

# ArchestrA Object Toolkit

## Developer's Guide

**Invensys Systems, Inc.**

Revision B

Last Revision: 10/20/09



## Copyright

© 2009 Invensys Systems, Inc. All Rights Reserved.

All rights reserved. No part of this documentation shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Invensys Systems, Inc. No copyright or patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this documentation, the publisher and the author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

The information in this documentation is subject to change without notice and does not represent a commitment on the part of Invensys Systems, Inc. The software described in this documentation is furnished under a license or nondisclosure agreement. This software may be used or copied only in accordance with the terms of these agreements.

Wonderware, Inc.  
26561 Rancho Parkway South  
Lake Forest, CA 92630 U.S.A.  
(949) 727-3200

<http://www.wonderware.com>

For comments or suggestions about the product documentation, send an e-mail message to [productdocs@wonderware.com](mailto:productdocs@wonderware.com).

## Trademarks

All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized. Invensys Systems, Inc. cannot attest to the accuracy of this information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

Alarm Logger, ActiveFactory, ArchestrA, Avantis, DBDump, DBLoad, DT Analyst, Factelligence, FactoryFocus, FactoryOffice, FactorySuite, FactorySuite A<sup>2</sup>, InBatch, InControl, IndustrialRAD, IndustrialSQL Server, InTouch, MaintenanceSuite, MuniSuite, QI Analyst, SCADAAlarm, SCADASuite, SuiteLink, SuiteVoyager, WindowMaker, WindowViewer, Wonderware, Wonderware Factelligence, and Wonderware Logger are trademarks of Invensys plc, its subsidiaries and affiliates. All other brands may be trademarks of their respective owners.

# Contents

<b>Welcome.....</b>	<b>11</b>
Documentation Conventions.....	11
Technical Support .....	12
 <b>Chapter 1 Overview and Concepts .....</b>	 <b>13</b>
About the ArchestrA Object Toolkit .....	13
About ApplicationObjects and Primitives .....	14
Workflow: Creating an ApplicationObject or Reusable Primitive .....	17
Tour of the User Interface.....	18
Additions to the Visual Studio Interface.....	19
ArchestrA Object Toolkit Toolbar.....	19
Object Design View .....	20
Logger View .....	21
Object Designer Window.....	22
Opening the Object Designer.....	22
Object Designer Panes .....	23
 <b>Chapter 2 Object Design Considerations.....</b>	 <b>25</b>
Guidelines for Designing the Structure of Control- Oriented Objects.....	25
Limitations to the Complexity of Primitive Hierarchies .....	27
Planning Attribute Usage .....	28

Performance Considerations.....	29
<b>Chapter 3 Working with Projects.....</b>	<b>31</b>
Creating a Project .....	32
Opening an Existing Project.....	33
Moving or Deleting Projects.....	34
Editing Projects in Code or in the ArcestrA Object Toolkit Designer .....	34
<b>Chapter 4 Defining an ApplicationObject.....</b>	<b>35</b>
Configuring the Object's Names and Description .....	36
Configuring Event Handlers.....	37
Configuring Config Time Event Handlers .....	37
Configuring Run Time Event Handlers .....	39
Working with Primitives.....	40
Adding a Local Primitive .....	40
Adding a Reusable Primitive .....	41
Overriding and Locking Attributes of Reusable Primitives .....	43
Deleting a Primitive .....	43
Working with Virtual Primitives.....	43
Defining "Optional" Attributes Using Virtual Primitives .....	45
"Arrays" of Related Primitives .....	45
Naming Considerations for Primitives .....	46
Adding a Custom Object Editor.....	47
Adding ArcestrA Controls to the Visual Studio Toolbox.....	48
Changing the Attribute Reference of ArcestrA Controls .....	49
Configuring Associated Files .....	49
Setting up Rules for Dependent Files .....	50
Setting up Rules for References that Don't Currently Exist in Visual Studio.....	51
Setting up Rules for References that Currently Exist in Visual Studio.....	53
Deleting and Re-Ordering Rules .....	54
Managing the Rules File for All Projects .....	54
Configuring Associated Files Manually .....	55
Configuring Additional Object Properties .....	57
Configuring Dump/Load Support for Dynamic Attributes and Virtual Primitives.....	58

Configuring Failover Support for Run Time	
Dynamic Attributes .....	59
Enabling “Advise Only Active” Support for the Object	60
Configuring the Object’s Minimum Application	
Server Version .....	60
Configuring the Object’s IDE Behavior .....	61
Setting the Object’s Toolset .....	62
Configuring Toolset Names .....	62
Configuring the Object’s Primitive Execution Order ..	63
Associating Different Assemblies with an Object .....	64
Adding Object Help .....	65
Importing an .aaDEF File from a Previous Object	
Version .....	67

## Chapter 5 Defining a Reusable Primitive ..... 69

Switching between Object/Primitive Mode .....	69
Differences Between Editing Objects and Primitives .....	70

## Chapter 6 Configuring Attributes..... 71

Adding Attributes to an Object or Primitive .....	72
Creating a Default Attribute .....	74
Creating a “Hidden” Attribute .....	74
Configuring Config Time Set Handlers .....	74
Example: Configuring a Config Time Set Handler ...	75
Configuring Run Time Set Handlers .....	76
Example: Configuring a Run Time Set Handler .....	77
Configuring Dynamic Attribute Set Handlers .....	78
Example: Configuring a Set Handler for a	
Dynamic Attribute .....	78
Configuring Attribute Extensions .....	79
Historizing an Attribute .....	79
Attributes of the History Primitive .....	81
Making an Attribute Alarmable .....	83
Example: Configuring a Value Alarm for an	
Attribute .....	86
Attributes of the Alarm Primitive .....	87
Adding Inputs and Outputs .....	89
Adding an Input .....	90
Attributes of the Input Primitive .....	91
Adding an Output .....	92
Attributes of the Output Primitive .....	94
Adding an Input/Output .....	95

Attributes of the Input/Output Primitive .....	97
Configuring “Advise Only Active” Support for an Attribute .....	101
Renaming or Deleting Attributes .....	104
 <b>Chapter 7 Internationalizing Objects.....</b>	<b>105</b>
About Internationalizing Objects .....	105
Configuring the Object Dictionary .....	106
Dictionary File Format and Structure .....	107
Editing the Dictionary in Visual Studio .....	107
Retrieving Localized Dictionary Strings.....	108
 <b>Chapter 8 Building and Versioning Objects .....</b>	<b>109</b>
Validating an Object .....	110
Configuring Build Options.....	110
Configuring Output Preferences .....	111
Configuring Galaxy Preferences.....	112
Configuring Additional Search Paths .....	113
Managing an Object's Versions .....	113
Building a New Minor Version of an Object .....	114
Building a New Major Version of an Object .....	115
Creating a New Build without Incrementing the Version Number .....	116
Manually Specifying the Version Number.....	117
Building an Object.....	117
Migrating Objects.....	119
Example: Migrating a Previous Object Version .....	121
Additional Guidelines for Migrating Objects.....	121
 <b>Chapter 9 Debugging Objects .....</b>	<b>123</b>
Attaching the Debugger to the Processes Running the Current Object Version.....	124
Attaching the Debugger during the Build Process.....	125
 <b>Appendix A Programming Techniques</b>	<b>127</b>
Programming Workflow .....	127
Configuring Internal and External Names.....	129
Providing Wrappers for Referencing ArcestraA Attributes .....	130
Config Time Coding.....	131

Config Time Set Handler .....	131
Set Handler Code .....	132
Performing Config Time Validation with the ConfigtimeValidate() Method .....	132
Adding a Virtual Primitive at Config Time with AddPrimitive .....	133
Removing a Virtual Primitive at Config Time with DeletePrimitive .....	134
Accessing Data in Attributes at Config Time .....	136
Accessing Data in Other Primitives at Config Time ..	136
Adding and Deleting Dynamic Attributes at Config Time .....	136
Run Time Coding.....	137
Runtime SetHandler .....	137
Set Handler Code .....	138
SetInfo Structure Event Arguments .....	138
Coding a RuntimeExecute() Method .....	138
Returning an Error Status String at Run Time .....	139
RuntimeGetStatusDesc Event.....	139
Event Handler for Get Status Description.....	140
Manipulating Data Quality at Run Time .....	140
Manipulating the Timestamp at Run Time .....	140
Getting Input (I/O) Values Using Utility Primitives at Run Time.....	141
Setting Output (I/O) Values Using Utility Primitives at Run Time.....	141
Accessing Data in Attributes at Run Time .....	142
Accessing Data in Other Primitives at Run Time .....	142
Adding and Deleting Dynamic Attributes at Run Time .....	143
Supporting AdviseOnlyActive at Run Time.....	143
AdviseOnlyActiveEnabled .....	144
Other AOT Wrappers for AdviseOnlyActive.....	145
IO Utilities .....	145
Triggering an Alarm at RunTime.....	145
Providing Access to External Attributes (BindTo) .....	145
CMxIndirect.....	147
Associating an Archestra Editor Control with an Attribute in Code.....	147
Referencing Attributes Using GetValue and SetValue .....	148
Local References .....	149
Referencing Down (child) .....	149
Referencing Up (parent).....	149

Array Usage.....	150
The External Build Process .....	150
Command Line Recompile Object .....	150
Command Line Repackage Object.....	151
Advanced Techniques.....	151
Configuring an Arcestra Attribute in Code.....	152
Specifying the Arcestra Attribute Array Length ..	154
Referencing Attributes from the Editor of the Object	154
Local Attribute Wrappers .....	155

## **Appendix B Development Best Practices 157**

General Guidelines.....	157
Naming Conventions.....	157
Naming Restrictions .....	158
Arcestra Naming Standards and Abbreviations..	158
Additional Naming Guidelines .....	160
Creating a Logical Attribute Hierarchy.....	161
Using “Unnamed” Primitives .....	161
Using Periods in Attribute Names .....	161
Working with the Logger .....	162
Raising Data Change Events.....	162
Changing or Enforcing the Length of an Array .....	162
Guidelines for Config Time Code Development.....	163
Ensuring Galaxy Dump/Load Support.....	163
Determining the Configuration Status .....	164
Changing an Attribute’s Data Type at Config Time ..	164
Guidelines for Run Time Code Development.....	165
Returning Warnings During Deployment.....	165
Avoiding Application Engine "Overscans" .....	165
OnScan/OffScan Behavior .....	165
Dealing with Quality .....	166
Best Practices for Dealing with Quality .....	167
Dealing with Timestamps.....	168
Dealing with Outputs on Object Startup .....	169
Dealing with the Quarantine State.....	170
Ensuring Failover Support for Run Time Dynamic Attributes .....	170
Guidelines for Custom Editor Development.....	171
Keeping Validation Rules out of the Editor Code.....	171
Creating a Complete Editor.....	171



## Appendix C Sample Projects 173

The Monitor Object.....	173
Object Structure .....	174
Custom Object Editor.....	174
Run Time Code .....	174
The WatchDog Object.....	175
Object Structure .....	175
Custom Object Editor.....	177
Config Time Code .....	177
Object Run Time Code.....	177
Stats Primitive Run Time Code.....	178

## Appendix D ArcestrA Data Types 179

List of ArcestrA Data Types .....	179
Coercion Rules for ArcestrA Data Types .....	181
Coercion from Boolean Values .....	181
Coercion from Integer Values .....	182
Coercion from Float or Double Values .....	182
Coercion from String or Big String Values .....	183
Coercion from Time Values.....	183
Coercion from Elapsed Time Values .....	184
Coercion from MxStatus Values .....	184
Coercion from Data Type Values.....	184
Coercion from Custom Enumeration Values .....	184
Coercion from Custom Structure Values .....	185
Using Data Types Correctly.....	185
Custom Enumeration vs. Integer .....	185
Absolute and Elapsed Times.....	185
Internationalized String .....	186
Big String.....	186
Attribute References .....	186
Variant (Unspecified) Data Type.....	187
Arrays.....	187

## Appendix E ArcestrA Attribute Categories 189

## Appendix F ArcestrA Security Classifications 193

Index .....	195
-------------	-----



# Welcome

This guide shows you how to create Arcestra ApplicationObjects using the Arcestra Object Toolkit in Microsoft Visual Studio 2008.

It explains how to use the Arcestra Object Toolkit Object Designer and how to configure objects by using this editor. It does not contain reference information that you may need when editing your object's code directly, such as information on functions, methods and data structures. For this type of information, see the *Arcestra Object Toolkit Reference Guide*.

You can view this document online or you can print it, in part or whole, by using the print feature in Adobe Acrobat Reader.

This guide assumes that you are familiar with Wonderware Application Server. It also assumes that you have at least some basic experience with C# development using Microsoft Visual Studio. If you are not familiar with Microsoft Visual Studio, see the Microsoft documentation.

## Documentation Conventions

This documentation uses the following conventions:

Convention	Used for
Initial Capitals	Paths and file names.
<b>Bold</b>	Menus, commands, dialog box names, and dialog box options.
Monospace	Code samples and display text.

## Technical Support

Wonderware Technical Support offers a variety of support options to answer any questions on Wonderware products and their implementation.

Before you contact Technical Support, refer to the relevant section(s) in this documentation for a possible solution to the problem. If you need to contact technical support for help, have the following information ready:

- The type and version of the operating system you are using.
- Details of how to recreate the problem.
- The exact wording of the error messages you saw.
- Any relevant output listing from the Log Viewer or any other diagnostic applications.
- Details of what you did to try to solve the problem(s) and your results.
- If known, the Wonderware Technical Support case number assigned to your problem, if this is an ongoing problem.

# Chapter 1

## Overview and Concepts

This section gives you a general overview of the Wonderware ArchestrA Object Toolkit and its features. We recommend that you read this section in its entirety to familiarize yourself with the key concepts, and then continue with the other sections for detailed information on specific tasks.

### About the ArchestrA Object Toolkit

The ArchestrA Object Toolkit is an add-on to Microsoft Visual Studio that lets you create custom ArchestrA ApplicationObjects in C# code. It provides an easy-to-use graphical Object Designer while still giving you full control over the object's source code.

Using the ArchestrA Object Toolkit, you can:

- Create custom ApplicationObjects for use in Wonderware Application Server without having to deal with its internals.
- Create a custom editor for your object that allows your users to easily configure the object's attributes using the Application Server IDE.
- Create reusable primitives, code modules that you can reuse in multiple custom ApplicationObjects.

