli-sense features of Enterprise Architect. The library can be used locally, or it can be deployed to a server location where it can service multiple clients. You select the scenario to use on the 'Sparx Intel Service' page of the Analyzer Script.

This feature is available from Enterprise Architect Release 14.1.



Access

On the Execution Analyzer window, locate and double-click on the required script - the script editor dialog will display. On that dialog, select the 'Code Miner > Libraries' page.

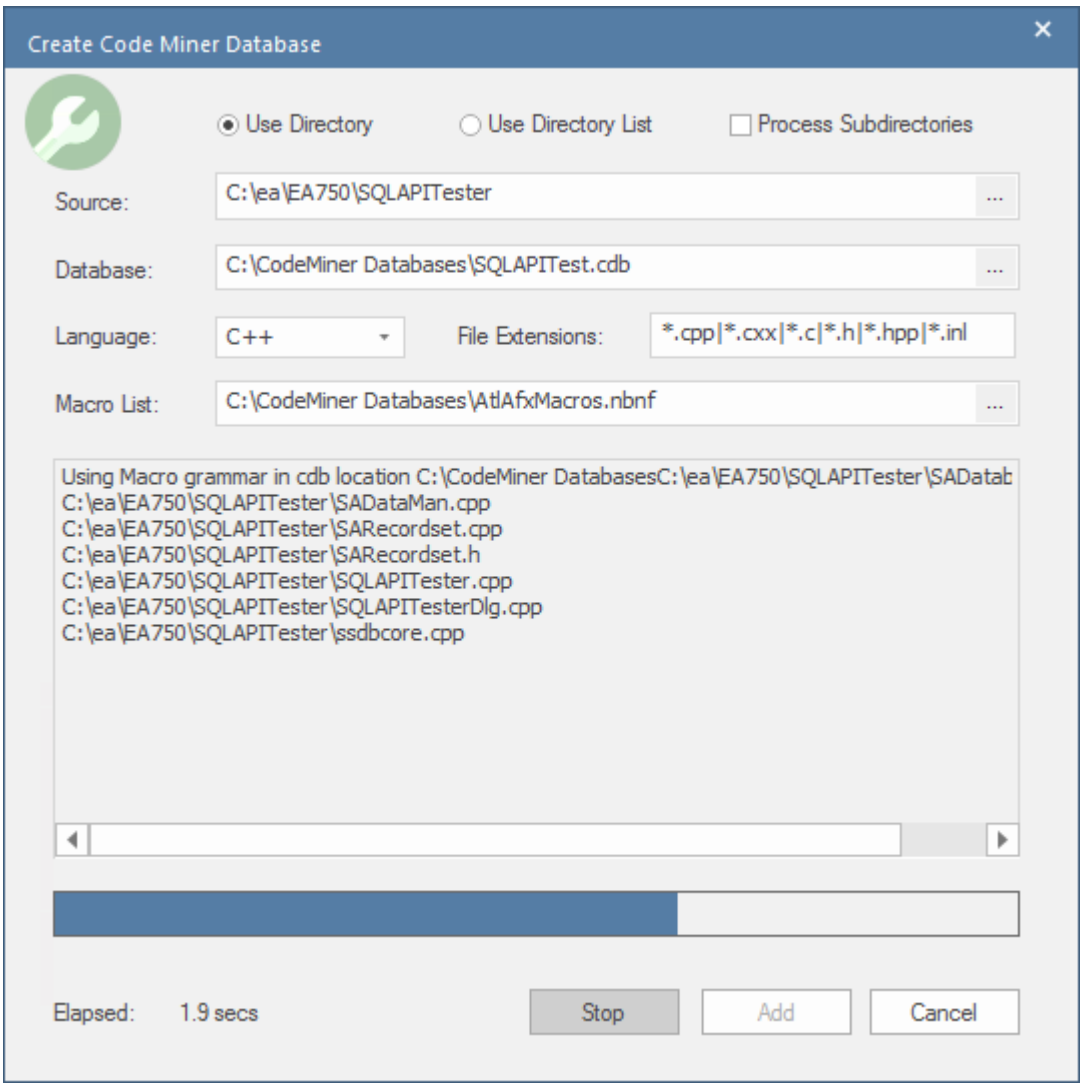
Ribbon	Execute > Tools > Analyzer, or Develop > Source Code > Execution Analyzer > Edit Analyzer Scripts

Creating a New Database

On the 'Code Miner | Libraries' page of the Analyzer's Script

Editor, click on the 'New' button to create a new database.

In the 'Create Code Miner Database' dialog, specify the folder(s) containing the project source code, select the programming language and enter the destination path for the Code Miner database. When you click on the 'Compile' button, details of the build are displayed in the log window.



When the process is complete click on the 'Add' button to add the newly created database to the library.

For detailed information on creating new databases, please see the Help topic *Creating a New Code Miner Database*.

Adding an Existing Database

Select an existing Code Miner database using the "..." selection button in the database path field (Code Miner databases have the .CDB file extension), then click on the Add button. Details about the database are listed in the library. The information presented displays the programming language grammar used to build the database. Also shown is the code base path parsed during the build and whether the parsing process was applied recursively through any sub directories.

Updating a Database

From time to time, as you update the source code for a project, you will want to update the Code Miner database built from that source code.

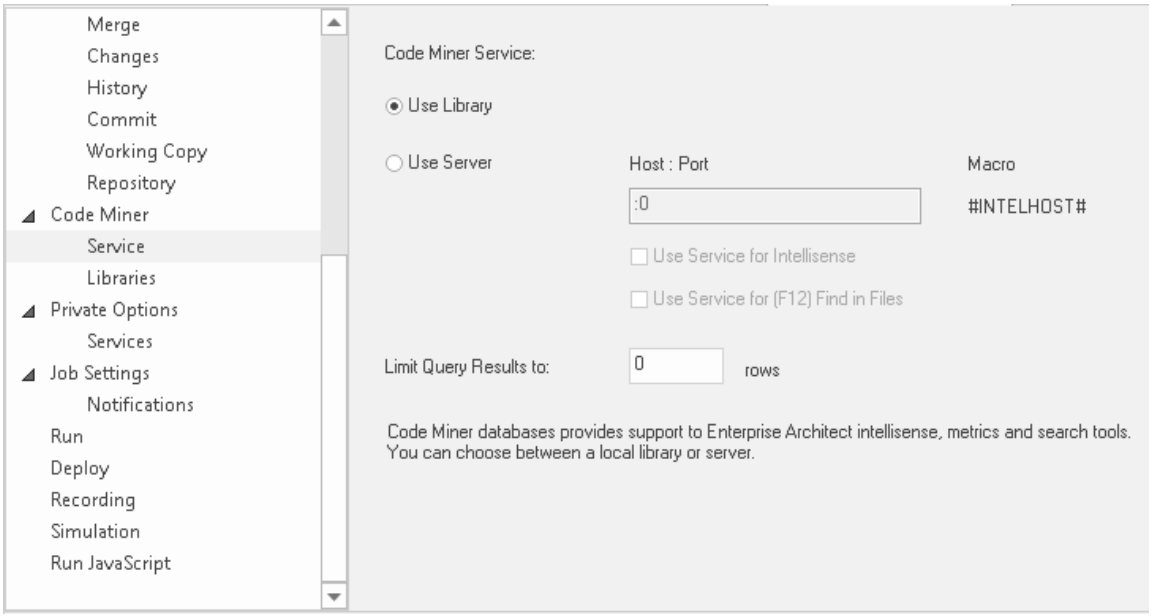
To update a single Code Miner database, select it from the list, right-click and choose 'Update selected' from its context menu. A dialog similar to the 'Create Database' dialog will display. Click on the 'Compile' button, the Code Miner will recreate the database from the updated code base.

Removing a Database

To remove a single Code Miner database, select it from the list and choose 'Remove Selected' from its context menu.

Configuring Enterprise Architect to use a Code Miner Library

In an Enterprise Architect Analyzer Script, choose the 'Sparx Intel Service' page and select 'Use Library'. Enterprise Architect then sources its Intelli-sense information from the databases listed in the 'Libraries' section of the currently active Analyzer Script.



Creating a New Code Miner Database

Enterprise Architect's Code Analyzer, the Intelli-sense features of its code editors and it's search tools all make use of Code Miner Databases.

A Code Miner Database is created by parsing source code files according to grammar rules for the selected language and storing the resulting Abstract Syntax Tree, in a read-optimized database. One or more databases can be combined to form a Code Miner Library.

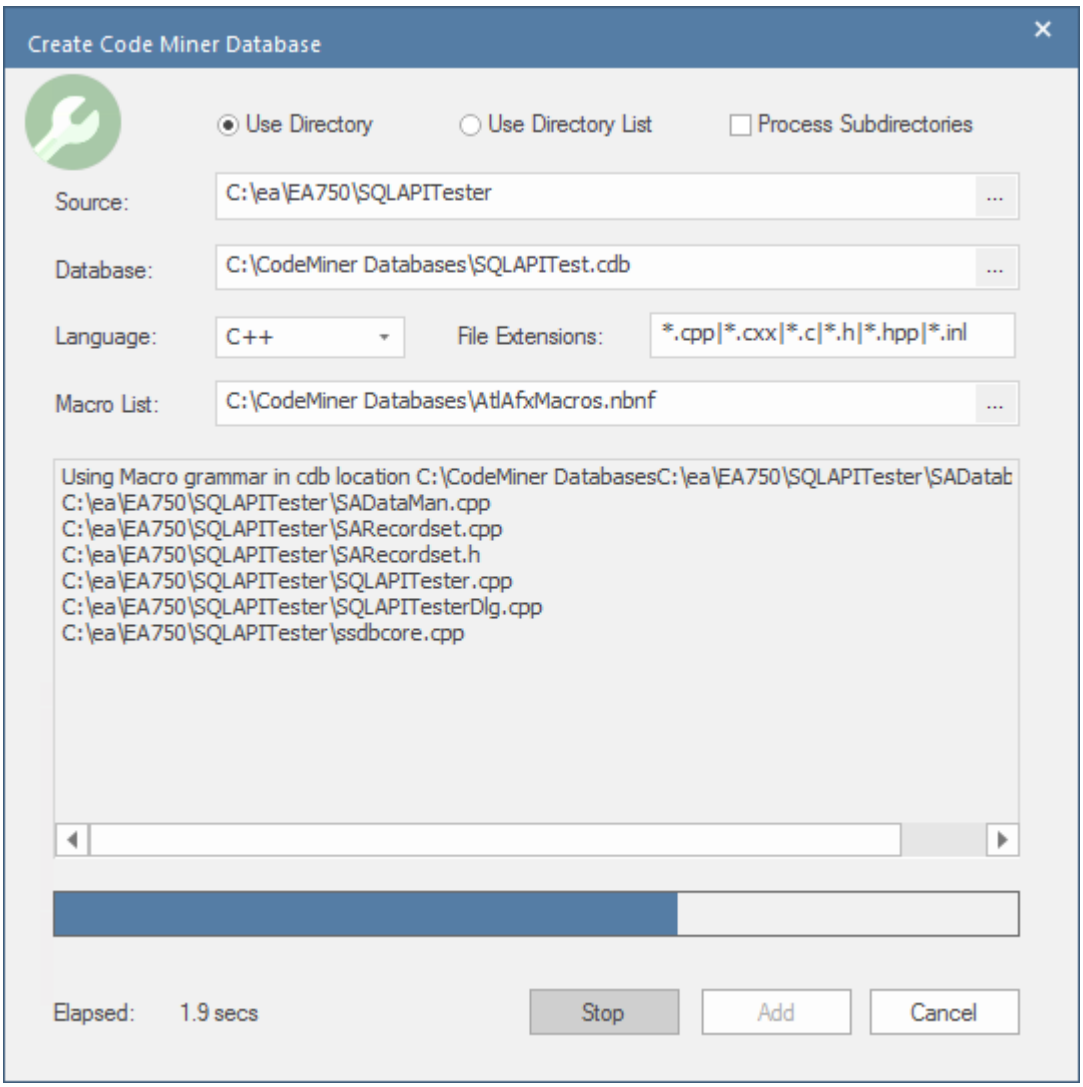
Access

Code Analyzer window	From the Code Analyzer window, click on the menu button, ≡, in the toolbar, then choose the menu option 'Create Database'.
Execution Analyzer Script Editor	With the Execution Analyzer's Script Editor window open, select the page 'Code Miner > Libraries', then click on the 'Create' button.

Create Code Miner Database Dialog

The 'Create Code Miner Database' dialog is used to initiate

the process of parsing source code files to create a Code Miner database. On the dialog, you specify a range of inputs used by the process, such as Source Code folder, Language and Macro List file, as well as the output filename. The dialog fields are described in the table presented below.



Field	Description
Use Directory	Select this option when all of the source files to be processed reside under one directory.

	When this option is selected, the checkbox 'Process Subdirectories' is enabled.
Use Directory List	Select this option when your project source code resides in multiple separate directories. In this case, you use the 'Source' field to specify a file that contains a list of directories containing the source code to be processed.
Process Subdirectories	This check-box is enabled when the 'Use Directory' option is selected. When selected, source code file residing within any subdirectories of the specified 'Source' directory will also be processed.
Source	This field is used to specify the directory (or directories) containing source code files that will be processed to create the Code Miner database. When the option 'Use Directory' is selected, this field is used to specify the root folder in which to search for source code files. When the option 'Use Directory List' is selected, this field is used to specify a user created file containing a list of path names to the directories that contain the

	source files to be processed. Clicking the ... button opens a 'File Chooser' dialog, that allows you to browse for and choose a file with the extension '.ssdirlist'. For more information, see the section *Directory List File* below.
Database	This field specifies the full path name of the Code Miner database file that will be created. The filename extension '.cdb' is used for this file.
Language	This is a drop-down list, where you specify the language used in the source code files being processed. There are a number of languages for which Enterprise Architect provides 'built-in' support. (There are built-in grammars used for parsing the supported languages). There is also an option to choose a 'Custom' language. If you choose to use a custom language, you will need to create your own grammar to support parsing of that language. When the 'Custom' option is selected, the field 'Grammar File' will be displayed, allowing you to specify the file that defines your custom grammar.
File	This field lists a number of filename

Extensions	extensions that are typically associated with source code files of the chosen language. Only files with filename extensions matching those in the list will be processed by the parser. You can add or remove filename extensions to suit your needs.
Macro List	When the language selected is 'C++', the 'Macro List' selection field is displayed. The Macro List field lets you specify a file that provides a list of macros that the parser should skip when it encounters them. For the C++ language, macros present a problem to the parser because they hide native language constructs. Adding the name of a macro to the Macro List file and updating the database will usually clear all the errors related to that macro. For more information, see the section *Extending the Macro List File* below.
Grammar File	Sparx Systems has developed grammars for all of the languages listed in the drop-down selection list. C++, C#, Java, XML and also MDGTechnology. There is also an option to select a

	'Custom' language. Users that develop a Custom language, will need to specify grammar rules for that language and save them into an nBNF file, so that the Code Miner can correctly parse source code written in that language. Enterprise Architect's Grammar Editor is designed specifically for that purpose. When you select "Custom" as the language, you should then specify the grammar file you created for that language, so that the Code Miner can correctly parse your source code. The Help Topic *Grammar Framework* provides detailed information on writing an nBNF grammar.
Output Window	The output window shows the progress of parsing the source code files. Upon completion, it also shows the names of the database file and the log file that were created along with the number of errors encountered.
Compile/Stop button	The 'Compile' button is used to start the processing operation. This button changes to a 'Stop' button once processing begins, allowing the user to abort the operation.

Add button	Once a database has been compiled, the 'Add' button can be used to add that database to a Code Miner Library. Multiple databases can be added together to build up a library that covers many source code projects. **Note**: When the 'Create Code Miner Database' dialog is opened from the Code Analyzer window, the 'Add' button is not displayed.

Directory List File

If you choose to specify a Directory List file, you will need to create a simple text file using the filename extension '.ssdirlist', that lists the full path to each directory you wish to process, with one path per line. For example:

c:\myprojects\project1\tools\scintilla

c:\myprojects\project2\src

d:\mylibs\lib1\src

If you wish to recursively process the subdirectories within a listed directory, precede that path with an exclamation mark like this:

!d:\mylibs\lib1\src

Any line that begins with a # character is treated as a comment:

include scintilla

c:\myprojects\project1\tools\scintilla

Extending the Macro List File

For the C++ language, macros present a problem for grammars because they hide native language constructs. The parser cannot not perform substitution on macros as they are often defined conditionally and the parser has no idea about the architecture. The Macro List file provides a list of macros that the parser should skip when it encounters them.

When you build a Code Miner database for a C++ source code repository, you might see errors listed. When an error occurs, use the error log to find and inspect the line of code that caused the error. This almost always identifies a macro that is causing the grammar failure. Adding that name to the macro list and updating the database will usually clear all the errors related to that macro.

For example, the error log shows this error:

C:\ea\EA750\SQLAPITester\SQLAPI\include\asa\sqlfuncs.h, line:12, col:18, Unexpected symbol ','.

Upon inspection, the line of code causing the error is this:

FUNC_INFO(extern, void, _esqlentry_, sqlstop, (SQLCA *))

(There are also many other similar lines using the macro 'FUNC_INFO'.)