- Easily navigate to all sections of your object's code using an Object Design View.
- Configure object attributes using an easy-to-use Object Designer.
- Build .aaPDF files and automatically import, instantiate and deploy them in your Galaxy for testing.

---

**Note** You can *not* create DeviceIntegration Objects with this release of the ArcestrA Object Toolkit. For information on creating DeviceIntegration Objects, please contact your Wonderware distributor.

---

## About ApplicationObjects and Primitives

**ApplicationObjects** are domain-specific objects that represent plant equipment and instrumentation such as pumps, valves, temperature transmitters, or conveyors. They usually get their source data from other objects, such as DeviceIntegration objects.

Wonderware Application Server already includes some basic ApplicationObject templates, such as the \$AnalogDevice object for a simple analog device. Using the ArcestrA Object Toolkit, you can create complex custom ApplicationObjects that represent specific types of equipment, for example, a pump system. These ApplicationObjects support various events that allow you to execute custom config time and run time code. You can also create a custom object editor for easy configuration.

**Attributes** are the data items of an ApplicationObject or primitive. By reading from and writing to attributes, objects can exchange data with each other. (Unless specifically noted, when this manual talks about “attributes,” we mean these ArcestrA attributes, not C# attributes.)

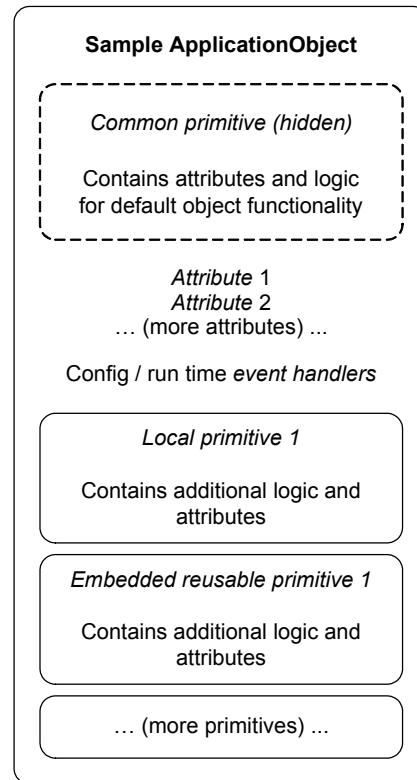
**Primitives** are modules containing code and attributes. Like the main ApplicationObject, they have their own config time and run time event handlers as well. You can think of them as the “building blocks” for your object. By “modularizing” your object into primitives, you can create a clear, logical structure and dynamically enable or disable functionality as needed. There are two types of primitives:

- *Local primitives* are defined locally in an object and are only used in that single object.
- *Reusable primitives* are “stand-alone” primitives that can be reused in multiple ApplicationObjects. Organizing common functionality into reusable primitives makes it easier to maintain your code. The Arcestra Object Toolkit includes predefined utility primitives for various tasks, such as historizing values to a database. You can also create custom reusable primitives for your objects.

Both local and reusable primitives can be made “virtual.” This allows you to dynamically add and delete instances of the primitive at config time. If you don’t add any instances of the primitive, it doesn’t become part of the object instance and is not deployed at run time. This reduces the run time processing load because only the functionality that is actually needed gets deployed.

All ApplicationObjects automatically contain the “Common” primitive, which provides common attributes and functionality that all objects need (for example, attributes for setting the object OnScan/OffScan, etc.) The Common primitive is hidden, and you can’t edit it in any way.

For example, a simple `ApplicationObject` might have the following structure:

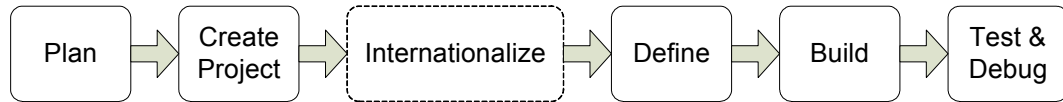


The structure of an `ApplicationObject` or primitive, as composed of its attributes and child primitives, is also called its “shape.”



## Workflow: Creating an ApplicationObject or Reusable Primitive

The basic steps to create an ApplicationObject or reusable primitive are as follows:



- 1 Planning.** When developing an ApplicationObject, you should consider certain requirements and best practices. For more information, see Chapter 2, Object Design Considerations.
- 2 Creating a project in Visual Studio.** Working with an Arcestra Object Toolkit project is similar to working with other projects in Visual Studio. For more information, see Chapter 3, Working with Projects.
- 3 Internationalizing the object** (optional). If your object will be used in different localized environments, you can define multilingual strings for your object's messages and other text. At run time, the object can then use the appropriate local language strings for the environment it's used in. It is best to internationalize any required messages right from the start and then refer to them as required as you write your code. For more information, see Chapter 7, Internationalizing Objects.
- 4 Defining the object or primitive.** In this step, you configure basic object properties and add the attributes (data items) that your object or primitive will use. You also add any code procedures that your object will need, such as run time set handlers or event handlers. Finally, you create a custom object editor that your users can use to configure the object's properties in the Application Server IDE.

For more information on defining ApplicationObjects and reusable primitives, see Chapter 4, Defining an ApplicationObject, and Chapter 5, Defining a Reusable Primitive. For information on configuring attributes, see Chapter 6, Configuring Attributes.

When coding your object, follow the “best practices” and guidelines outlined in Appendix B, Development Best Practices.

**5 Building the object.** In this step, you create an .aaPDF object file that contains your custom ApplicationObject (or an .aaPRI file if you're developing a reusable primitive). You can then import and use the object in Application Server. You can also import, instantiate and deploy the object automatically as part of the build process, and you can configure various build options. For more information, see Chapter 8, Building and Versioning Objects.

**6 Testing and debugging the object.** If you encounter problems while testing your object, see Chapter 9, Debugging Objects, for troubleshooting hints.

The ArcestraA Object Toolkit includes some sample projects to help you get started. For more information, see Appendix C, Sample Projects.

---

**Important** To build objects on Windows Vista and later operating systems, you must run Visual Studio with administrative privileges.

---

## Tour of the User Interface

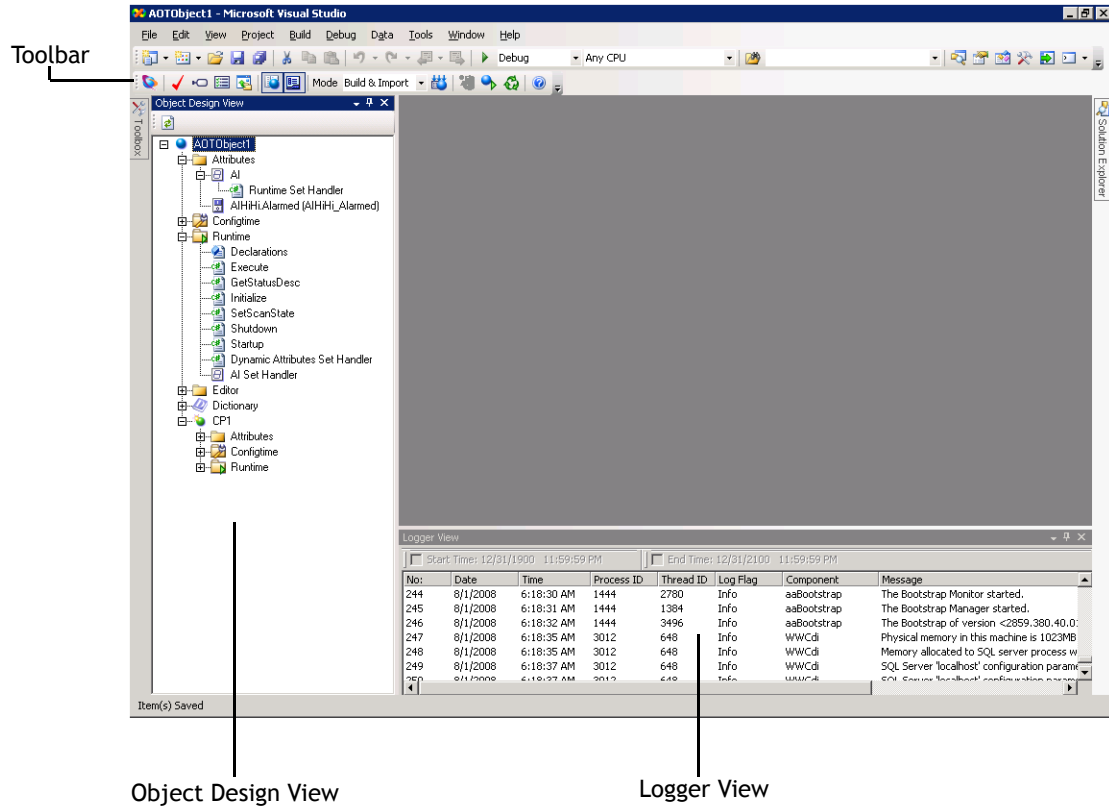
The user interface of the ArcestraA Object Toolkit consists of:

- **Additions to the regular Microsoft Visual Studio user interface:** a toolbar, an Object Design View and a Logger view. By default, these are always visible when an ArcestraA Object Toolkit project is opened in Visual Studio.
- An **Object Designer window** that you can open and close as required while working with an ArcestraA Object Toolkit project.

The following sections describe each of these components.

## Additions to the Visual Studio Interface

When you create or open an ArcestraA Object Toolkit project in Visual Studio, the Visual Studio environment shows extra items.



**Note** When you create your first project after installing the ArcestraA Object Toolkit, the Object Design View and Logger View are not docked. We recommend that you dock the Object Design View to the left of the Visual Studio window, and the Logger view to the bottom (as shown above).

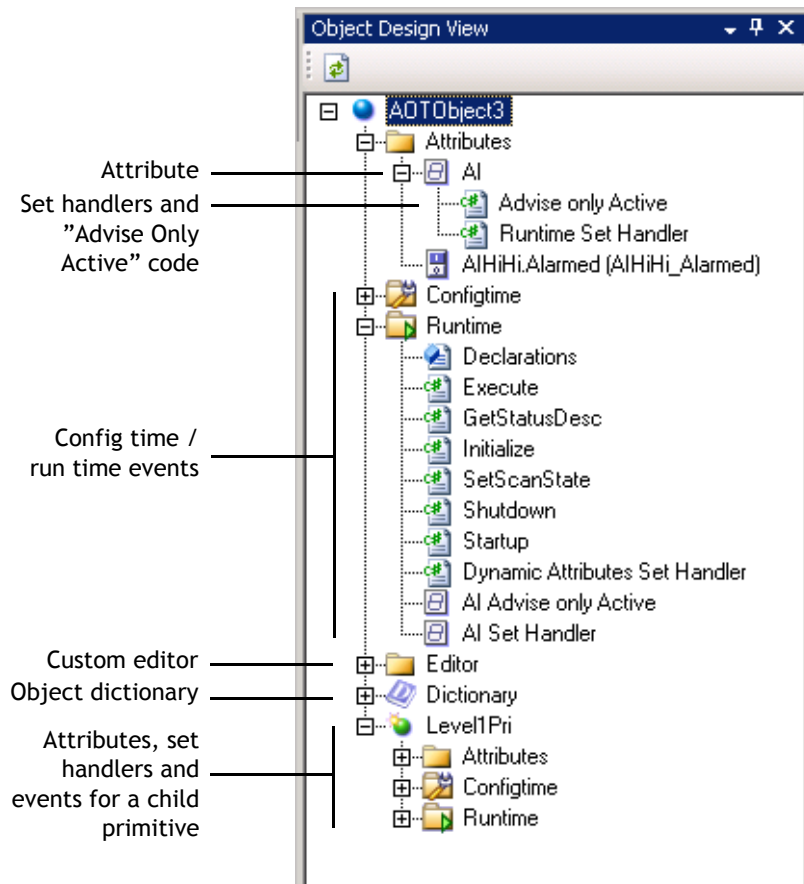
### ArcestraA Object Toolkit Toolbar

The ArcestraA Object Toolkit toolbar lets you access all main features of the ArcestraA Object Toolkit. The individual features and icons are described throughout this documentation.



## Object Design View

The Object Design View allows you to easily navigate your object's code. Double-clicking on an item in the Object Design View automatically opens the code section that controls the respective aspect of your object. For example, the Object Design View might look like this:



- **Attributes:** Double-click an attribute name to see its definition in the code.

- **Attribute set handlers and “Advise Only Active” code:** Double-click **Configtime Set Handler** or **Runtime Set Handler** to open the code section that contains the config time or run time set handler for an attribute (if enabled). Double-click **Advise only Active** to open the “Advise only Active” code section for an attribute (if enabled)
- **Config time/run time events:** Double-click an event name to open the code section linked to the event. Once you have added custom code for an event, its name is shown in bold type.
- **Custom editor:** Double-click **Editor** to open the object’s custom editor in the Visual Studio design view.
- **Object dictionary:** Double-click **Dictionary** to edit the object dictionary. Expand this item to see the content of the dictionary.



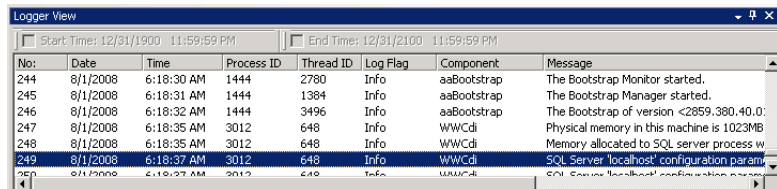
To refresh the Object Design View after making changes to the code, click its **Refresh** icon. This automatically validates the code as well.



To hide or re-open the Object Design View, click the **Object Design View** icon in the ArchestrA Object Toolkit toolbar.

## Logger View

The Logger view shows the same ArchestrA Logger messages that you would see in the ArchestrA Log Viewer. Check this view for any errors and warnings that may be reported by the ArchestrA Object Toolkit. The Logger view is intended for quick reference. It does not offer the full functionality of the ArchestrA Log Viewer.



No:	Date	Time	Process ID	Thread ID	Log Flag	Component	Message
244	8/1/2008	6:18:30 AM	1444	2780	Info	aaBootstrap	The Bootstrap Monitor started.
245	8/1/2008	6:18:31 AM	1444	1384	Info	aaBootstrap	The Bootstrap Manager started.
246	8/1/2008	6:18:32 AM	1444	3496	Info	aaBootstrap	The Bootstrap of version <2859.380,40.0>
247	8/1/2008	6:18:35 AM	3012	648	Info	WWCdi	Physical memory in this machine is 1023MB
248	8/1/2008	6:18:35 AM	3012	648	Info	WWCdi	Memory allocated to SQL server process w
249	8/1/2008	6:18:37 AM	3012	648	Info	WWCdi	SQL Server 'localhost' configuration param
250	8/1/2008	6:18:37 AM	3012	648	Info	WWCdi	SQL Server 'localhost' configuration param



To hide or re-open the Logger View, click the **Logger View** icon in the ArchestrA Object Toolkit toolbar.