So, we edit the default Macro List file, 'AtxAflMacros.nbnf', adding this line:

"FUNC_INFO" "(" skipBalanced("(", ")") ")" |

This line instructs the parser, upon encountering the macro "FUNC_INFO", to apply the function skipBalanced("(", ")"), which takes two parameters; in this case they are the opening and closing parentheses. So, the parser is instructed to ignore everything in between the opening and closing parentheses.

When the change to the Macro List file is saved and the database is recompiled (updated), all of the errors pertaining to the macro "FUNC_INFO" have been eliminated.

Learn more

- Grammar Framework
- Code Analyzer

Code Miner Queries

Code Miner queries are best considered as functions written in the Code Miner NBNF Query Language (mFQL). As such, they have unique names, can be grouped by namespace and can take one or more parameters. Queries are bundled together into one source file. This source file is identified to Enterprise Architect by naming it in your Analyzer Script.

When specified, the queries it contains are available in the Code Miner control. Parameters to these queries can be taken from selected text in a code editor, the model context or typed directly into the search field of the control.

This feature is available from Enterprise Architect Release 14.1.

```
188
189 namespace java
190 {
191 //
192 // Find all references
193 //
194 query::findByName($param1)
195 {
196     distinct(GetByValue( $param1 +))
197 }
198
199 query::findMethodByName($name)
200 {
201     move( 1, "METHOD", intersect( GetByNode("NAME"), GetByValue( $name ) )  )
202 }
203
204 query::findMethodCall($name)
205 {
206     filter( "METHOD_ACCESS", intersect(GetByNode("NAME"), GetByValue( $name )) )
207 }
208
```

This image illustrates an mFQL query from the Sparx Queries file distributed with Enterprise Architect installations. The syntax for composing an mFQL query and the mFQL language itself is described here.

Query Syntax

The syntax for composing mFQL queries is:

namespace
{
    query:*name*([$*param1 [*, $*param2]]*)
    {
        mfql-expression
    }
}

where:

- *namespace* names the collection of queries
- *name* is the 'function' name of the query
- *$param1 and $param2* are placeholders for argument substitutions at runtime
- *mfql-expression* is an mFQL expression

Code Miner Query Language (mFQL)

The Code Miner system provides fast and comprehensive access to the information in existing source code. By parsing all source code and storing the resulting Abstract Syntax Tree (AST) in a read-optimized database, the system provides complete access to all aspects of the original source code, in a machine understandable format.

The core goal behind the system is to provide access to the data hidden within source code in a timely and effective manner. Great pains have been taken to ensure maximal performance, while providing the simplest interfaces possible. As a result the system can be used to analyze program structure, calculate metrics, trace relationships and even perform refactoring.

mFQL

mFQL is the query language of the Code Miner. The language itself is reasonably simple, providing a small number of commands. Simple as the language is, it supports queries of arbitrary size and complexity. The design provides extreme performance for all queries, great and small.

The language is set-based; it operates primarily on sets of abstract data obtained through discrete vertical indices. For our purposes, a set is an ordered array of numbers, each of which is a pointer to a node in the AST Store. A discrete vertical index provides a mechanism to retrieve sets by

discrete value.

The language includes the three basic set-joining operations. These are 'intersect', 'union', and 'except'. The 'except' join is, more precisely, a 'symmetric difference' join. A 'complement' join can be achieved by using a short sub-query; this is detailed in the 'except' join documentation. The 'offsetIntersect' join is also discussed in detail there.

The Code Miner database provides three discrete vertical indices in its AST Store. These indices are 'node name', 'attribute name', and 'attribute value'. Each vertical index can be queried for a discrete value, which will return a set of all nodes where that value is present. The three vertical indices are queried using the functions 'getByNode', 'getByName' and 'getByValue', respectively.

Set 'traversal routines' provide mechanisms to filter sets based on patterns in the AST. The traversal routines are either destructive (move) or non-destructive (filter). Destructive traversals modify the set member values to point to the target node; non-destructive traversals ensure the target node exists. In both cases, nodes that cannot complete the traversal are removed.

Please note that all traversals in mFQL are upwards. Downwards traversals are technically complex, as a node could have any number of child nodes. Conversely, upward traversals are much simpler, with every node having zero or one parent node. For these reasons, downward traversals are not supported in the query language.

Although there are only a small number of operations in mFQL, the language is capable of expressing very finely

grained and complex queries. The language is functional in design, and supports arbitrary nesting calls.

mFQL queries execute at lightning speed. The backend database was designed from the ground up for read performance. The query parser was hand optimized. Knowing that it always has pure ordered sets, the low-level code takes several shortcuts to perform joins with minimal work effort.

In order to use nBNF effectively one must possess a working knowledge of the target language, and an intimate knowledge of the grammar used to parse it.

The mFQL Language

This section provides a list of Code Miner NBNF Query Language (mFQL) queries with explanations and comments.

The queries shown here demonstrate different capabilities and different approaches to exploring and extracting data using mFQL and the Code Analyzer in Enterprise Architect. The mFQL queries help make the syntax human-readable and intuitive, and have been extended in Enterprise Architect to include additional functions necessary to do real things with Code Miner databases.

The Query Language

String parameters are indicated by **string**, set parameters are indicated by **set** and number parameters are indicated by **numbers**.

Notes

1. Case sensitivity is defined by the case sensitivity of the language of the source code used to populate the database. If the source language is case sensitive (such as C++) all string literal parameters are case sensitive. If the source language is case insensitive (such as SQL) all string literal parameters are case insensitive.
2. Hierarchical traversals in mFQL are generally upwards. Downwards traversals are not optimal, as a node might have any number of child nodes. Upward traversals are much simpler, with every node having zero or one parent node. Downward-looking queries such as 'children' only

query one level down.

3. Synonyms of some keywords are provided to better express a query intent or action in particular circumstances, and to support legacy queries. Synonyms are simple alternatives for the base function keyword. For example, 'type(str)' can be written as 'node(str)' or 'byNode(str)' or 'getByNode(str)'. The current specified version is the preferred one, with the synonyms only intended for use in exceptional circumstances.

Statement	Description
type(value)	type(value) Extracts a set based upon node name. The exact name for a node is defined by the grammar used to parse the original source. In this example, find all nodes within the database of type "CLASS". type("CLASS") *Synonyms:* • node • byNode • getByNode
with(name)	with(name) Searches the database for any element that has a named attribute matching the search string. The value of the attribute is

	ignored - this is a query for the attribute NAME only. All nodes with one or more attributes of the specified name are returned. If a single node has two attributes of the same name, one instance of that node is returned. This example will find all elements in the database that have an attribute named "Type": with("Type") *Synonyms:* • name • byName • getByName
find(value) find([+] value [+ value] [+])	find(value) find([+] value [+ value] [+]) Search the database for any element having an attribute value with the provided search term. The match is case sensitive and must match the whole word. You can extract a set based upon an attribute value; when extracting nodes by attribute value, the values of all attributes for the node are considered. Wildcards allow for specifying a subset of attribute values for a node. Wildcards

	can be used at either the beginning or end of a value specification: • A leading concatenation symbol allows for any number of attributes preceding the first matched attribute • A trailing concatenation symbol allows for arbitrary trailing attributes In both cases, if the node would match without wildcards, it will match with them – the wildcard specifies any number of leading/trailing attributes, including none. In this example, we retrieve a set of nodes that have their last two attributes being “.” and “sun”. The leading concatenation symbol specifies that any number of attributes (including none), with any value, can exist before the matched attributes, but none can follow. find(+ “.” + “sun”) The next example has a trailing wildcard. Any node with “com”, “.” and “sun” as the first three attributes will be returned. Any number of trailing attributes can exist. find(“com” + “.” + “sun” +) Both wildcards can be used together. In this example nodes with attributes with

	the three specified values as names, in order, regardless of leading or trailing attributes, will be returned. find(+ “com” + “.” + “sun” +) Example: Find all nodes in the database that have any attribute with a value of "CString": find("CString") Example: Find all nodes in the database with a set of attributes having these values in this order: find("com" + "." + "sun") *Synonyms:* • value • byValue • getByValue
has(name,value)	has(name, value) Finds all elements that have a named attribute with the value supplied. Unlike the intersection of 'find' and 'with', this query will only return rows with an exact name/value pair. has("Type","CString")
having(name,	having (name, value, set)

value, set)	Finds all elements within the supplied set that have a named attribute with the given value. Similar to 'has' but supplies a predefined input set to search. Whether to use 'has' or 'having' is generally determined by the kind of query structure being used, its depth and its readability. Example 1: Find all Property elements with a name of "m_strName" that have a Type attribute of CString: having("Type","CString",this("PROPERTY","NAME","m_strName")) Example 2: Extend Example 1 to only include those that store a CString *: having("Reference","*", having("Type","CString",this("PROPERTY","NAME","m_strName")))
this(type,name,value)	this(type, name, value) Function finds one or more elements that have a matching TYPE, and WITH a named attribute having the specified VALUE. Example: Find all operations named "Import Solution":

	this("OPERATION","NAME","ImportSolution") *Synonyms:* • object • item
like(name,like,set)	like(name, like, set) like(name, like, set, caseSensitivity) Finds a set of elements that have an attribute that starts with the search sub-string. Note that this is not a fully wild-carded search but is case sensitive and must be an exact match for the length of the search string. caseSensitivity: specify one of: • "CaseSensitive" • "CaseInsensitive" • "Default" - uses the code language to determine which to use Note that case-insensitive searching can be slower. Example: Find all Classes in the database whose NAME attribute starts with "CMapStr": like("NAME","CMapStr",gettype("CLAS

	S"))
contains(name,contains,set)	contains(name, search_term, set) contains(name, search_term, set, caseSensitivity) Finds a set of elements that have an attribute that contains the search sub-string. Similar to 'like' except this will search the entire string, not just from the start. caseSensitivity: specify one of: • "CaseSensitive" • "CaseInsensitive" • "Default" - uses the code language to determine which to use Note that case-insensitive searching can be slower. Note that this search can be slower than using 'like'. Example: Find all Classes in the database whose NAME attribute contains "MapStr": contains("NAME","MapStr",gettype("CLASS"))
and(set1,set2,	and(set1, set2, ...)

...)	Returns the intersection of nodes between two or more sets. To be included in the final set, an element must exist in ALL the input sets. *Synonyms:* • intersect(set, set,...) • {set, set, ...}
union(set1,set2,...)	union(set1,set2, ...) Returns the distinct union of ALL nodes present in the input sets. *Synonyms:* • or(set, set ...) • [set, set ...]
ancestor(str,set)	ancestor(str, set) ancestor(num, set) ancestor(num, str, set) The ancestor function traverses each node in a set of a number of parent nodes, excluding any nodes that fail the traversal. The number of nodes to traverse, the name of the target node for the traversal, or both can be provided as parameters.

	When the number of nodes is provided, but the target node name is not, any nodes with the specified number of parents will pass the traversal. Any node that runs out of parents will be dropped from the set. When the name of the target is specified, but the number of nodes to traverse is not, any nodes with a parent with a matching name, at any point in the hierarchy, will pass the traversal. Any node with no matching parent is excluded. When both the number of nodes and the target name are provided, only nodes that have a parent node with the specified name, at the specified offset, pass the traversal. All other nodes are removed from the set. In this example the set hasParameter("CString","&",1) is moved up to an ancestor node named "OPERATION". If the move fails the node is dropped from the result. ancestor("OPERATION",hasParameter("CString","&",1)) In this example the set is moved up one

	rung to its parent. If there is no parent, the node is dropped from the result. ancestor(1,hasParameter("CString","&",1)) In this example the set is moved up three steps to its parent->parent->parent . If there is no such node, the node is dropped from the result. ancestor(3,hasParameter("CString","&",1)) *Synonyms*: • move
filter(str,set)	filter(str, set) filter(num, set) filter(num, str, set) The filter function is the same as the 'ancestor' function, except that it returns nodes from the original child set rather than new ancestor nodes. If a node is unable to pass the specified traversal, it is removed from the set. Nodes that pass the traversal are left in place, unmodified. In this example the set hasParameter("CString","&",1) is tested for an ancestor node named

	"OPERATION". If the move fails the node is dropped from the result. The result set is a set of parameter types that meet the criteria. filter("OPERATION",hasParameter("CString","&",1))
match(NameA,setA,NameB,setB)	match(NameA,setA, NameB,setB) 'Match' takes two input sets and two attribute names and returns all those in 'setA' that have a matching record in 'setB', as determined by comparing the values of the named attributes 'strA' and 'strB'. That is, a 'setA' row is included if the value of attribute 'strA' in 'setA' exists in 'setB' as the value of an attribute of name 'strB'. 'Match' is useful for finding where one element feature is used in a different context elsewhere in the database. For example, where a unique element name or GUID is referenced by another element. In this example, we match the attribute named 'TYPE' from the right set to the attribute 'NAME' in the left set. The result will be all "CLASS" type objects from the left set with NAME == TYPE(s) as

	specified in the right set. match("NAME",type("CLASS"),"TYPE",this("PROPERTY","NAME","m_pLink"))
graph(targetType, targetName, linkType, linkName, start)	graph(targetType, targetName, linkType, linkName, start) Find a recursive set of elements that form some kind of graph when linked by attribute pairs, in a manner similar to 'match'. The starter set is queried for all owned instances of the linkType with link Name and these are matched against a new query based on the targetType with targetName. The new set is filtered in a manner similar to 'match', and all elements in the new query that share the same NAME/VALUE pair as from the starter set are kept; all others are discarded. The resultant set is then fed back into the original set as the starter for the next iteration, with the results at each stage being added together to form the final result set. Example: Return the Class hierarchy for a Class named "Car". graph("CLASS","NAME","GENERALIZATION","GENERAL",this("CLASS","N

	AME","Car"))
prune(set_test,str,set_base)	prune(set_test, str, set_base) prune(set_test, num, set_base) For two sets of nodes, temporarily move one set UP to the named or numeric position in its ancestry and filter out any nodes that do not exist by strict intersection in the TEST set. The first set is the TEST set, the right or last set is the BASE set. The set returned is all the elements in the BASE set that, when moved to the TEST position, matched something in the TEST set. The returned nodes are the original nodes from the BASE set and are not moved up when returned. Example 1 finds the set of parameter types used for operation parameters named "CustomerName" across the whole database. prune(this("PARAMETER","NAME","CustomerName"),"PARAMETER",type("PARAMETERTYPE")) Example 2 finds all Properties of a Class named Customer, assuming the grammar

	used to compile the database placed the Property definition two hierarchy levels below the Class definition. prune(this("CLASS","NAME","Customer"),2,type("PROPERTY"))
andat(str,test, base)	andat(str, base, test) andat(num, base, test) andat(num, str, base, test) For two sets of nodes, temporarily move one set UP to the named or numeric position in its ancestry and filter out any nodes that do not exist by strict intersection in the TEST set. The first set is the TEST set, the right or last set is the BASE set. The set returned is all the elements in the BASE set that, when moved to the TEST position, matched something in the TEST set. The returned nodes are the original nodes from the BASE set and are not moved up when returned. Similar to 'prune', this query supports additional options and structures the inputs in a different order to facilitate different kinds of stacked searches. The 'andat' function performs both a non-destructive tree traversal and an

	intersect join in one operation. Each node in the left set is traversed according to parameters provided, then the result of the traversal is intersected with the right set. If the intersect passes, the original node is added to the result set. If the intersect fails, the node is excluded from the result set. The traversal parameters for 'andat' are the same as for 'ancestor' and 'filter'. For more information about the traversal parameters, see the 'ancestor' function. Example: For the set of all "PROPERTY" nodes in the database, move them up to a parent node of type CLASS and then intersect the result with the right hand set - in this case a CLASS named CDiagram. All nodes that pass this test are returned as PROPERTY nodes, effectively giving the set of all properties of the Class CDiagram. andat("CLASS",type("PROPERTY"),this("CLASS","NAME","CDiagram")) *Synonyms:* • offsetIntersect • offsetx
unique(left,ri	unique(left, right)

ght) / except(left,right)	except(left, right) Except joins return sets that contain any nodes from either set that do not appear in both sets. This join is similar to a bitwise XOR operation. In set theory, this type of join is referred to as a 'symmetric difference join'. {1, 2, 3} excepted with {2, 3, 4} results in {1, 4}
omit(left,right) / exclude(left,right)	omit(left, right) exclude(left, right) Exclude joins return a set that contains all nodes from the left set that do not appear in the right set. In set theory, this type of join is referred to as a 'relative complement join'. {1, 2, 3} complemented with {2, 3, 4} results in {1}
differ(name,set,name,set)	differ(name, set, name, set) Return a set of nodes that do not have a matching row in another set, using a NAME/VALUE pair from each set to match on. Example: This more complex example tests the complete set of Generalizations for a Class hierarchy and identifies

	missing or unresolved Class names in the total inheritance hierarchy. Like the 'match()' function discussed later, this function iterates over attribute name/value pairs as specified in the left and right input sets, but only includes rows in the final set where there is NO match. differ("GENERAL", children("GENERALIZATION", graph("CLASS","NAME","GENERALIZATION","GENERAL", this("CLASS","NAME","CMainFrame"))), "NAME", graph("CLASS","NAME","GENERALIZATION","GENERAL", this("CLASS","NAME","CMainFrame")))
children(type,set)	children(type, set) Return a set of child nodes of a specified type for one or more parents in the source

	set. For all children regardless of type, use an empty string. For example, in the first query we return ALL first level children of the CMainFrame Class. In the second query we restrict the nodes returned to be of type "REGION" only. children("",this("CLASS","NAME","CMainFrame")) children("REGION",this("CLASS","NAME","CMainFrame"))
childcount(num,type,set)	childcount (num,type,set) Return nodes that exactly match the number of specified children of a specified type. For example, only return operations that have 5 parameters. An example usage is in specifying an exact operation signature, so we check firstly that parameter1 and parameter2 match the type we are querying for, then move those to their operation ancestor and intersect the result with the operation name "GetFromCache" we are interested in. To rule out spurious hits with operations having more than 2 parameters, we explicitly add childcount(2, ...) to ensure we only get

	operations that have 2 parameters.
	