## Object Designer Window

The Object Designer lets you easily edit and configure your object's general properties, attributes and primitives. We recommend that you edit your objects using this editor. However, you can always edit all aspects of your object directly in the underlying code.

The code and the data you see in the Object Designer are always synchronized. For example, when you change the properties of an attribute in the code, you see the updated property values the next time you open the Object Designer, and vice versa.

### Opening the Object Designer

You can open and close the Object Designer as required while working with your project. You can also have the Object Designer open automatically when you open an Arcestra Object Toolkit project in Visual Studio.

#### To open the Object Designer



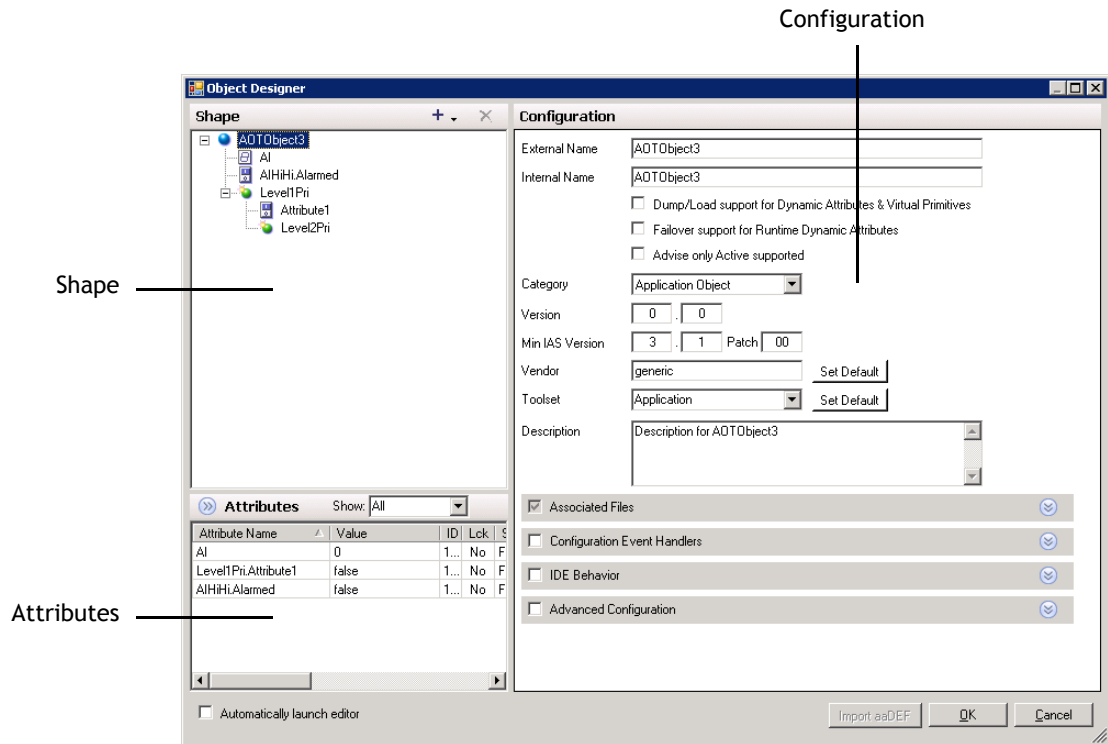
- ◆ Click the **Object Designer** icon in the Arcestra Object Toolkit toolbar.

#### To have the Object Designer open automatically when opening a project

- 1 Open the Object Designer.
- 2 In the bottom left corner of the Object Designer, select the **Automatically launch editor** check box.
- 3 Click **OK**.

## Object Designer Panes

The Object Designer contains the following panes:



- **Shape pane:** This pane shows the structure (“shape”) of your object. You can edit your object here by adding and deleting attributes and primitives.
- **Configuration pane:** Use this pane to configure the item that is currently selected in the **Shape** pane.
- **Attributes pane:** This pane shows a list of all custom attributes in your object, including their configuration. You can expand this pane to see more information at once. Also, you can choose to see config time or run time attributes only. The abbreviated column headings mean the following:



Heading	Description
FA	“Frequently Accessed” option is enabled
CQ	“Calculated Quality” option is enabled
Cfg	Config time set handler is enabled
Run	Run time set handler is enabled
Al	Attribute is alarmed
His	Attribute is historized





# Chapter 2

## Object Design Considerations

Before you start defining your object, you should plan its features and structure. You should:

- Decide how to design the object's structure by using primitives.
- Make a list of the attributes that your object and any of its primitives will need, and develop a naming structure.

### Guidelines for Designing the Structure of Control-Oriented Objects

This section describes basic guidelines for developing objects that perform feedback control actions at the supervisory level. The AnalogDevice, configured as an AnalogRegulator, and the DiscreteDevice objects that are provided with Wonderware Application Server provide good working examples of how to design such objects.

Developing objects in accordance with these guidelines ensures that objects are structured in a manner that fits well within the overall architecture of ArchestrA, and that they offer a consistent structure to end users.

In general, you should separate control objects into three primary blocks: Feedback processing, Control processing, and one or more Alarm processing blocks.

- The **Feedback** processing block includes the attributes and logic for reading field inputs and processing them to derive a primary calculated result that is placed into an attribute called the “PV” (process value). The feedback processing should also include a “PVMode” attribute that determines whether the PV is calculated automatically or set manually.
- The **Control** processing block includes the attributes and logic for receiving setpoints or commands that result in output control actions. The control processing should also include a “CtrlMode” (control mode) attribute that determines whether the control actions are commanded manually or cascaded from another object. The run time set handler for the control attributes can then reject set requests coming from disallowed sources.
- The **Alarm** processing block(s) includes the attributes and logic for alarm detection for the feedback and control blocks. The feedback alarming determines whether the input(s) are abnormal or unexpected. The control alarming determines whether control actions are failing or have had unexpected results. You can implement these blocks of logic as part of the Feedback or Control processing blocks or separately, depending on what makes sense for your object.

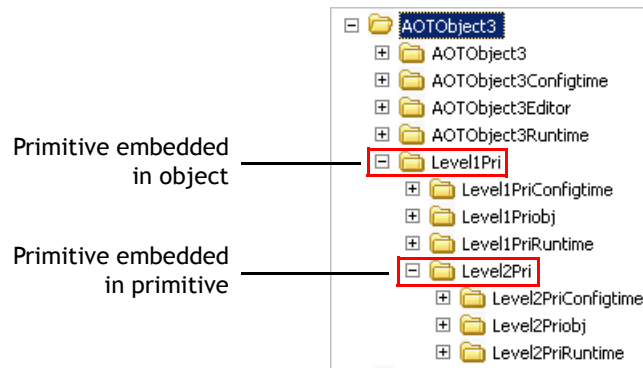
By using primitives, you can model these multiple blocks in a clean and easy way. The ArcestrA Object Toolkit also allows you to develop “virtual” primitives that allow you to add primitive instances as required at config time. For more information, see Working with Virtual Primitives on page 43.

In many objects, it will be most appropriate to separate the object into a top-level primitive that brings in an optional child Control virtual primitive. When the user enables Control functionality, then the config time logic adds the Control primitive dynamically. Since alarms are commonly optional, it also makes sense to use virtual primitives for one or more blocks of alarm detection functionality. The \$AnalogDevice object that is provided with Application Server provides a good example of these concepts.

## Limitations to the Complexity of Primitive Hierarchies

When you arrange local primitives in a hierarchy (that is, embed primitives within other primitives, and so on), there is a limit to the number of levels the hierarchy can have.

When you add a local primitive to an ApplicationObject, the ArchestrA Object Toolkit adds a subfolder to the solution folder that contains the code files for that primitive. When you add another primitive to that primitive, it creates a subfolder in the primitive subfolder, and so on. Each level in the primitive hierarchy corresponds to another level of folders in the ArchestrA Object Toolkit solution folder. (For more information on the content of the solution folder, see *Creating a Project* on page 32.)



However, the Windows operating system imposes a limit on the maximum length of a file path. On current Windows versions, this limit is 260 characters for the complete path including all formatting characters. No path to any code file in any ArchestrA Object Toolkit project folder may exceed this limitation. Therefore, there is a limit to how many levels of primitives you can use.

The exact number of levels depends on the path length of the base ArchestrA Object Toolkit solution folder and on the length of the primitive names. The shorter the primitive names, the shorter the folder names, and the more levels you can use.

## Planning Attribute Usage

Attributes are the data items of an object or primitive. By reading from and writing to attributes, objects can exchange data with each other. In addition, your objects can also have Inputs and Outputs to communicate with other attributes in the ArcestrA environment.

Before you start developing your object, you should determine the following:

- **Which attributes do you need?** As a guideline, only expose those data items as attributes that your users actually need to see and/or change. For data items that you'll only need locally or temporarily in the object's code, simply use local C# variables. Those attributes that you do expose should be organized and named in a way that is intuitive to your users. Also determine whether your attributes will require I/O capabilities.
- **What level of access should users have to these attributes?** There might be attributes that are only relevant at config time, but not at run time. For other attributes, you might want to allow run time write access only to users with certain privileges.

In the ArcestrA Object Toolkit, you can use three strategies to control attribute structure and access:

- **Developing a logical naming hierarchy.** You can use named and unnamed primitives as well as structured attribute names to organize attributes logically and only expose those attributes that the user really needs. For more information, see *Creating a Logical Attribute Hierarchy* on page 161.
- **Security classifications.** By specifying different security classifications for individual attributes, you can restrict attribute access to users with the right privileges. For more information, see *Appendix F, ArcestrA Security Classifications*.
- **Attribute categories.** By using different attribute categories, you can specify whether or not an attribute should be available and readable/writable at config time and/or run time. For more information, see *Appendix E, ArcestrA Attribute Categories*.

## Performance Considerations

There is no arbitrary limit to the number of primitives, attributes, and inputs/outputs that you can add to your `ApplicationObject`. As with all software, the more complex your object gets, the more resources it will need. Depending on your system resources, there will eventually be a practical limit where performance becomes unacceptable. However, under normal circumstances, this should not be an issue except for very large and complex `ApplicationObjects`.

For additional restrictions on the complexity of primitive hierarchies, see [Limitations to the Complexity of Primitive Hierarchies](#) on page 27.



# Chapter 3

## Working with Projects

When creating ApplicationObjects or reusable primitives, you manage your development work in *projects*. ArchestrA Object Toolkit projects are simply Visual Studio projects of a special type. You create and manage them just like other Visual Studio projects.

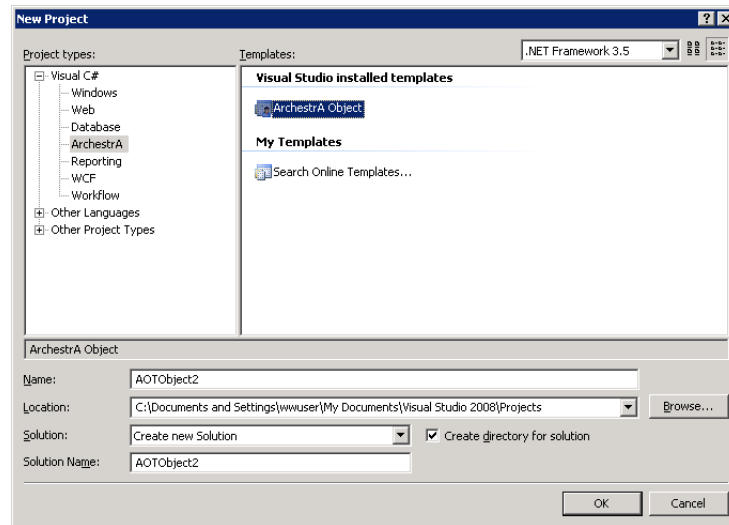
You create a project for every ApplicationObject or reusable primitive that you want to create. Visual Studio then creates a solution folder with subfolders for the projects corresponding to each of your object's components and primitives. That solution folder contains all code related to the ApplicationObject or reusable primitive you're developing.

## Creating a Project

When you create a project, the Arcestra Object Toolkit creates all files and basic structures that you need to define your ApplicationObject or reusable primitive. The steps to create a project are the same for ApplicationObjects and reusable primitives.

### To create a project

- 1 Open Microsoft Visual Studio.
- 2 On the **File** menu, point to **New** and then click **Project**. The **New Project** dialog box appears.



- 3 In the **Project types** list, expand **Visual C#** and click **Arcestra**.
- 4 In the **Templates** pane, click **Arcestra Object**.
- 5 In the **Name** box, enter a name for the project. This is also used as the name of your object. You can change the object name later.
- 6 In the **Location** box, enter the path where you want the project folder to be created.
- 7 In the **Solution** list, leave **Create New Solution** selected. Leave the **Create directory for solution** check box selected as well.
- 8 Click **OK**.



Visual Studio creates a solution folder for the ArcestrA Object Toolkit project. The solution folder contains the following subfolders:

Subfolder	Contents
<Project name>	Code files that define the object shape and attributes
<Project name>Configtime	Code files for the object's config time ("package") code
<Project name>Editor	Code files for the custom object editor UI and code
<Project name>Runtime	Code files for the object's run time code
Output	Build output (.aaPDF or .aaPRI files, .aaDEF file). This folder is created when you build your project for the first time.

When you add a local primitive to an ApplicationObject, the ArcestrA Object Toolkit adds a subfolder to the solution folder that contains the code files for that primitive. The primitive folder, in turn, has different subfolders for config time and run time code similar to the ones described above. When you add another primitive to that primitive, the ArcestrA Object Toolkit creates a subfolder in the primitive subfolder, and so on. For limitations due to this approach, see *Limitations to the Complexity of Primitive Hierarchies* on page 27.

## Opening an Existing Project

When you open an existing ArcestrA Object Toolkit project in Visual Studio, the ArcestrA Object Toolkit windows appear, and all related commands are available. When you open any other type of project, these windows and commands are not available, except for some commands that are not project-specific.