	childCount(2,"PARAMETER",
	{
	ancestor("OPERATION",hasParameter("CString","&",1)),
	ancestor("OPERATION",hasParameter("CString","&",2)),
	this("OPERATION","NAME","GetFromCache")
	}
	)

byAddress(num)	byAddress(num) The byAddress function is used in applying the results of one query to another. For example, we might have a node of particular interest, and want our query to return only nodes that join (in some way) to the specified node. byAddress(node: number) This example builds a set containing the single node related to the address specified: byAddress(11256)

byPosition(File, Offset)	byPosition(File, Offset) The byPosition function is used to return the inner-most node that covers a certain position in a file. This function is useful for locating a position in the AST based upon a file position.
distinct(set)	distinct(set) The distinct function ensures that a set has no duplicate values. All duplicate values are excluded from the result set.

Set Extraction

These procedures extract sets from discrete vertical indices. There are three indices available, each with a specific extraction function. String literal parameters to these functions could be case sensitive. Case sensitivity is defined by the language of the source code used to populate the database. If the source language is case sensitive (as C++ is) all string literal parameters are case sensitive. If the source language is case insensitive (as SQL is) all string literal parameters are case insensitive.

type

type(value: string)

Extract a set based upon a node name. The exact name for a node is defined by the grammar used to parse the original source. In this example, all nodes with the name "OPERATION" are returned.

type("OPERATION")

_

with

with(value: string)

Extract a set based upon attribute name. All nodes with one or more attributes of the specified name are returned. If a single node has two attributes of the same name, one instance of that node is returned. This example returns all

nodes with one or more attributes named "NAMEPART".

with("NAMEPART")

find

find([+] value: string [+ value: string] [+])

Extract a set based upon an attribute value. When extracting nodes by attribute value, the value of all attributes for the node are considered. Wildcards allow for specifying a subset of attribute values for a node.

When a single value is provided, all nodes that have a single attribute with the value specified are returned. If a node has any other attributes, it is excluded. In this example, all nodes with exactly one attribute with the value of 'i' are returned.

find("i")

More than one value can be specified by using a concatenation symbol. When more than one value is specified, the resulting set will contain all nodes that have attributes with exactly the values specified, in the order specified. Any node with extra leading or trailing attributes is excluded. This example retrieves a set of all nodes with a set of three attributes with the values “com”, “.” and “sun”, in that order.

find("com" + "." + "sun")

Wildcards can be used at either the beginning or end of a value specification. A leading concatenation symbol allows for any number of attributes preceding the first matched attribute. A trailing concatenation symbol allows for

arbitrary trailing attributes. In both cases, if the node would match without wildcards, it will match with them – the wildcard specifies any number of leading/trailing attributes, including none.

In this example, we retrieve a set of nodes that have their last two attributes being “.” and “sun”. The leading concatenation symbol specifies that any number of attributes (including none), with any value, can exist before the matched attributes, but none can follow.

find(+ “.” + “sun”)

The next example has a trailing wildcard. Any node with attributes “com”, “.” and “sun” as the first three attributes will be returned. Any number of trailing attributes can exist.

find(“com” + “.” + “sun” +)

Both wildcards can be used together. In this example, nodes with attributes named as the three values specified, in order, regardless of leading or trailing attributes, will be returned.

find(+ “com” + “.” + “sun” +)

—

Set Traversal

ancestor

ancestor(count: number, source: set)

ancestor(value: string, source: set)

ancestor(count: number, value: string, source: set)

The 'ancestor' function traverses each node in a set up a number of parent nodes, excluding any nodes that fail the traversal. The number of nodes to traverse, the name of the target node for the traversal, or both can be provided as parameters.

- When the number of nodes is provided, but the target node name is not, any nodes with the specified number of parents will pass the traversal; any node that runs out of parents will be dropped from the set
- When the name of the target is specified, but the number of nodes to traverse is not, nodes with a parent with a matching name at any point in the hierarchy will pass the traversal; any node with no matching parent is excluded
- When both the number of nodes and the target name are provided, only nodes that have a parent node with the specified name at the specified offset pass the traversal; all other nodes are removed from the set

It is possible - even likely - that these calls will generate sets having duplicate values. This is by design, as the concrete

rules for sets do not define them as being discrete. If (as in most cases) you want your set to be discrete, use the 'distinct' function described in the *The mFQL Language* Help topic.

This sample extracts a set of all nodes named 'OPERATION', then traverses each node up one level to its immediate parent. Any 'OPERATION' node with no parent is excluded.

ancestor(1, getByNode("OPERATION"))

This sample extracts a set of all nodes named 'OPERATION', then traverses each node up to the first 'CLASS' parent node. Any 'OPERATION' node with no 'CLASS' parent is excluded.

ancestor("CLASS", getByNode("OPERATION"))

This sample extracts a set of all nodes named 'OPERATION', then traverses each node up one level to its immediate parent. If the parent node is not a 'CLASS' node, or the node fails to traverse though a lack of parent nodes, it is excluded.

ancestor(1, "CLASS", getByNode("OPERATION"))

_

filter

filter(count: number, source: set)

filter(value: string, source: set)

filter(count: number, value: string, source: set)

The 'filter' function is the same as the 'ancestor' function, except that it does not modify nodes – it is non-destructive. If a node is unable to pass the specified traversal, it is removed from the set. Nodes that pass the traversal are left in place, unmodified.

It is often desirable to filter a set by the current node name. This can be used to ensure that the nodes returned from a 'with' or 'find' call are of a particular node type. This example returns all nodes with an attribute with the value of “CFoo”, where the resulting node is a “TYPE” node.

filter(0, “TYPE”, find(“CFoo”))

For more details on the use of the 'filter' function, see the 'ancestor' function.

—

Set Joining

and

and(left: set, right: set)

An 'and' join will return a set containing all nodes that exist in both the left and right set. This join is comparable to a bitwise AND operation. In set theory, this type of join is called an 'intersection'.

{1, 2, 3} intersected with {2, 3, 4} results in {2, 3}

This example returns a set that contains all nodes that have a single attribute with the name of "TYPE" and the value of "int".

```
and(
find("int"),
with("TYPE")
)
```

union

union(left: set, right: set [, right: set])

'Union' joins return a set that includes all nodes found in either the left or the right set. This join is used to combine the results of two or more sub-queries into a single set. A 'union' join is similar to a logical OR operation. In set theory, the 'union' join is known as a union.

The 'union' join is able to operate on more than two sets. The result is a set that contains all nodes from all supplied sets. The 'union' join is the only join able to operate on more than two sets.

The result of a 'union' join is always a discrete set, unless one of the source sets contained duplicates. This means that duplicates in source sets will be preserved, but the 'union' join itself will not generate duplicates.

{1, 2, 3} unioned with {2, 3, 4} results in {1, 2, 3, 4}

This sample creates a set containing all nodes with an attribute named “TYPE” or a single attribute with the value of “int”.

```
union(
find("int"),
with("TYPE")
)
```

except

except(left: set, right: set)

'except' joins return sets that contain any nodes from either set that do not appear in both sets. This join is similar to a bitwise XOR operation. In set theory, this type of join is referred to as a 'symmetric difference' join.

{1, 2, 3} excepted with {2, 3, 4} results in {1, 4}

For more information on the 'symmetric difference' join in set theory, see

https://en.wikipedia.org/wiki/Symmetric_difference

This sample returns a set of all nodes with an attribute named "TYPE" but no single attribute with the value of "int", plus all nodes with an attribute with the value of "int" that are not named "TYPE".

```
except(
 find("int"),
 with("TYPE")
)
```

exclude

exclude(left: set, right: set)

'exclude' joins return a set that contains all nodes from the left set that do not appear in the right set. In set theory, this type of join is referred to as a relative complement join.

{1, 2, 3} complemented with {2, 3, 4} results in {1}

This sample returns a set of all nodes with a value of “int” that are not “TYPE” nodes:

```
Exclude(
find(“int”),
with(“TYPE”)
)
```

andat

andat(count: number, left: set, right: set)

andat(value: string, left: set, right: set)

andat(count: number, value: string, left: set, right: set)

The andat function performs both a non-destructive tree traversal and an intersect join in one operation. Each node in the left set is traversed according to parameters provided, then the result of the traversal is intersected with the right set. If the intersect passes, the original node is added to the result set. If the intersect fails, the node is excluded from the result set.

The traversal parameters for andat are the same as for 'ancestor' and 'filter'. For more information about the traversal parameters, see the 'ancestor' function described in the *Set Traversal* Help topic.

This sample takes all “NAME” nodes, traverses them up one parent, and intersects them with a set of all “CLASS” nodes. If a “NAME” node passes both the traversal and intersect join, it is added to the result set. The result is a set of all “NAME” nodes whose immediate parent is a “CLASS” node.

andat(1,

type(“NAME”),

type(“CLASS”)

)

—

Sparx Intel Service

The Sparx Intel service program provides a means for development projects and players to gain valuable insight into the code bases and software frameworks they are working with. The service acts as a provider to Enterprise Architect clients, allowing access to Intelli-sense in code editing and insightful search results in search tools.

The Sparx Intel service is part of the Sparx Satellite Services umbrella. The service can run on a local network or Cloud running Microsoft Windows. The Sparx Intel Satellite service can be installed as a Windows service or run as a standalone process. The service allows multiple Enterprise Architect clients to access and query the same information from many different software domains and frameworks.

This feature is available from Enterprise Architect Release 16.0

Sparx Intel Service Configuration

The program SparxIntelService.exe runs one or more intel services for Enterprise Architect. The program is located in the same install folder as Enterprise Architect, and it uses a configuration file that names the services that can run on the local machine.

In the examples in this topic, the program will attempt to use the file c:\mystuff\myservices.config. It will look for a service named *EA* and, if found, start it.

```
SparxIntelService.exe listen service=EA
config=c:\mystuff\myservices.config
```

The Config File Format

The configuration file has this format:

```
# comment
# comment
# comment
{       # start of service definition
  ...   # list of directives as pairs
}       # end of service definition
{       # start of service definition
  ...   # list of directives as pairs
}       # end of service definition
```

Comments are indicated by the # character.

If the config directive is omitted (not recommended), the program will look for a config file of the same name as the program, in the same directory as the program.

In this example the program will attempt to use the file SparxIntelService.config in the same folder:

SparxIntelService.exe listen service:EA

Directive	Description
name	When a service is named on the command line, the service with the matching name attribute will be started.
status	When status = ON, the service will be started; otherwise, it will not be started.
lazyload	When lazyload is 'true', any Code Miner database will be delay loaded until an Intel request is made to the service.
loglevel	Defines the level of information logged, as a combination of keywords { information, warning, error} separated by a '|'. For example: loglevel= Information|warning|error
logoutput	Specifies the full pathname of the log file to write to. For example: logoutput=c:\logfiles\intel-service-project

	1.log
database	Specifies the full path name of the Code Miner database to be loaded. For example: database=c:\intel--service\project1.cdb Multiple 'database' directives are allowed, each specifying a different database.
allow	Identifies the IP address that is permitted to connect to the service on the Port. For example: allow=localhost allow=127.0.0.1 allow=172.160.* (wildcards are allowed when the 'network' directive has a value of 'network' or 'public', but not 'local')
network	Allows service connections to be restricted. • local - the service will not listen on any connection other than localhost • network - when used with wildcard 'allow' directives, allows clients on an allowed IP address wild card to connect • pubic - allows any connection

show	When 'true', the Console window for the service will be shown; the default is 'false'.
port	The Port on which the service will listen.

The Service Configuration Template

When choosing the 'Execute > Tools > Services > Code Miner Service > Edit Configuration File' ribbon option you display the Windows 'Save As' browser through which you can choose either the config file to open or where a file should be created.

If no config file is recorded in the registry and you specify a non-existent filename, that file is created, filled with a 'bare bones' configuration skeleton and saved. The selected/new configuration is then shown in the Enterprise Architect default editor.

The 'bare bones' template is shown here.

```
#-----------------------------------------------------------------
---------
# Sparx Intel Service Configuration File
#
-----------------------------------------------------------------
--------
```

```
# This file is used to describe one or more intel services and the code miner databases that they support
# This file can be used in EA to manage a number of services on the local machine
#
-------------------------------------------------------------------------
# Service Attributes
#
-------------------------------------------------------------------------
# name              The unique name of the service in this file
# status            "ON" - service can run, "OFF" service will never run
# lazyload          "true" - databases are loaded n demand, "false" - databases are loaded when service starts
# port              Unique Port number that service will listen on and EA will connect to
# network           [optional,default=local] Restricts service to listening to locahost only (local), to a range of addresses (network) or any address (public)
# allow             Allows a specific IP address or wildcard IP address to connect (if network is NOT local)
#                   (There can be multiple allow directives present)
# autoupdate        "true" - will detect updates to listed databases and reload them, "false" default, changes are not
```

```
detected
# show                [optional,default=false] shows the console window for the service
# logoutput           [optional] The path of a log file which service can write to
# loglevel            [optional] The levels of information logged. Combine with '|' character, e.g.: { information|warning|error }
# database            [Required] The full path to a codeminer database which usually has the .cdb file extension
#                     (There can be multiple database directives present)
#
#
-------------------------------------------------------------------------
# Attribute Values
#
-------------------------------------------------------------------------
#
# <string>   - text. (do not include quotes)
# <boolean>  - text, { true, false, ON, OFF }
# <path>     - fully specified file path to codeminer database
# <number>   - digits
#
```

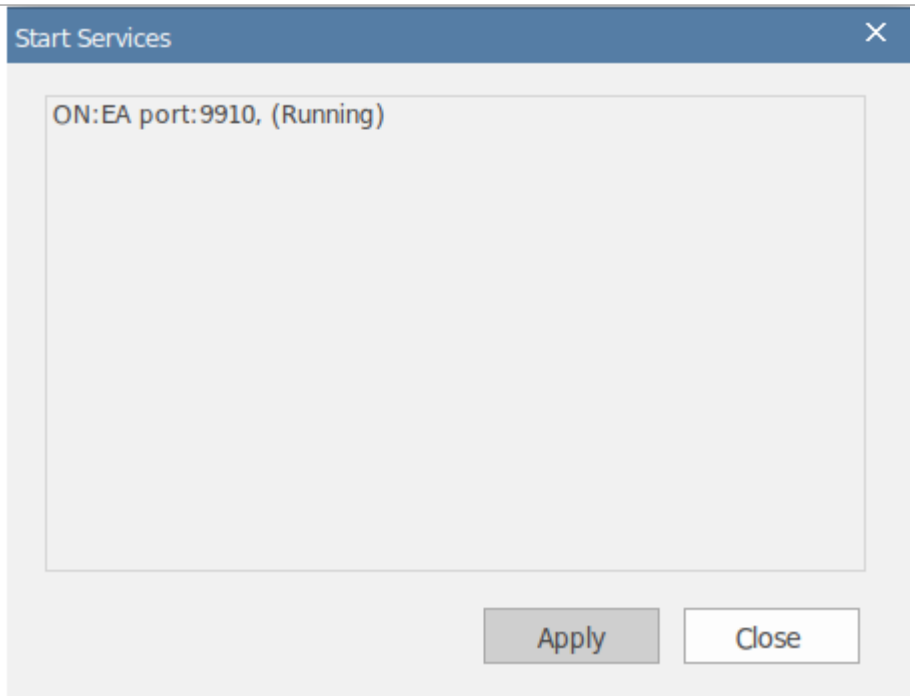
```
--------------------------------------------------------------------------------
#
{
    name=<string>,
    status=<boolean>,
    lazyload=<boolean>,
    port=<number>,
    allow=<string>,
    allow=<string>,
    network=<string>,
    autoupdate=<string>,
    show=<boolean>,
    logoutput=<string>,
    loglevel=<string>,
    database=<path>,
    database=<path>,
    database=<path>
},
{
    name=Project1,
    status=ON,
    lazyload=TRUE,
    allow=localhost,
    allow=127.0.0.1,
```

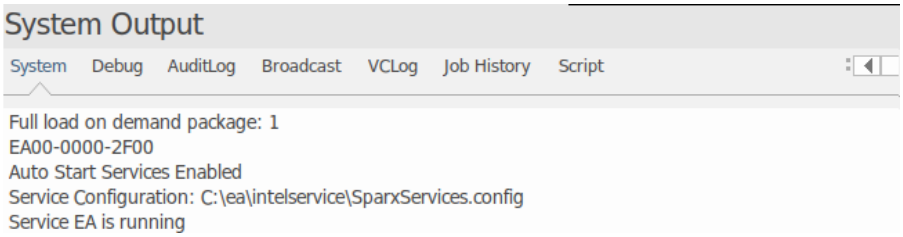
```
    port=9999,
    autoupdate=true,
    database=c:\Project1\Project1.cdb
}
```

The Sparx Intel Service Ribbon Options

When a Service Configuration file exists, you can edit it or execute it using a number of options available from the 'Execute > Tools > Services' ribbon option within the Code Miner menu option group.