### To open an existing project

- 1 On the **File** menu, point to **Open** and then click **Project/Solution**. The **Open Project** dialog box appears.
- 2 Select the solution file you want to open and click **Open**. Visual Studio opens the project.

## Moving or Deleting Projects

To move or delete an ArcestraA Object Toolkit project, simply move or delete the entire Visual Studio solution folder. When moving a project, pay attention to the following:

- The Windows folder structure only allows paths to be a certain length. If you move a project with a long hierarchy of child primitives (which are stored in nested subfolders) to an already long path, some of the paths may become too long.
- After you have moved the project, check your code for any relative references to dependent files that may need updating.
- If you are moving a project from one computer to another, make sure that all references are available on the new computer.

## Editing Projects in Code or in the ArcestraA Object Toolkit Designer

The ArcestraA Object Toolkit provides a graphical Object Designer that makes it easy to configure the properties and attributes of your ApplicationObject (or reusable primitive). We recommend that you edit your objects using this Object Designer. However, you can always edit all aspects of your object directly in the underlying code.

The code and the data you see in the Object Designer are always synchronized. For example, when you change the properties of an attribute in the code, you see the updated property values the next time you open the Object Designer, and vice versa.

This manual describes how to edit ApplicationObjects and reusable primitives by using the Object Designer. For more information on editing properties and attributes in code, see the *ArcestraA Object Toolkit Reference Guide*.

For information on opening the Object Designer interface, see Object Designer Window on page 22.

# Chapter 4

## Defining an ApplicationObject

Once you have created an Arcestra Object Toolkit project, you can start defining your object. This section explains how to configure object properties and add primitives using the Object Designer, and how to add custom code using the Object Design View. For information on configuring attributes, see Chapter 6, *Configuring Attributes*.

Common steps when defining your object are:

- Configuring the object's names and description. See *Configuring the Object's Names and Description* on page 36.
- Adding code to the object's various event handlers. See *Configuring Event Handlers* on page 37.
- Working with primitives to structure your object. See *Working with Primitives* on page 40.
- Adding a custom object editor that allows end users to configure the object in the Arcestra IDE. See *Adding a Custom Object Editor* on page 47.
- Configuring associated files in case your object uses any external files or assemblies. See *Configuring Associated Files* on page 49.

- Configuring other object properties, including “Advise Only Active” support, the object’s behavior in the ArcestrA IDE, and dump/load or failover support for dynamic attributes. See [Configuring Additional Object Properties](#) on page 57.
- Adding object help that end users can access from the ArcestrA IDE. See [Adding Object Help](#) on page 65.

If you have an existing ApplicationObject developed with a previous version of the ArcestrA Object Toolkit, you can easily import its shape by importing the object’s .aaDEF file. For more information, see [Importing an .aaDEF File from a Previous Object Version](#) on page 67.

You can also configure object properties directly in the code. For more information, see the *ArcestrA Object Toolkit Reference Guide*.

## Configuring the Object's Names and Description

You can configure an object’s internal and external names and give it a meaningful description. The *internal name* is the name by which you can refer to the object from within its code. The object’s *external name* and description are used in the ArcestrA IDE. It is also used to create default names for object instances.

You can also configure an object’s vendor name. This name shows the end user who created the object and is used to uniquely identify the object for upgrade purposes.

For more information on ArcestrA naming conventions and restrictions, see [Naming Conventions](#) on page 157.

---

**Note** For brevity, do not use the word “Object” or “Template” in an object’s name.

---

### To configure an object’s names and description



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the topmost node. The object properties appear in the **Configuration** pane.
- 3 In the **External Name** box, enter the object’s external name. The name must be ArcestrA compliant. The maximum length is 31 characters.
- 4 In the **Internal Name** box, enter the object’s internal name. The name must be C# compliant. The maximum length is 329 characters.

- 5 In the **Vendor Name** box, enter the vendor name. This name must not contain double-byte characters or any characters that are invalid in Windows file names.
- 6 In the **Description** box, enter the object description. The maximum length is 1024 characters.
- 7 Click **OK**.

## Configuring Event Handlers

Event handlers are the main place where you will add the custom code for your ApplicationObject. The object and each of its local primitives have a variety of config time and run time events that you can link with custom handler code. The following sections describe which events exist, and how you can associate them with code.

You can also execute custom code when the value of an attribute is changed at config time or run time. For more information, see [Configuring Config Time Set Handlers](#) on page 74 and [Configuring Run Time Set Handlers](#) on page 76.

## Configuring Config Time Event Handlers

ArchestrA ApplicationObjects support a number of “config time” events that are triggered when a user works with the object in the ArchestrA IDE. By implementing handlers for these events, you can link configuration actions with custom code. For example, you could execute certain code after an object instance is created.

---

**Note** Config time event handler code is executed only on the Galaxy Repository node. Therefore, it cannot directly interact with the user. For example, if you call a message box within an event handler, the message box appears on the Galaxy Repository node.

---

All ApplicationObjects have the following standard config time event handlers enabled:

---

Event	Occurs
Initialize	When the object is initialized. Use this event handler for any custom initialization code.
Migrate	When derived templates or instances are migrated. See <a href="#">Migrating Objects</a> on page 119.

---

Event	Occurs
PostCreate	After the object (instance or derived template) is created.
PreValidate	Before the object is validated (when the user has edited its configuration and saves it).
Validate	When the object is validated. Use this event handler for any custom validation code (e. g. checking for invalid combinations of attribute values).

**To add code to a config time event handler**

- 1 In the Object Design View, expand the **Configtime** folder.
- 2 Double-click the event name. The code section for the config time event handler appears.
- 3 Enter the code for the config time event handler. When you are done, save your changes.

## Configuring Run Time Event Handlers

Archestra ApplicationObjects support a number of run time events that are triggered as the object is executed at run time. By implementing handlers for these events, you can link custom code with these events. For example, you could execute certain code on every scan cycle.

All ApplicationObjects have the following run time event handlers:

Event	Occurs
Execute	On every scan cycle of the hosting AppEngine while the object is OnScan.
GetStatusDesc	When the run time component requests a detailed message for an error, e. g. after a set handler returns a failure.
Initialize	After the object is created in run time (usually after deployment, but also after a failover or after the bootstrap is restarted). Occurs before the Startup event. No attribute information is available at this time.
SetScanState	When the object's scan state (OnScan/OffScan) is changed.
Shutdown	When the object is shutting down (usually after the object is set to OffScan, but during a failover OffScan may not be set). This event does not occur if the object goes off the network during a network failure.
Startup	When the object is being started (after the Initialize event and before it goes OnScan). You can use the event's startup context to find out whether the object is starting up after a failover or other reasons.

### To add code to a run time event handler

- 1 In the Object Design View, expand the **Runtime** folder.
- 2 Double-click the event name. The code section for the run time event handler appears.
- 3 Enter the code for the run time event handler. When you are done, save your changes.

## Working with Primitives

By using primitives, you can structure your object's code and attributes logically and efficiently. If you define a primitive as virtual, you can also enable or disable instances of it as required at config time. For more information, see *Working with Virtual Primitives* on page 43.

---

**Note** When you arrange local primitives in a hierarchy (that is, embed primitives within other primitives, and so on), there is a limit to the number of levels the hierarchy can have. For more information, see *Limitations to the Complexity of Primitive Hierarchies* on page 27.

---

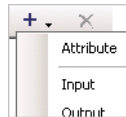
### Adding a Local Primitive

Local primitives are defined locally in an object and are only used in that single object.

#### To add a local primitive



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the primitive. For example, if you want to add a primitive to another primitive, select that primitive or one of its attributes.
- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Local Primitive**. The properties of the new primitive are shown in the **Configuration** pane.
- 5 In the **External Name** box, enter a unique external name for the primitive. This is the name by which other objects can access the primitive's attributes. The name must be ArcestraA compliant. For applicable restrictions, see *Naming Conventions* on page 157.  
You can also leave it blank. For more information, see *Naming Considerations for Primitives* on page 46.
- 6 In the **Internal Name** box, enter a unique internal name for the primitive. This is the name by which you can refer to the primitive in the object's code. The name must be C# compliant. The maximum length is 329 characters.
- 7 If required, select the **Dump/Load support for Dynamic Attributes & Reusable Primitives** and **Failover support for Runtime Dynamic Attributes** check boxes. For more information on these options, see *Configuring Dump/Load Support for Dynamic Attributes and Virtual Primitives* on page 58 and *Configuring Failover Support for Run Time Dynamic Attributes* on page 59.





- 8 Select the **Virtual** check box if the primitive should be virtual. For more information, see Working with Virtual Primitives on page 43.
  - 9 If required, select the **Advanced Configuration** check box to specify additional options:
    - a In the **Execution Group** list, select the execution group for the primitive. This determines the execution order of the object's primitives within each scan of the AppEngine. For more information, see Configuring the Object's Primitive Execution Order on page 63.
    - b If required, use the **Primitive GUID**, **Package CLSID** and **Runtime CLSID** boxes to specify that the primitive should use other assemblies than the ones automatically generated by the ArcestrA Object Toolkit. For more information, see Associating Different Assemblies with an Object on page 64.
  - 10 If required, configure associated files for the primitive. This works the same as configuring associated files for the main ApplicationObject. For more information, see Configuring Associated Files Manually on page 55.
  - 11 Click **OK**, or go back to Step 2 to add more primitives.
- The Object Design View now shows the new primitive in the tree. It has its own sub-entries for attributes, config time and run time events. You can configure custom code for the primitive's events and set handlers in the same way as you would configure it for the object itself.

## Adding a Reusable Primitive

A reusable primitive is a primitive that is intended to be included into multiple objects. By implementing common features as reusable primitives, you avoid code duplication. You can also add multiple instances of a reusable primitive to the same object.

Using the ArcestrA Object Toolkit, you can create your own custom reusable primitives. For more information, see Chapter 5, Defining a Reusable Primitive.

Standard reusable primitives installed by Wonderware Application Server are available at C:\Program Files\Common Files\ArcestrA\ReusablePrimitives\ArcestrA.

On a 64-bit operating system, standard reusable primitives will be installed at C:\Program Files (x86)\Common Files\ArcestrA\ReusablePrimitives\ArcestrA.

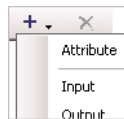
Technically, the Inputs and Outputs as well as the Alarm and History extensions that you can add in the Object Designer are reusable primitives as well. However, there are separate mechanisms in the Object Designer for adding and configuring these primitives. For more information, see:

- Adding Inputs and Outputs on page 89 for information on Inputs and Outputs
- Historizing an Attribute on page 79 for information on the History primitive
- Making an Attribute Alarmable on page 83 for information on the Alarm primitive

### To add a reusable primitive to your object



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the primitive. For example, if you want to add a primitive to another primitive, select that primitive or one of its attributes.



- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Reusable Primitive**. The **Add Existing (Predefined) Primitive** dialog box appears.
- 5 Select the .aaPRI file of the reusable primitive you want to add. Click **Open**.
- 6 The reusable primitive and its attributes appear in the **Shape** pane. The primitive's properties are shown in the **Configuration** pane.
- 7 In the **External Name** box, enter a unique external name for the primitive. This is the name by which other objects can access the primitive's attributes. The name must be ArchestrA compliant. For applicable restrictions, see Naming Conventions on page 157.

You can also leave it blank. For more information, see Naming Considerations for Primitives on page 46.

- 8 In the **Internal Name** box, enter a unique internal name for the primitive. This is the name by which you can refer to the primitive in the object's code. The name must be C# compliant. The maximum length is 329 characters.
- 9 Select the **Virtual** check box if the primitive should be virtual. For more information, see Working with Virtual Primitives on page 43.
- 10 Click **OK**, or go back to Step 2 to add more primitives.

The Object Design View now shows the new primitive in the tree. Unlike with a local primitive, you can't configure custom code for the primitive's events and set handlers because that code is already configured in the reusable primitive itself.

### Overriding and Locking Attributes of Reusable Primitives



When you include a reusable primitive into another primitive or object, you may be able modify the default values and security classification of its attributes. This is called *overriding*.

When appropriate, you can also lock these overridden values, which prevents them from being changed after the object is imported into a Galaxy. A common example is an object that is designed to monitor only Boolean items from the field. To do so, include an Input Primitive, override its "data type" attribute to Boolean, and lock it.

## Deleting a Primitive

You can delete primitives from your object. In that case, you must check whether the object still contains any references to the deleted primitive or its attributes, and change those references accordingly.

#### To delete a primitive

-  1 Open the Object Designer.
- 2 In the **Shape** pane, select the primitive you want to delete.
-  3 Click the **Delete** icon. A confirmation message appears.
- 4 Click **Yes** to delete the primitive.

## Working with Virtual Primitives

You can use a *virtual primitive* to implement a block of optional functionality that can be enabled as required by an end user. This ensures that only required primitives are actually deployed at run time, reducing overhead and processing load.

You design and implement a virtual primitive in the same way as any other primitive. However, from the end user's perspective, a virtual primitive does not appear to be part of the object by default. Instead, it is only made "real" at configuration time by programmatically adding instances of the virtual primitive to the object as needed. For more

information, see the documentation on the `AddPrimitive` and `DeletePrimitive` methods in the *Archestra Object Toolkit Reference Guide*.

For example, you might want to provide an optional Hi alarm for an object's PV (process value). To do this, you would add a virtual primitive that contains the alarming functionality, and an attribute that enables or disables the Hi alarm. When the user enables that attribute at config time, you create an instance of the virtual primitive via a call in the attribute's config time set handler. If the attribute stays disabled, you never create an instance of the primitive, and the primitive never gets deployed.

You can create multiple instances of a virtual primitive in the same object. Each instance behaves like a separate primitive. For example, you could re-use the same Hi alarm primitive for multiple attributes by simply creating multiple instances of it. However, when you add the instances, you must provide a unique external and internal name for each instance to avoid naming conflicts.

For an example of using virtual primitives, see the WatchDog sample object. It uses a virtual primitive for providing optional statistics calculations. See Appendix C, Sample Projects.

You can also use virtual primitives to define "optional" attributes and "arrays" of related primitives. For more information, see the following sections.

### Defining "Optional" Attributes Using Virtual Primitives

Using virtual primitives, you can include primitives (and their attributes) in an object only when they are really necessary. For example, if the monitoring of a particular input is optional, you can mark the input primitive as virtual and include it in the object only when a certain attribute (e. g. "EnableInputMonitoring") is set to true. To do this, you would include logic for creating or removing an instance of the virtual primitive in that attribute's config time set handler.

### "Arrays" of Related Primitives

Using the same technique as described above, you can easily monitor a variable number of inputs (e. g. 0 to 4). The only difference is that you create more than one instance of the virtual primitive. Each primitive instance must have unique internal and external names so its attribute names do not collide with those of the other primitive instances.

## Naming Considerations for Primitives

This section explains how to use a primitive's names and what happens if you change them.

A primitive has two names: its *internal name* and its *external name*. In many cases, these two names will be identical.

- The **internal name** is used to refer to the primitive from config time and run time code. Avoid changing this name after you've used it in code. If you do change it, you must manually update any references where the primitive name is passed as a string.

Keep this name as short as possible. Long names can make source file and folder names excessively long, and may increase memory usage at run time.

- The **external name** determines the names of the primitive's attributes. For example, if you have an attribute named "Condition" in a primitive named "AlarmHi," you can access the attribute as "AlarmHi.Condition" in the object's namespace. If you change this name, references in the config time or run time code are not affected. However, you must update any references where it is passed as a string, e. g. in the object's custom editor.

A primitive's external name can be empty. In this case, the external name does not become part of the namespace that is visible to the end user. In the example above, if you have an attribute named "Condition" in a primitive with an empty external name, you can simply access that attribute as "Condition" in the object's namespace. However, in that case, you must pay extra attention that no naming conflicts occur between the primitive's attributes and any attributes of the containing object or primitive.

## Adding a Custom Object Editor

**Note** This section is about creating the custom object editor for end users to configure your object in the ArcestrA IDE. For information on the ArcestrA Object Toolkit Object Designer that you use to define your object in Visual Studio, see Object Designer Window on page 22.

By creating a custom object editor, you provide a graphical interface for configuring your object's attributes. The custom object editor appears when the user opens the object for configuration in the ArcestrA IDE. It should allow the user to configure all configurable attributes of the object and its primitives.

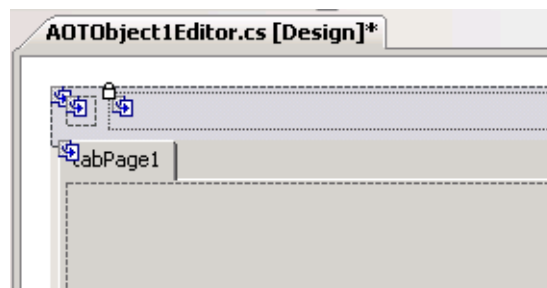
You can create multiple custom tab pages for your object editor. In the ArcestrA IDE, these custom tab pages appear alongside the standard tab pages that are shown for every object (**Object Information**, **Scripts**, **UDAs**, etc.). You can disable these standard tabs if you don't need them. See *Configuring the Object's IDE Behavior* on page 61.

Your custom object editor can use standard controls from the Visual Studio toolbox or special ArcestrA controls. For information on adding the ArcestrA controls to Visual Studio, see *Adding ArcestrA Controls to the Visual Studio Toolbox* on page 48.

Your editor can also include controls for configuring attributes of virtual primitives. In the ArcestrA IDE, these controls are automatically enabled or disabled depending on whether the virtual primitive instance exists or not.

### To create the custom object editor

- 1 In the Object Design View, double-click the **Editor** folder. The custom object editor appears in the Visual Studio Design view. It already contains a tab page. This is the first custom tab page that appears when you edit your object in the ArcestrA IDE.



- 2 Add controls to the tab page. You can use Visual Studio's standard UI design features for this. You can also:
  - Drag attributes from the Object Design View onto the tab page. The ArcestrA Object Toolkit then automatically adds a standard label as well as ArcestrA controls for editing the value, security classification and lock status of that attribute.
  - Drag object dictionary items from the Object Design View onto the tab page. The ArcestrA Object Toolkit then automatically adds a label that will show the correct localized value at run time.
- 3 If required, add more tab pages.
- 4 When you are done, save your work.

## Adding ArcestrA Controls to the Visual Studio Toolbox

You can add special ArcestrA controls to the Visual Studio toolbox so you can use them in your custom object editor.

### To add ArcestrA controls to the Visual Studio toolbox

- 1 Right-click the Visual Studio toolbox and then click **Add Tab**. A new tab appears in the toolbox. Give it a descriptive name, such as "ArcestrA."
- 2 Right-click the new tab and then click **Choose Items**. The **Choose Toolbox Items** window appears with the **.NET Components** tab selected.
- 3 Click **Browse**. Browse to the C:\Program Files\Wonderware\Toolkits\ArcestrA Object\Bin folder and select the ArcestraEditorFramework.dll file.  
On a 64-bit operating system, browse to the C:\Program Files (x86)\Wonderware\Toolkits\ArcestrA Object\Bin folder to select the file.
- 4 Click **Open**. The ArcestrA controls are added to the list of controls.
- 5 In the **Name** column, check those ArcestrA controls that you want to see on your new tab. Click **OK**.

The ArcestrA controls now appear on your new tab in the Visual Studio toolbox.



## Changing the Attribute Reference of ArchestrA Controls

You can change the attribute references of ArchestrA controls after you have added them to your object editor. To do this, set the control's "Attribute" property to the external name of the attribute. You can also do this programmatically from config time code. This allows you, among other things, to configure multiple primitive instances using the same editor page.

For example, to set the attribute reference of the control instance "aaTextBox1" to "Prim1.Attr1", use this statement:

```
aaTextBox1.Attribute = "Prim1.Attr1";
```

## Configuring Associated Files

If your project contains references to custom files or libraries/assemblies, you must associate these "dependent files" with the object definition. This ensures that they are included when you build the object. The associated files become part of the .aaPDF object package file. When you later import the object on the target system, each associated file is automatically registered on that system based on its type.

If you know that the required files will already be present on the target system, you can also tell the ArchestrA Object Toolkit to specifically ignore these files. In that case, the files are not included in the object package. This is handy for references to standard Windows or ArchestrA assemblies.

There are two ways to configure associated files:

- For files that *are* set up as references in Visual Studio (e. g. custom or third-party assemblies), you set up rules using the Dependent File Manager. Rules are regular expressions that can cover multiple references. All files covered by a rule are then automatically added to the object's **Associated Files** list. You must set up rules for all project references that you have set up in Visual Studio before you can build your object.
- For files that *are not* set up as project references in Visual Studio (e. g. custom data files), you set up the association manually in the Object Designer's **Associated Files** list.

The following sections explain both options.

## Setting up Rules for Dependent Files

To associate files with your object that are set up as project references in Visual Studio (for example, custom or third-party assemblies), you set up rules using the Dependent File Manager. Rules are regular expressions that can cover multiple references. All files that are linked to the references covered by a rule are automatically added to the object's **Associated Files** list.

---

**Note** The file and folder names of associated files must not contain any multi-byte characters.

---

You can configure a default set of rules for all projects, and you can configure specific rules for a single project.

- When you open the Dependent File Manager while no ArcestrA Object Toolkit project is opened, you can configure only the default rules for all projects.
- When you open the Dependent File Manager while an ArcestrA Object Toolkit project is opened, you can configure both the default rules for all projects as well as specific rules for the current project.

You can set up rules before or after you have added the relevant references to your project in Visual Studio. In the latter case, the references automatically appear in the Dependent File Manager, and you can create rules for them very easily without having to type the reference again. See the following two sections for each method.

Rules are checked in the order that they appear in the Dependent File Manager. After the ArcestrA Object Toolkit finds a rule that matches a particular reference, it ignores any subsequent rules that might also match that reference.

You can also configure additional search paths for dependent files. See *Configuring Additional Search Paths* on page 113.

---

**Note** The **System** folder of the Dependent File Manager always contains a set of default rules for references to core system libraries. You can't edit the rules in this folder.

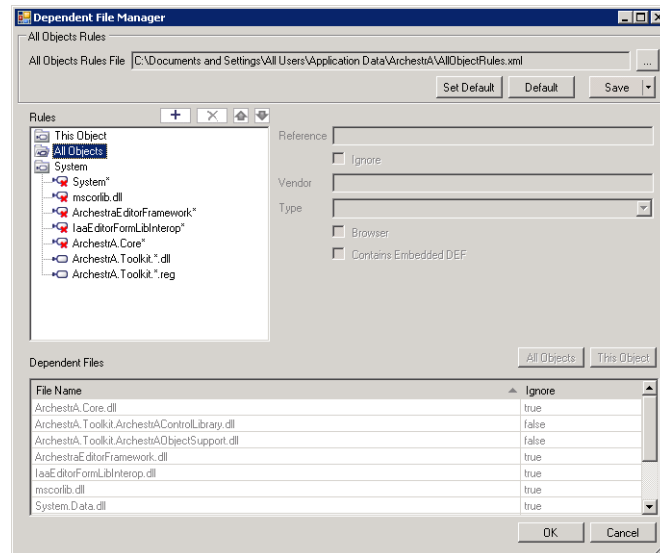
---

## Setting up Rules for References that Don't Currently Exist in Visual Studio

To set up rules for dependent files that you haven't yet added as references in Visual Studio, use the following procedure.

### To set up rules for dependent files manually

- 1 Click the **Dependent File Manager** icon in the Archestra Object Toolkit toolbar. The **Dependent File Manager** dialog box appears.



### 2 Create a rule.

- a In the **Rules** list, click either **This Object** or **All Objects**, depending on where you want the rule to apply.
- b Click the **Add** icon. A new rule appears in the list.
- c In the **Reference** box on the right, enter the reference expression to which this rule should apply. This can be a .NET regular expression. For example, if you are using references to "MyMathLib.Core.dll" and "MyMathLib.Data.dll," you can enter "MyMathLib\*" to cover both.
- d Select the **Ignore** check box if the dependent file(s) covered by this rule should be ignored. In that case, the files are not added to the object package file generated by the build process. Use this option if you know that the files will already be present on the target system.

- e In the **Vendor** box, enter the vendor name for the file(s).
- f In the **Type** list, select the file type. This determines if and how files covered by this rule are registered on the target system. The types work as follows:

Type	Description
Dictionary	An ArchestrA Dictionary (.aaDCT) file.
NETFrameworkAssembly	Strongly named .NET Framework Assembly. The file is installed into the Global Assembly Cache.
ComDLL	COM in-proc server DLL. The file is registered on the target system using regsvr32.
Normal	A normal file. Nothing is done on the target system except install the file.
ComEXE	COM local server EXE. The file is executed on the target system with the "/RegServer" parameter.
NTService	A file that runs as a Windows service.
MergeRegistryScript	A .reg file with registry information. The file is imported into the registry.
MsiMergeModule	A bundled subcomponent of an installer.
NETFrameworkAssemblyNIG	.NET Framework Assembly. The file is <i>not</i> installed into the Global Assembly Cache.
Unknown	Only applicable if you selected the <b>Ignore</b> check box.

- g Leave the **Browser** check box unchanged. This feature is reserved for future use.
- h Select the **Contains Embedded DEF** check box if the file contains the object's aaDEF file as an embedded resource. Typically, you don't need to use this setting because the aaDEF file is managed by the ArchestrA Object Toolkit automatically.

- 3 If the dependent files covered by the rule require dependent files themselves, add each of those files to the rule. For example, if you are using a COM DLL, the reference in the project is actually to the auto-generated interop assembly, but not the COM DLL itself. In that case, you would add the actual COM DLL file as a dependent file to the rule. Do the following for each file:
  - a In the **Rules** list, select the rule.
  - b Click the **Add** icon. A new file item appears in the list.
  - c In the **Reference** box on the right, enter the complete path to the file, or click the browse button to select the file.
  - d Configure the remaining options as described in the previous step.
- 4 Click **OK**, or go back to step 2 to create more rules.



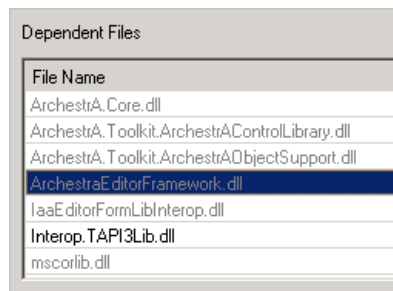
### Setting up Rules for References that Currently Exist in Visual Studio

To set up rules for dependent files that you have already configured as references in Visual Studio, use the following procedure.

#### To set up a rule for a reference already set up in Visual Studio



- 1 Click the **Dependent File Manager** icon in the ArchestrA Object Toolkit toolbar. The **Dependent File Manager** dialog box appears.
- 2 In the **Dependent Files** list, select the reference for which you want to set up a rule. References not currently covered by a rule are highlighted in black type.







- 3 Click either the **All Objects** or **This Object** button, depending on where you want the rule to apply. A new rule for this reference appears in the corresponding section of the **Rules** list.
- 4 Configure the remaining options as described in the previous procedure.

## Deleting and Re-Ordering Rules

You can edit the rules list by deleting rules or changing their order.


### To delete or re-order rules

-  **1** Open the **Dependent File Manager**.
- 2** In the **Rules** list, select the desired rule.
- 3** Do one of the following:
  -  • To delete the rule, click the **Delete** icon.
  -   • To move the rule up or down in the list, click the arrow icons.
- 4** Click **OK**.

## Managing the Rules File for All Projects

The default dependent files rules for all projects are stored in an XML file. You can specify which file to use. This allows you to save and use different sets of default rules.

### To manage the rules file for all projects

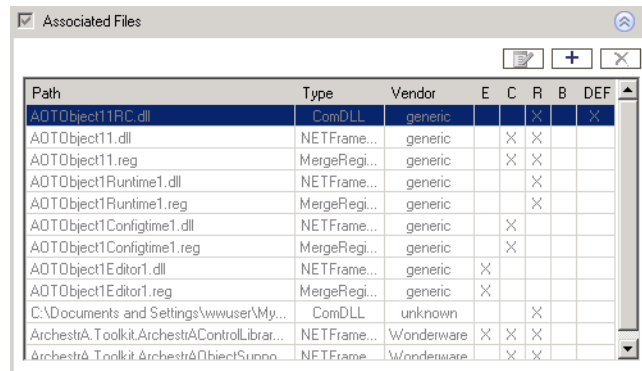
-  **1** Open the **Dependent File Manager**.
- 2** In the **All Objects Rules** area, manage the rules file as follows:
  - To open a different rules file, click the browse button next to the **All Objects Rules File** box.
  - To save the current rules configuration under the file name that is shown, click **Save**.
  - To save the current rules configuration under a different file name, click the down arrow on the **Save** button and then select **Save As**.
  - To set the currently shown file as the default file, click **Set Default**.
  - To use the file that is set as the default file, click **Default**.