Option	Description
View Status of All Services	(Above all categories of Service.) This option displays a view that lists the status of each Enterprise Architect service named in the current Configuration file, and its state.
Start	This option reads the current Service Configuration file and starts services that are configured to run, and stops running services that are not configured to run. A service is configured if: 1. It is named in the config file. 2. It has the attribute status:ON.

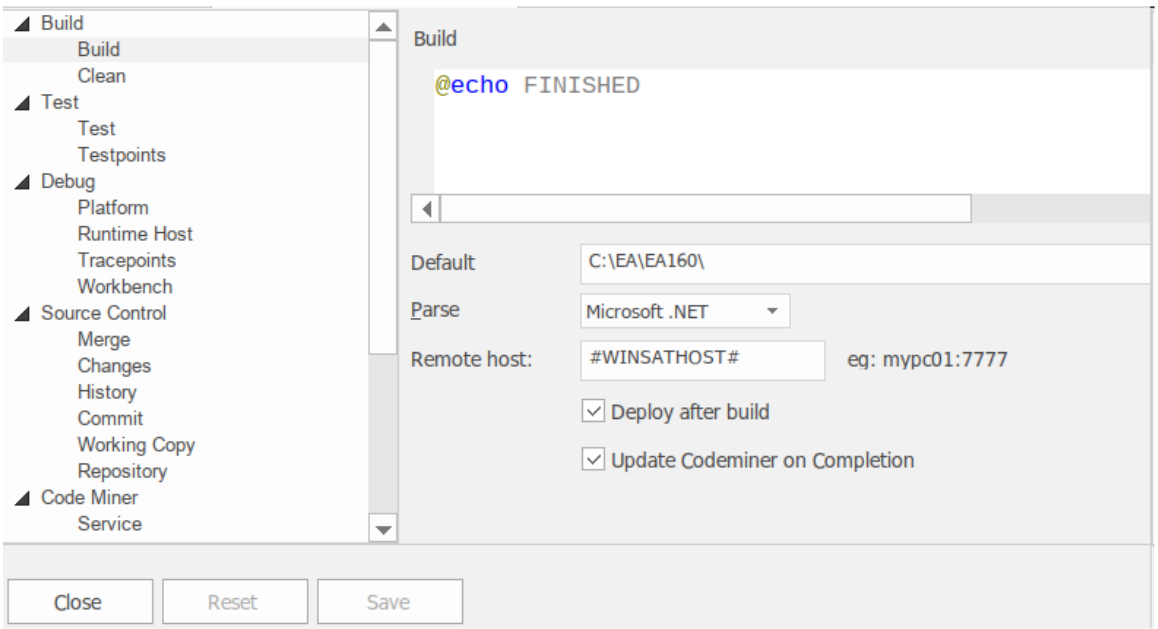
<table>
<tr><td></td><td> 
</td></tr>
<tr><td>Stop All</td><td>This option stops any services that are currently running.</td></tr>
<tr><td>Edit Configuration File</td><td>This option prompts for the Service Configuration file to use, then opens that file in an Enterprise Architect text editor. The system remembers where the file is held.
<pre>
<number>   - digits
--
#
{
    name=project1,
    status=ON,
    lazyload=true,
    port=9910,
    allow=localhost,
    network=local,
    autoupdate=true,
    show=true,
    logoutput=c:\My Documents\project1.txt,
    loglevel=information|warning|error,
    database=c:\My Documents\project1\project1.cdb
}
</pre></td></tr>
</table>

Auto Start with EA	This option automatically starts services having the 'status:ON' attribute when the model opens.  The messages logged to the System Output window here when the model is opened indicate that the service was already running.
Auto Stop on Close	This option automatically stops running services when Enterprise Architect is closed down.

Sparx Intel Service Automatic Update

When you execute the Build command for an Analyzer Script , a job is added to the Job Queue.

If the Build script has the 'Update Codeminer on Completion' checkbox ticked in the Analyzer Script Editor, an additional task is added to the job to update each of the Codeminer databases listed in the script.



The libraries can be seen in the Code Miner | Libraries section of the script.

How the Task Runs

The Code Miner update task runs the program SSCodeMiner.exe with two arguments.

The first argument specifies the database to perform the incremental build on and has this form:

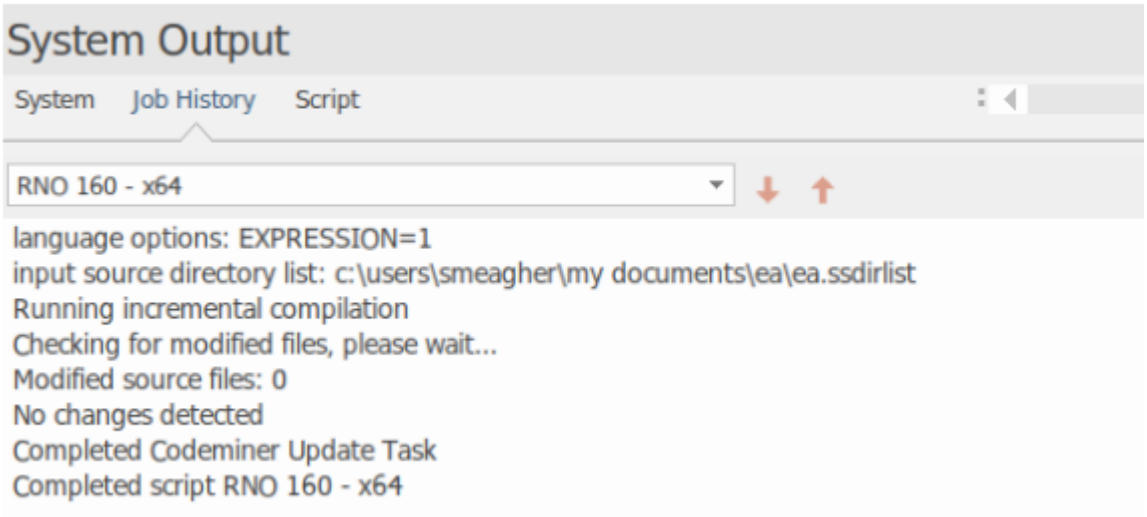
```
update="c:\path\ea.cdb"
```

The second argument is optional and specifies an auxiliary macro grammar file to use when compiling the database; it has this form:

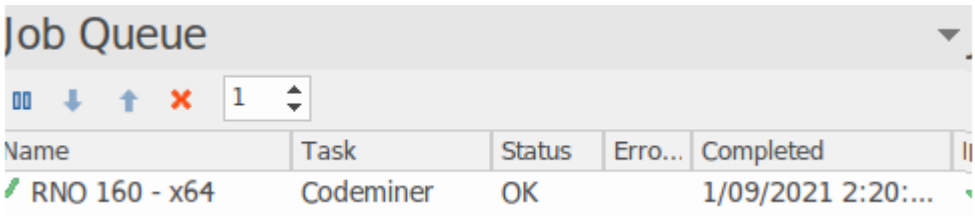
```
macros="c:\ea\ea160\config\CodeMiner\SparxProjectMacros.nbnf"
```