## Configuring Associated Files Manually

To associate files with your object that are not set up as project references in Visual Studio (e. g. custom data files), you set up the association manually in the Object Designer's **Associated Files** list. Files listed here are included in the object package file when you build the object, and optionally registered on the target system when you import the object.

### To manually add associated files

- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the topmost node. The object properties appear in the **Configuration** pane.
- 3 Click the downward arrow to the right of the **Associated Files** heading. The section expands.



- 4 Click the **Add** icon. The **Associated File** dialog box appears.
- 5 In the **Filename** box, enter the complete path to the file, or click the browse button to select the file.
- 6 In the **Type** list, select the file type. This determines if and how files covered by this rule are registered on the target system. The types work as follows:

Type	Description
Dictionary	An ArchestrA Dictionary (.aaDCT) file.
NETFrameworkAssembly	Strongly named .NET Framework Assembly. The file is installed into the Global Assembly Cache.
ComDLL	COM in-proc server DLL. The file is registered on the target system using regsvr32.
Normal	A normal file. Nothing is done on the target system except install the file.

Type	Description
ComEXE	COM local server EXE. The file is executed on the target system with the "/RegServer" parameter.
NTService	A file that runs as a Windows service.
MergeRegistryScript	A .reg file with registry information. The file is imported into the registry.
MsiMergeModule	A bundled subcomponent of an installer.
NETFrameworkAssemblyNIG	.NET Framework Assembly. The file is <i>not</i> installed into the Global Assembly Cache.

- 7 In the **Vendor** box, enter the vendor name for the file(s).
- 8 Select the **Needed at Config time**, **Needed at Run time** and **Needed by the Editor** check boxes depending on which components of your object use the file.
- 9 Leave the **Needed by the Browser** check box unchanged. This feature is reserved for future use.
- 10 Select the **Contains Embedded DEF** check box if the file contains the object's aaDEF file as an embedded resource. Typically, you don't need to use this setting because the aaDEF file is managed by the Archestra Object Toolkit automatically.
- 11 Click **OK**.

#### To edit or delete an associated file

**Note** You can only edit or delete files that you manually added to the Associated Files list. To edit or delete a file that was added through a rule, use the Dependent File Manager.

- 1 Open the **Associated Files** list.
- 2 Select the file you want to edit or delete.
- 3 Do one of the following:



- To edit the file, click the **Edit** icon. The **Associated File** dialog box appears. Edit the properties as described in the previous procedure.



- To delete the file, click the **Delete** icon and confirm the deletion.



## Configuring Additional Object Properties

You can configure various additional properties for your `ApplicationObject`:

- You can enable dump/load support for dynamic attributes and virtual primitives as well as failover support for dynamic attributes. See [Configuring Dump/Load Support for Dynamic Attributes and Virtual Primitives](#) on page 58 and [Configuring Failover Support for Run Time Dynamic Attributes](#) on page 59.
- You can enable “Advise Only Active” support for the object to reduce processing and network load when its attributes aren’t subscribed to. See [Enabling “Advise Only Active” Support for the Object](#) on page 60.
- You can set a minimum Application Server version to prevent users from importing your object into earlier versions. See [Configuring the Object’s Minimum Application Server Version](#) on page 60.
- You can set IDE Behavior options to control various aspects of the object’s behavior in the Arcestra IDE. See [Configuring the Object’s IDE Behavior](#) on page 61.
- You can set the toolset that the object should be placed in when it is imported into the Arcestra IDE. See [Setting the Object’s Toolset](#) on page 62.
- You can specify the execution order for the object’s primitives. See [Configuring the Object’s Primitive Execution Order](#) on page 63.

The following sections explain each of these additional properties.

## Configuring Dump/Load Support for Dynamic Attributes and Virtual Primitives

You can enable dump/load support for config time dynamic attributes and virtual primitives in instances or derived templates of your ApplicationObject. These attributes and primitive instances are only added and configured at config time, so their status and configuration may be different in each object instance or derived template. Enabling dump/load support allows you to preserve this configuration when using the Galaxy Dump/Load and Export/Import features on those instances or derived templates.

The dump/load support setting always applies to a specific hierarchy level. For example, when you enable it on the top hierarchy level of your ApplicationObject, it applies to dynamic attributes and virtual primitive instances created on that level. If your object uses child primitives, you must configure the setting separately for each of those primitives.

Using the detailed AddAttribute and AddPrimitive methods, you can exclude specific dynamic attributes or virtual primitives from the dump/load support. For more information, see the *Archestra Object Toolkit Reference Guide*.

---

**Important** When you enable dump/load support, the Archestra Object Toolkit automatically adds an attribute named “InternalDumpLoadData1” to the object. Do not edit or remove this attribute. Otherwise, the dump/load support doesn’t work.

---

### To enable dump/load support



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name.
- 3 In the **Configuration** pane, select the **Dump/Load support for Dynamic Attributes & Virtual Primitives** check box.
- 4 Click **OK**.

## Configuring Failover Support for Run Time Dynamic Attributes

You can enable failover support for run time dynamic attributes. This is relevant when using your `ApplicationObject` in a redundant environment with dual `ApplicationEngines` configured for failover. In such an environment, when the primary `AppEngine` fails, all of its hosted objects become available on the backup `AppEngine`.

When failover support is enabled, any dynamic attributes created on your object during run time are preserved in case of such a failover. Otherwise, run time dynamic attributes are lost when the failover occurs and have to be recreated.

The failover support setting always applies to a specific hierarchy level. For example, when you enable it on the top hierarchy level of your `ApplicationObject`, it applies to dynamic attributes created on that level. If your object uses child primitives, you must configure the setting separately for each of those primitives.

Using the detailed `AddAttribute` method, you can exclude specific dynamic attributes from the failover support. For more information, see the *ArchestrA Object Toolkit Reference Guide*.


You can also use failover support to restore dynamic attributes after a normal object startup, not just after a failover. For additional information and guidelines, see *Ensuring Failover Support for Run Time Dynamic Attributes* on page 170.

---

**Important** When you enable failover support, the ArchestrA Object Toolkit automatically adds an attribute named “\_InternalFailoverData1” to the object. Do not edit or remove this attribute. Otherwise, the failover support doesn’t work.

---

### To enable failover support

-  1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name.
- 3 In the **Configuration** pane, select the **Failover support for Runtime Dynamic Attributes** check box.
- 4 Click **OK**.

## Enabling “Advise Only Active” Support for the Object

You can enable “Advise Only Active” support for your ApplicationObject. When “Advise Only Active” support is enabled, you can configure individual attributes to stop updating if noone is subscribing to them. This reduces the processing and network load.

After enabling “Advise Only Active” support for the object, you still need to configure each individual attribute for “Advise Only Active” support as required. For more information, see [Configuring “Advise Only Active” Support for an Attribute](#) on page 101.

### To enable “Advise Only Active” support for the object



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name.
- 3 In the **Configuration** pane, select the **Advise only Active supported** check box.
- 4 Click **OK**.

## Configuring the Object’s Minimum Application Server Version

You can configure a minimum Application Server version for your ApplicationObject. This prevents users from importing the object into earlier versions of Application Server.

---

**Note** This version check is only performed in Application Server 3.1 and higher.

---

### To configure the minimum Application Server version



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name.
- 3 In the **Configuration** pane, enter the minimum version in the **Min IAS Version** and **Patch** boxes.
- 4 Click **OK**.

## Configuring the Object's IDE Behavior

You can customize the object's behavior in the ArcestrA IDE. For example, you can hide the object from certain views and disable some commands.

### To configure the object's IDE behavior



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name.
- 3 In the **Configuration** pane, select the **IDE Behavior** check box. A list of options appears.
- 4 In the **General** area, select the check boxes for the general options you want to enable:
  - **Hide Tagname:** Hides the object's tagname in the ArcestrA IDE views (Model, Deployment, Derivation). This option is only applicable if the object is contained. Otherwise, the tagname is shown even if this option is enabled. When the object's tagname is hidden, users can only change its contained name in the ArcestrA IDE.
  - **Hide Contained Name:** Hides the object's contained name in the ArcestrA IDE views.
  - **Disable ObjectViewer Menu:** Disables the **View in Object Viewer** menu option in the ArcestrA IDE.
  - **Disable Template Derivation:** Makes it impossible to derive templates from the object.
  - **Disable Instance Creation:** Makes it impossible to derive instances from the object.
  - **Hide Standard Editor Tabs:** Hides the standard tabs (Object Information, Scripts, UDAs, etc.) in the custom object editor. Only your custom tabs are shown.
- 5 In the **Appearance** area, select the check boxes for the appearance options you want to enable:
  - **Hide in Browser:** Hides the object in the Galaxy Browser.
  - **Hide Template in Template Toolbox:** Hides the object template in the **Template Toolbox**.
  - **Hide Instance in Model View:** Hides the object's instances in the **Model** view.
  - **Hide Instance in Deployment View:** Hides the object's instances in the **Deployment** view.
  - **Hide in Security Editor Object List:** Hides the object from the security group configuration (**Security Groups** tab in the **Configure Security** dialog box).
- 6 Click **OK**.

## Setting the Object's Toolset

You can specify which toolset the object is placed in when you import the object into the ArcestraA IDE.

---

**Note** You can configure the list of toolsets available for selection. See [Configuring Toolset Names](#).

---

### To set the object's toolset



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name. The object properties appear in the **Configuration** pane.
- 3 In the **Toolset** list, click the toolset that the object should be placed in after importing. Alternatively, type a custom toolset name.
- 4 Click **OK**.

### Configuring Toolset Names

You can configure the toolsets that you can select for your objects in the Object Designer. Toolset names are saved in an XML file and apply to all ArcestraA Object Toolkit projects on your system. You can also save this file in a central network location and share it across multiple systems.

### To configure toolset names



- 1 In the ArcestraA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2 In the left pane, click **Toolset Names**. The toolset list appears in the right pane. Default toolsets appear with a blue icon. You can't edit these toolsets.
- 3 Configure the toolset names as follows:



- To add a toolset, click the **Add** icon and then click Toolset. In the **Name** box, enter a name for the toolset. The name must be ArcestraA compliant.



- To delete a toolset, select it, click the **Delete** icon, and confirm the deletion.
- To save the toolset names file or to open a different one, use the **Save** and browse buttons.
- To save the current rules configuration under a different file name, click the down arrow on the **Save** button and then select **Save As**.
- To set the currently shown file as the default file, click **Set Default**.
- To use the default file, click **Default**.

- 4 Click **OK**.

## Configuring the Object's Primitive Execution Order

You can configure an execution order for the primitives in an `ApplicationObject` by specifying one of ten execution groups for each primitive. The execution group specifies the order in which the primitive should be executed relative to other primitives in the object. For example, if primitive B depends on data calculated by primitive A, you would want to make sure that primitive A is executed before primitive B so that primitive B gets the latest data in each scan cycle of the AppEngine. To do this, you would place primitive A in an “earlier” execution group than primitive B.

Available execution groups are “Custom 1” to “Custom 10.” Primitives in the “Custom 1” group are executed first, then primitives in the “Custom 2” group, and so on.


Technically, all code that you implement at the `ApplicationObject` level, such as the object's own Startup or Execute event handlers, is contained in a special primitive as well. Therefore, you can also set an execution group at the object level to specify when that code should be executed relative to the code of other primitives in the object.

---

**Note** You can configure the execution group of a reusable primitive when you develop the primitive, but you can't change it after you have embedded the primitive in an `ApplicationObject`.

---

### To configure the object's primitive execution order

-  1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name or a primitive. The object or primitive properties appear in the **Configuration** pane.
- 3 Select the **Advanced Configuration** check box. The **Advanced Configuration** section expands.
- 4 In the **Execution Group** list, select the execution group for the primitive.
- 5 Click **OK**.

## Associating Different Assemblies with an Object

An ApplicationObject consists of a number of different assemblies. For example, there is one assembly for config time code and another for run time code. These assemblies are tied to the main ApplicationObject by means of their CLSIDs.

By default, when you create a new Arcestra Object Toolkit project, the Arcestra Object Toolkit automatically generates a project folder with subfolders and code files for each of these assemblies (see [Creating a Project](#) on page 32). These are the files that you open and edit as you work with the Object Designer and Object Design View.

Initially, the Arcestra Object Toolkit automatically configures the CLSIDs so that your object uses these new default assemblies. However, you can change these auto-configured CLSIDs to specify that your object should *not* use these default assemblies, but different ones.

In most circumstances, if you're creating an object completely from scratch in the current version of the Arcestra Object Toolkit, you would not change the CLSID configuration, but simply keep the default values. But, for example, there might be situations where you would want to use an existing run time assembly that you created by some other means, such as a previous version of the Arcestra Object Toolkit. By specifying that assembly's CLSID, you can tell your ApplicationObject to use that assembly instead of the default assembly that's part of the Arcestra Object Toolkit project.

If you change the CLSID configuration to use custom assemblies, you must manually include these assemblies as associated files so that they are installed and registered on the target system. For more information, see [Configuring Associated Files](#) on page 49.

---

**Note** Every time you increment your object's major version, the CLSID configuration for the object and all child primitives is automatically reset to new, auto-generated values. Therefore, if you are using custom CLSIDs, you must restore them after each major version update.

---



#### To change an object's CLSID configuration



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the object name. The object properties appear in the **Configuration** pane.
- 3 Select the **Advanced Configuration** check box. The **Advanced Configuration** section expands.
- 4 Enter the assembly CLSIDs as follows:
  - a **Primitive GUID**: GUID of the main object assembly. Usually you won't have to change this GUID.
  - b **Package CLSID**: CLSID of the assembly that contains the config time code
  - c **Runtime CLSID**: CLSID of the assembly that contains the run time code
  - d **BRO CLSID**: Reserved for future use. Do not change this setting.
- 5 Click **OK**.

## Adding Object Help

The ArchestrA Object Toolkit does not provide a means to add object help to your object. However, you can use the standard means of the ArchestrA IDE to do this, and then export the object including the object help as an .aaPKG file.

Your object help file must be a standard HTML file.

#### To add object help

- 1 Develop and test your object as usual.
- 2 Open the ArchestrA IDE and import your object's .aaPDF file. The object template is now shown in the **Template Toolbox**.

- 3 Right-click the object template and select **Object Help**. The help window appears with a message that no object help file could be found.



- 4 Save your object help file under the name shown in the error message. For example, if the message says that no help file was found at "C:\Program Files\ArchestraA\Framework\FileRepository\MyGalaxy\Objects\551\Help\1033\help.htm," save the help file in that folder and under that name.
- 5 Close the help window. Repeat step 3 to verify that your help file now appears.
- 6 Export your object as an .aaPKG file.  
The exported .aaPKG file now contains your original object as well as the help file that you manually copied into the help folder. When you import the .aaPKG file into a different galaxy, the help file is automatically imported and saved at the correct location.

## Importing an .aaDEF File from a Previous Object Version

If you have an existing ApplicationObject developed with a previous version of the ArcestrA Object Toolkit, you can easily re-create its shape by importing the object's .aaDEF file. This saves time because you can reuse the existing object shape.

To use this feature, you must use a newly created ArcestrA Object Toolkit project for which you haven't defined any attributes or primitives yet. Otherwise, the .aaDEF import is disabled.

### To import an existing .aaDEF file



- 1 Open the Object Designer.
- 2 Click the **Import aaDEF** button in the bottom. The **Browse for AADef Files** dialog box appears.
- 3 Select the .aaDEF file you want to import and click **Open**. The ArcestrA Object Toolkit imports the .aaDEF file. When the import is finished, the Object Designer shows the object shape as defined in the .aaDEF file.



---

# Chapter 5

## Defining a Reusable Primitive

A reusable primitive is a primitive that is intended to be included into multiple objects. This allows you to share component-level code across objects. The Input and Output primitives that you can add in the Object Designer are good examples of how reusable primitives are beneficial.

Using the ArcestrA Object Toolkit, you can create your own custom reusable primitives and use them in your objects. For more information on how to add a reusable primitive to an object, see [Adding a Reusable Primitive](#) on page 41.

From a development perspective, creating a reusable primitive is very similar to creating an object. Therefore, this section describes only the procedures that are specific to developing a primitive.

---

**Note** Reusable primitives must be added to an object before you can import them into a Galaxy. You can't import a reusable primitive alone.

---

## Switching between Object/Primitive Mode

When working on an ArcestrA Object Toolkit project, you can switch between `ApplicationObject` and reusable primitive mode at any time. For example, when you start working on an `ApplicationObject` but decide that you want to implement its functionality as a reusable primitive instead, you can switch to primitive mode. If you later change your mind and decide that you do want to implement it as an object after all, you can simply switch back to object mode.

Any properties and features that are not relevant to the current mode are blocked from access in the Object Designer and Object Design View. However, they are still preserved in the project code, so when you switch back to the other mode, they are available again.

#### To switch between object and primitive mode



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the topmost node. The object properties appear in the **Configuration** pane.
- 3 In the **Category** list, select either **Application Object** or **Reusable Primitive**, depending on which mode you want.
- 4 Click **OK**.

## Differences Between Editing Objects and Primitives

When developing a reusable primitive, there are certain differences as compared to developing an `ApplicationObject`:

- The Object Designer doesn't have any fields for editing the minor version, toolset name or description. These settings are irrelevant for a reusable primitive.
- The **Configuration Event Handlers** and **IDE Behavior** sections are removed from the Object Designer. Both are irrelevant for a reusable primitive. (The standard configuration event handlers are still available via the Object Design View.)
- A reusable primitive does not have its own setting for “Advise Only Active” support. “Advise Only Active” support is determined by the `ApplicationObject` that the primitive is used in.
- Reusable primitives don't have their own custom editor. Any required configuration UI must be implemented in the editor of the object that contains it.
- The **Build & Import**, **Build & Instantiate** and **Build & Deploy** build modes are not available. They are not applicable to a reusable primitive.
- The **Increment Minor Version** versioning option is not applicable to reusable primitives.
- The output file created by the build process is an `.aaPRI` file, not an `.aaPDF` file. You must add the reusable primitive to an object before you can import it into a Galaxy. You can't import a reusable primitive alone.

# Chapter 6

## Configuring Attributes

Attributes are the data items of an object or primitive. By reading from and writing to attributes, objects can exchange data with each other.

You can configure attributes for an `ApplicationObject` or reusable primitive by using the ArchestrA Object Toolkit's Object Designer. You can:

- Add, edit and delete attributes and array attributes. See [Adding Attributes to an Object or Primitive](#) on page 72 and [Renaming or Deleting Attributes](#) on page 104.
- Configure set handlers for attributes. See [Configuring Config Time Set Handlers](#) on page 74 and [Configuring Run Time Set Handlers](#) on page 76.
- Make attributes historizable and alarmable. See [Configuring Attribute Extensions](#) on page 79.
- Add inputs and outputs to read and write data to and from the field. See [Adding Inputs and Outputs](#) on page 89.
- Configure “Advise Only Active” support for attributes. See [Configuring “Advise Only Active” Support for an Attribute](#) on page 101.

You can also configure attributes directly in the code. For more information, see the *ArchestrA Object Toolkit Reference Guide*.

Attributes that you configure using the ArcestraA Object Toolkit are different from the User-Defined Attributes (UDAs) that you can configure in the ArcestraA IDE. You can only view or edit their configuration in the ArcestraA IDE using your custom object editor, but not the standard UDAs page.

Also, the ArcestraA attributes that we talk about here are not the same as C# attributes. Unless specifically noted, when this manual talks about “attributes,” we mean ArcestraA attributes, not C# attributes.

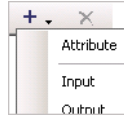
## Adding Attributes to an Object or Primitive

The easiest way to add attributes and array attributes to an `ApplicationObject` or reusable primitive is by using the Object Designer.

### To add an attribute



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the attribute. For example, if you want to add an attribute to a local primitive, select that primitive or one of its attributes.




- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Attribute**. The properties of the new attribute are shown in the **Configuration** pane.
- 5 In the **External Name** box, enter a unique external name for the attribute. This is the name by which other objects can access the attribute. The name must be ArcestraA compliant. See Naming Conventions on page 157 for applicable restrictions.

If you change this name later, you must manually update any references where the attribute name is passed as a string, e. g. in the custom object editor.

- 6 In the **Internal Name** box, enter a unique internal name for the attribute. This is the name by which you can refer to the attribute in the object’s code. The name must be C# compliant. The maximum length is 329 characters.

If you change this name later, you must manually update any references where the attribute name is passed as a string.



- 7 In the **Type** list, select a data type for the attribute. For available options, see Appendix D, ArchestrA Data Types. If the data type isn't known in advance, select **Variant**. You can then set the data type as required using custom config time or run time code.
- 8 In the **Category** list, select a category for the attribute. The category determines who can write to the attribute and whether it is lockable in the template. It also determines whether the additional attribute options in the following steps are available or not. For more information, see Appendix E, ArchestrA Attribute Categories.
- 9 If you want to create custom config time and/or run time set handlers for the attribute, select the **Configuration Set Handler** and/or **Runtime Set Handler** check boxes.
- 10 Select the **Supports Calculated Quality and Calculated Time** check box if the object should be able to set the attribute's quality and timestamp. This may be necessary if you use field data (with potentially Bad or Uncertain quality) to calculate the attribute's value. If you clear this check box, the attribute's quality is always Good, and the timestamp is always the object's startup time.
- 11 Select the **Frequently Accessed** check box to mark the attribute as a "frequently accessed" attribute for the Galaxy Browser. (Users can enable a filter to only display these attributes in the Galaxy Browser.)
- 12 Select the **Advise only Active** check box if you want to implement "Advise Only Active" support for the attribute. For more information, see Configuring "Advise Only Active" Support for an Attribute on page 101.
- 13 In the **Value** box, enter the attribute's default value.
-  14 Click the shield icon next to the **Value** box and select the attribute's security classification. For available options, see Appendix F, ArchestrA Security Classifications.
- 15 To make the attribute an array:
  - a Select the **Array** check box. The array properties appear.
  - b In the **Array Length** box, enter the size of the array.
  - c In the grid, enter default values for each array element.
- 16 Click **OK** to save the attribute, or go back to step 2 and add more attributes.

## Creating a Default Attribute

If you create an attribute with an external name of “PV,” this attribute is considered the object’s “default” attribute. This attribute is used when a reference only specifies the object name without any attribute name. For example, a script can refer to “Tank2Volume.PV” simply as “Tank2Volume”. The attribute “PV” is implied.

## Creating a “Hidden” Attribute

You can “hide” an attribute so that it doesn’t appear in the Galaxy Browser or Object Viewer by default. Other objects can still access the hidden attribute, but regular users won’t see it unless they explicitly choose to display hidden attributes.

Some good reasons to create hidden attributes are:

- To preserve private configuration data in the configuration database
- To preserve private run time data in the checkpoint file
- To allow private data to be transferred from the config time component to the run time component when the object is deployed

### To create a hidden attribute

- ◆ When defining the attribute in the Object Designer, prefix its external name with an underscore. For example, `_MyHiddenAttribute`.

## Configuring Config Time Set Handlers

You can configure a config time set handler for any attribute that can be written to at config time. The code in this set handler is executed every time a value is written to the attribute during configuration. The set handler can then accept the value and write it to the attribute, or reject it. For an example, see [Example: Configuring a Config Time Set Handler](#).

A set handler can also perform other actions, like modifying the values of other attributes or clamping a value.

When rejecting a value, a config time set handler should not generate an alarm, event, or Logger message. Instead, return a localized message to the client. See the example in the section below.

**To configure a config time set handler for an attribute**

- 1 Make sure the **Configuration Set Handler** check box is selected in the attribute's configuration. For more information, see [Adding Attributes to an Object or Primitive](#) on page 72.
- 2 In the Object Design View, expand the **Attributes** folder.
- 3 Expand the attribute name.
- 4 Double-click **Configtime Set Handler**. The code section for the config time set handler appears in the Visual Studio code editor.
- 5 Enter the code for the config time set handler. When you are done, save your changes.

**Example: Configuring a Config Time Set Handler**

Assume you want to return a custom, localized error message to the configuration client if the requested value for an attribute "Attr1" is out of range. First, you set up the error message in the object dictionary. Let's say you give it an ID of 10001 (IDs up to 10000 are reserved for standard messages). Then you code the config time set handler for Attr1 to look something like this:

```
if (<conditions for valid value>)
{
    Attr1 = e.value; // set the new value
}
else
{
    // Reject the value and set the error message if the
    // value is out of range
    e.Message = GetText(10001); // ID of your error
    message
}
```

Now, when an invalid message is written to the attribute at config time, the localized message corresponding to the ID 10001 is shown.

## Configuring Run Time Set Handlers

You can configure a run time set handler for any attribute that can be written to at run time. The code in this set handler is executed whenever anyone other than the primitive logic associated with that attribute attempts to write to it. The set handler can then accept the value and write it to the attribute, or reject it.

A set handler can also perform other actions, e. g. modify the values of other attributes or clamp a value.

When the set handler is called at run time, it is passed not only the new value, but also information about the client making the call (whether the client is a user, another object, or another primitive within the same object). The set handler can take this information into account when deciding whether to accept the new value.

When rejecting a value, run time set handlers should not generate an alarm, event, or Logger message. Instead, return an appropriate error code to the client. See the example in the section below. The `MxStatusDetail` enumeration provides a number of standard error codes that you should use if they fit your object's error situations.

### To configure a run time set handler for an attribute

- 1 Make sure the **Runtime Set Handler** check box is selected in the attribute's configuration. For more information, see *Adding Attributes to an Object or Primitive* on page 72.
- 2 In the Object Design View, expand the **Attributes** folder.
- 3 Expand the attribute name.
- 4 Double-click **Runtime Set Handler**. The code section for the run time set handler appears in the Visual Studio code editor.
- 5 Enter the code for the run time set handler. When you are done, save your changes.

### Example: Configuring a Run Time Set Handler

Assume you want to return a custom, localized error message to the client if the requested value for an attribute “Attr1” is out of range. First, you set up the error message in the object dictionary. Let’s say you give it an ID of 10001 (IDs up to 10000 are reserved for standard messages). Then you code the run time set handler for Attr1 to look something like this:

```
if (<conditions for valid value>)
{
    Attr1 = e.value; // set the new value
}
else
{
    // Reject the value and set the error status if the
    // value is out of range
    e.status.detail = 10001; // ID of your error message
    e.status.Category =
    MxStatusCategory.MxCategoryOperationalError;
    e.status.detectedBy =
    MxStatusSource.MxSourceRespondingAutomationObject;
}
```

When the client receives the error, the `GetStatusDesc` run time event handler is triggered. By default it returns the localized message corresponding to the ID that you put into `e.status.detail`, which is automatically passed to `GetStatusDesc` as `e.DetailedErrorCode`:

```
switch (e.detailedErrorCode)
{
    default:
        e.status = GetText((int)e.detailedErrorCode);
        break;
}
```

## Configuring Dynamic Attribute Set Handlers

The ArcestraA Object Toolkit allows you to dynamically create attributes at config time or run time. For more information, see the documentation on the `AddAttribute` and `DeleteAttribute` methods in the *ArcestraA Object Toolkit Reference Guide*. As with regular attributes, you can create set handler code for these “dynamic” attributes. The ArcestraA Object Toolkit provides a special code section for this.

Technically, there is only one set handler for all dynamic attributes. However, when this set handler is called, the name of the attribute that it’s called for is passed as an argument. By checking this name, you can branch your code and use different set handler code for different dynamic attributes.

### To edit the dynamic attribute set handler code

- 1 In the Object Design View, expand the **Configtime** or **Runtime** folder, depending on which set handler you want to edit.
- 2 Double-click **Dynamic Attributes Set Handler**. The set handler region for dynamic attributes appears in the code editor.
- 3 Enter any required code, and then save your changes.

### Example: Configuring a Set Handler for a Dynamic Attribute

Assume your object has three dynamic attributes at config time: `DynAtt1`, `DynAtt2`, and `DynAtt3`. You want to assure that `DynAtt1` is only set to positive values, whereas any values are valid for the other two attributes. You would configure a set handler like the following:

```
string attrName = Get(e.attributeHandle.shAttributeId,
    e.attributeHandle.shPrimitiveId, EATTRIBUTEPROPERTY.idxAttribPropName); //
    Get name of attribute for which set was made
if (attrName == "DynAtt1") // In this case, reject negative values
{
    if (e.Value < 0)
    {
        e.Message = string.Format("Value for {0} must be positive", attrName);
    }
    else
    {
        SetValue(attrName, e.Value);
    }
    return;
}
SetValue(attrName, e.Value); // In all other cases, just set the value
```

## Configuring Attribute Extensions

You can “extend” an attribute’s functionality in the following ways:

- Historizing the attribute
- Making the attribute alarmable

The following sections explain how to configure these extensions in the Object Designer.