Job Output

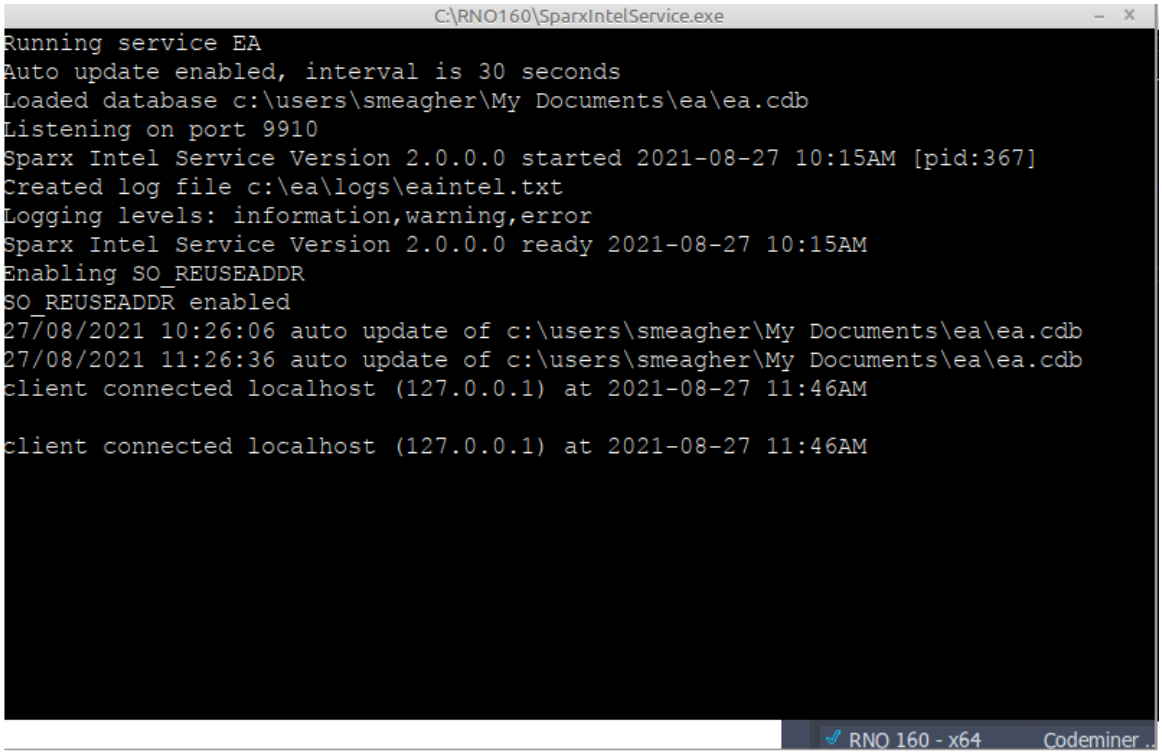
As the Code Miner update task runs, output from the captured SSCodeMiner.exe update process is sent to the 'Job History' tab of the System Output window, in the same form as is displayed when performing a manual update of a Code Miner database in Enterprise Architect. In this illustration we can see that the Analyzer Script RNO 160 -x64 has completed successfully.



The Job Queue window shows that the job has completed. The last task to run was the Code Miner update.



The 'Job History' tab showed that no source code files had changed. If modified source code changes are detected - that is, the Code Miner service has detected a new build of ea.cdb and automatically updated it - this information is displayed:


C:\RNO160\SparxIntelService.exe
Running service EA
Auto update enabled, interval is 30 seconds
Loaded database c:\users\smeagher\My Documents\ea\ea.cdb
Listening on port 9910
Sparx Intel Service Version 2.0.0.0 started 2021-08-27 10:15AM [pid:367]
Created log file c:\ea\logs\eaintel.txt
Logging levels: information,warning,error
Sparx Intel Service Version 2.0.0.0 ready 2021-08-27 10:15AM
Enabling SO_REUSEADDR
SO_REUSEADDR enabled
27/08/2021 10:26:06 auto update of c:\users\smeagher\My Documents\ea\ea.cdb
27/08/2021 11:26:36 auto update of c:\users\smeagher\My Documents\ea\ea.cdb
client connected localhost (127.0.0.1) at 2021-08-27 11:46AM
client connected localhost (127.0.0.1) at 2021-08-27 11:46AM
RNO 160 - x64
Codeminer

Service Configuration

Service program

The name of the service program is SparxintelService.exe.

Configuration File

The service is configured by the file SparxIntelService.config.

The file must be located in the same directory as the service program.

The file contains a number of directives and also lists the Code Miner databases to be served.

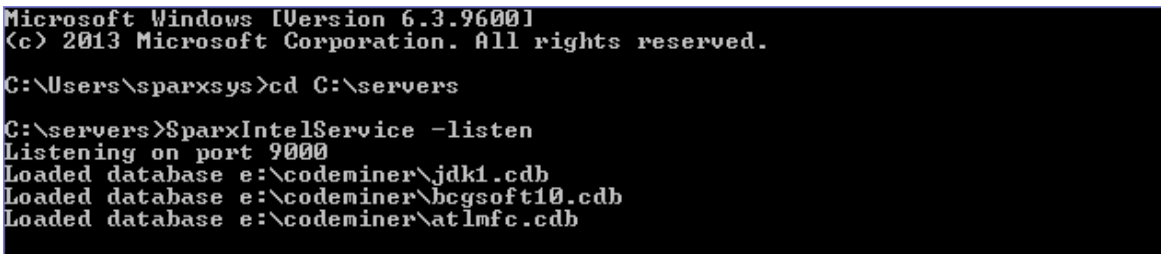
The file is read once when the service is started.

Directives	Description
port	The Port number on which the service will listen.
allow	Names a domain or IP address that is allowed access: 198.* or 127.0.0.1
network	Values can be "public", "network" or "private". • Use "private" when allow directives specify one or more single IP addresses

	• Use "network" when allow directives specify a wildcard domain: 198* • Use "public" to allow all clients
database	Names the full physical file path of a Code Miner database on the server.

Running the program standalone

From a normal console enter the command:
SparxIntelService -listen

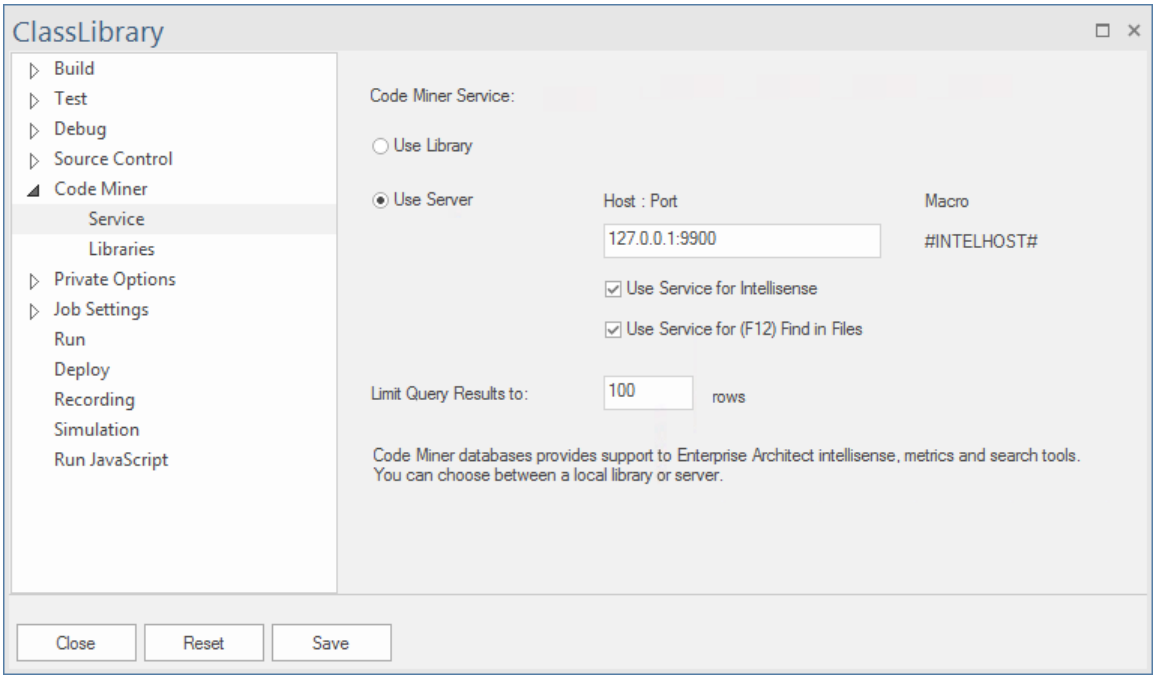


Installing as Windows Service

From an Administrative console enter the command:
SparxIntelService -install

Client Configuration - Configuring Enterprise Architect to Use a Code Miner Service

Enterprise Architect uses components known as Analyzer Scripts for the configuration of many support systems. This is where the location of the server is specified. This image shows the 'Code Miner Service' page of a script.



Access

Ribbon	Develop > Source Code > Execution Analyzer > Edit Analyzer Scripts > Double-click on a Script > Code Miner > Service

Configuration Fields

Use Server	Select this radio button to set up the Code Miner server to use.
Host : Port	Type in the number of the Port through which the Service will operate.
Use Service for Intelli-sense	Select the checkbox to use the Intel Service for Intelli-sense field completion.
Use Service for [F12] Find in Files	Select the checkbox if you want to use the Service instead of the Find In Files window to run search queries, when you press F12.
Limit Query Results to rows	Type in the number of rows of query results to display per page.
Save	Click on this button to save the configuration details you have entered.