Technically, the extension features are implemented as primitives. The following sections also include reference information on these primitives’ attributes.

### Historizing an Attribute

You can enable history for attributes of the following data types: Float, Double, Integer, Boolean, String, CustomEnum, and ElapsedTime. When you do this, the attribute’s run time values are historized according to the settings of the engine that the object is deployed to. (If historization is disabled for the engine, no attribute values are historized even if history is enabled for the attribute.)

---

**Note** Some attribute categories don’t support historization. For example, attributes that exist only at config time can’t be historized.

---

As a guideline, enable history only for those attributes that most of your users would want to historize. If your users want to historize additional attributes, they can always do so by setting up attribute extensions in the Archedra IDE.

Technically, when you enable history for an attribute, a history primitive is added to the object. You can make this primitive virtual so that your users can choose whether or not they actually need the history functionality. For information on the primitive’s attributes, see Attributes of the History Primitive on page 81.

#### To enable history for an attribute



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the attribute name.
- 3 In the **Configuration** pane, select the **Historizable** check box. The historization options appear. Depending on the attribute’s data type, some options may be disabled.

- 4 Configure the history options for the attribute:
  - a In the **Engineering Units** list, select the attribute that contains the engineering units string for the attribute. (The list shows all string attributes defined in your object.)
  - b In the **Value Deadband** box, enter the value deadband (in engineering units) for historization. If the attribute value changes, the new value is only historized if it exceeds this deadband.
  - c In the **Trend Scale Max** and **Trend Scale Min** boxes, enter the default maximum and minimum scale values for showing the attribute's data in a trend.
  - d In the **Forced Storage Period** box, enter a time interval in milliseconds. The attribute value is always historized at this interval, regardless of whether it has changed. A value of 0 disables this setting, that is, attribute values are only historized if and when they change.
  - e In the **Interpolation Type** list, select the interpolation type to be used.
  - f In the **Rate Deadband** box, enter a deadband rate for swinging door storage (if applicable).
  - g In the **Roll Over Value** box, enter the rollover value (only applicable if Counter retrieval will be used for this attribute's data).
  - h In the **Sample Count** box, enter the number of samples to be stored in the Active Image buffer of the Historian.
  - i In the **Description** box, enter a description to be stored for the attribute on the Historian. This can also be a reference to an attribute that contains the description.
  - j Select the **Enable Swinging Door** check box to enable swinging door storage.
- 5 Select the **Virtual** check box to make the history primitive virtual.

This automatically enables the **Add Attribute to enable/disable History** check box as well. When this check box is enabled, an additional attribute named "<AttributeName>.Historized" is added with config time set handler code to enable and disable the history primitive. You can simply add this attribute to the custom object editor to allow your users to enable or disable history for the attribute. If you don't want this attribute, clear the check box.



- 6 To lock or unlock any history option, click the padlock icon next to its input box. Locked options can't be changed at run time.
- 7 Click **OK**, or go back to step 2 and configure history for additional attributes.

### Attributes of the History Primitive

You can use the following attributes of the History primitive to monitor or re-configure it at config time and run time.

Name	Type	Category	Description
_EngUnitsAttrName	String	PackageOnly_Lockable	Name of the attribute that defines the engineering units string for the value being historized. Only applies to numerical attributes.
_InterPolationTypeEnum	String[3]	Constant	Possible interpolation types: "Stairstep", "Linear", "SystemDefault"
_ValueAttrName	String	PackageOnly_Lockable	Name of the attribute whose values should be historized.
Description	String	Writeable_US C_Lockable	A brief description for the historized attribute. May be a literal string or a reference to another string attribute containing the description. The content is only considered to be a reference if the reference is of the form "me.AttrName". By default, the object's "ShortDesc" attribute is used. (Value can't be set at run time.)
EnableSwingingDoor	Boolean	Writeable_US C_Lockable	Enable or disable swinging door storage. (Value can't be set at run time.)

Name	Type	Category	Description
ForceStoragePeriod	Integer	Writeable_US C_Lockable	The time interval, in milliseconds, at which the value is always stored, regardless of the value deadband setting. Effectively, this allows a continuous storage interval to be superimposed upon the value deadband mechanism. A value less than or equal to 0 disables this feature. As an example, a setting of 360000 indicates the value must be stored once per hour (measured from the time the object was last put OnScan). A value less than the host engine's scan period causes the forced storage to occur every scan cycle.
InterpolationType	Custom Enum	Writeable_US C_Lockable	Interpolation type (Stairstep, Linear, SystemDefault, or None). (Value can't be set at run time.)
RateDeadBand	Float	Writeable_US C_Lockable	Deadband rate for swinging door storage. (Value can't be set at run time.)
RolloverValue	Float	Writeable_US C_Lockable	Rollover value for Counter retrieval. (Value can't be set at run time.)
SampleCount	Integer	Writeable_US C_Lockable	Number of samples to be stored in Active Image. (Value can't be set at run time.)
TrendHi	Float	Writeable_US C_Lockable	The default top of the trend scale for clients. Must be greater than or equal to TrendLo.

Name	Type	Category	Description
TrendLo	Float	Writeable_US C_Lockable	The default bottom of the trend scale for clients. Must be less than or equal to TrendHi.
ValueDeadBand	Float	Writeable_US C_Lockable	The amount, in engineering units, by which the value of the historized attribute must change in order for the new value to be historized. A value of 0 means that all new values are historized. Also, when the attribute's quality changes, the value is always historized regardless of this setting. Only expose this attribute in the configuration environment for numerical datatypes such as Float, Double, or Integer.

## Making an Attribute Alarmable

To configure alarms for an attribute, you create an additional Boolean attribute for each alarm type that you want to enable. Then, you enable the alarm extension and configure alarm options for each of these Boolean attributes. Finally, you create custom code that checks for the alarm conditions and changes the value of the these Boolean attributes accordingly to raise or clear the respective alarms. An alarm is active when the alarmed Boolean attribute is True, and inactive when it is False.

For an example, see Example: Configuring a Value Alarm for an Attribute on page 86.

**Caution** Simply making an attribute alarmable does not ensure that the alarm condition is actually monitored at run time! You must create custom run time code that checks for the alarm condition and raises or clears the alarm as required. The alarm extension only *reports* the alarm to the alarm system, but it does not *detect* it on its own.

As a guideline, configure only those alarms that most of your users would want to enable. If your users want to configure additional alarms, they can always do so by setting up attribute extensions in the Arcestra IDE.

Technically, when you make an attribute alarmable, an alarm primitive is added to the object. You can make this primitive virtual so that your users can choose whether or not they actually need the alarm functionality. For information on the primitive's attributes, see *Attributes of the Alarm Primitive* on page 87.

We recommend that you lock any alarm settings that you don't expect your users to change (such as the alarm category).

#### To make an attribute alarmable



- 1 Open the Object Designer.
- 2 In the **Shape** pane, click the Boolean attribute name.
- 3 In the **Configuration** pane, select the **Alarmable** check box.
- 4 In the **Category** list, select the category to be shown for the alarm. Use the main categories as follows:

Category	Purpose
Value	Limit alarms (LoLo, Lo, Hi, HiHi)
Deviation	Deviation from a setpoint (major, minor)
ROC	Rate-of-change alarms (value changes slower or faster than expected)
Batch	Alarms or events associated with a batch process
Discrete	Discrete alarms
Process	Alarms or events associated with the physical process/plant
SPC	SPC alarms (out-of-spec, out-of-control, "run rules;" etc.)
System	Alarms or events associated with the automation system
Software	Alarms or events associated with a software operation/logic (such as "divide by zero" in a script)

- 5 In the **Priority** box, enter a priority for the alarm (0 = highest, 999 = lowest).
- 6 Optionally, in the **Engineering Units** list, select the attribute that contains the engineering units string for the attribute. (The list shows all string attributes defined in your object.)
- 7 Optionally, in the **Value** list, select the attribute whose value the alarm relates to.
- 8 Optionally, in the **Limit** list, select the attribute that contains the alarm limit value.
- 9 Optionally, in the **Description** list, select the attribute whose value should be used as the alarm description.
- 10 Select the **Virtual** check box to make the alarm primitive virtual.

This automatically enables the **Add Attribute to enable/disable Alarm** check box as well. When this check box is enabled, an additional attribute named “<AttributeName>.Alarmed” is added with config time set handler code to enable and disable the alarm primitive. You can simply add this attribute to the custom object editor to allow your users to enable or disable the alarm. If you don’t want this attribute, clear the check box.

- 11 To lock or unlock any alarm option, click the padlock icon next to its input box. Locked options can’t be changed at run time.
- 12 Click **OK**, or go back to step 2 and configure alarms for additional attributes.

### Example: Configuring a Value Alarm for an Attribute

Assume you want to set up a HiHi value alarm for an Integer attribute named “AlmAtt1.” To do this, you would follow these general steps:

- 1 Create attributes to manage the alarm. You need at least the Boolean attribute that represents the alarm condition. In this example, we will also set up attributes for the limit value and description of the alarm. It’s convenient to group these attributes in a separate primitive. So, add a new local primitive with an empty external name and the following attributes:

Name	Data type	Description
AlmAtt1.HiHi	Boolean	Indicates if the alarm condition is met
AlmAtt1.HiHi.Limit	Integer	HiHi limit value
AlmAtt1.HiHi.AlmDesc	String	Alarm description/comment

- 2 Make AlmAtt1.HiHi alarmable (see Making an Attribute Alarmable on page 83). Set the category to “ValueHiHi” and specify a priority. Set the **Value** attribute to “AlmAtt1,” the **Limit** attribute to “AlmAtt1.HiHi.Limit,” and the **Description** attribute to “AlmAtt1.HiHi.AlmDesc.”
- 3 Add code to the Execute run time event handler of the local primitive you added. The code should:
  - Check the value of AlmAtt1 to see if it exceeds the value of AlmAtt1.HiHi.Limit.
  - If yes, and AlmAtt1.HiHi is False (i. e. the actual alarm condition has just occurred), set AlmAtt1.HiHi to True. If AlmAtt1.HiHi is already True, there is no need to set it again.
  - If no, and AlmAtt1.HiHi is True (i. e. the value has just returned to normal), set AlmAtt1.HiHi to False. If AlmAtt1.HiHi is already False, there is no need to set it again.

Now, when the value of AlmAtt1 exceeds its limit at run time, the primitive code detects this and sets AlmAtt1.HiHi to True. Because you made AlmAtt1.HiHi alarmable, its alarm primitive detects this change in value and reports an alarm using the information that you configured (Category = HiHi, value = current value of AlmAtt1, etc.)

### Attributes of the Alarm Primitive

You can use the following attributes of the Alarm primitive to monitor or re-configure it at config time and run time.

Name	Type	Category	Description
_AlmEngUnitsAttrName	String	PackageOnly_Lockable	Name of the attribute containing the Engineering Units string.
_AlmValueAttrName	String	PackageOnly_Lockable	Name of the attribute whose value is monitored for the alarm condition.
_CategoryEnum	String [14]	Constant	Possible values for the Category attribute: Discrete, Value LoLo, Value Lo, Value Hi, Value HiHi, DeviationMinor, DeviationMajor, ROC Lo, ROC Hi, SPC, Process, System, Batch, Software
_ConditionAttrName	String	PackageOnly_Lockable	Name of the Boolean attribute that represents the alarm condition.
_LimitAttrName	String	PackageOnly_Lockable	Name of the attribute that contains the limit value for the alarm condition.
Acked	Boolean	Calculated	Indicates whether the alarm is acknowledged. This attribute is updated when a user sets the AckMsg attribute. It is always set to false when a new alarm condition is detected (i. e. when the InAlarm attribute changes from false to true).
AckMsg	String	Writeable_US	Operator acknowledgement comment. <b>Run time set handler:</b> Stores received text and sets the Acked attribute to true. Also sets the TimeAlarmAcked attribute to the current time.

Name	Type	Category	Description
AlarmInhibit	Boolean	Writeable_US	When true, the alarm is disabled. This attribute is intended to be written to typically by a script or user or input extension. Only the individual alarm is disabled. No other alarms are disabled in the same object or in any assigned or contained objects.
AlarmMode	Custom Enum	Calculated Retentive	Current alarm mode (based on the commanded mode).
AlarmModeCmd	Custom Enum	Writeable_US	Currently commanded alarm mode.
Category	Custom Enum	Writeable_US C_Lockable	Category of the alarm. The label of each alarm category is fixed. See the <code>_CategoryEnum</code> attribute for possible values. <b>Run time set handler:</b> Ensures that the value is between 1 and 15.
DescAttrName	String	Writeable_US C_Lockable	Description for the alarm condition. May be a literal string or a reference to another string attribute containing the description. The content is only considered to be a reference if the reference is of the form "me.AttrName". By default, the object's "ShortDesc" attribute is used.
InAlarm	Boolean	Calculated	This bit represents the alarm state. This is exactly the same as the attribute in the host primitive that represents the alarm condition except when the alarm state is disabled. In that case, InAlarm is set to false regardless of the actual condition state.



Name	Type	Category	Description
Priority	Integer	Writeable_US C_Lockable	Priority of the alarm. Valid values are 0 to 999. 0 is the highest priority.
TimeAlarmAcked	Time	Calculated	Time stamp indicating the last time the alarm was acknowledged.
TimeAlarmOff	Time	Calculated	Time stamp indicating the last time the alarm went off.
TimeAlarmOn	Time	Calculated	Time stamp indicating the last time the alarm went on.

## Adding Inputs and Outputs

By adding inputs and outputs to your objects or primitives, you can read and write data to and from other ArchestrA objects. For example, your object could have an input that reads data from an attribute of a DIO object, which in turn reads data from an item in a physical PLC.

Technically, inputs and outputs are implemented not as single attributes, but as primitives that expose multiple attributes. However, you can add and configure them much like attributes in the Object Designer. There are three types of input/output primitives:

- **Input:** Reads values from an external reference.
- **Output:** Writes values to an external reference.
- **Input/Output:** Reads and writes values to and from an external reference. Optionally, the input reference can be different from the output reference, i. e. values can be written to a different address than the one they are read from.

An instance of the relevant primitive is added for every input or output that you configure. To read and write the I/O data at run time, you simply use the attributes of each primitive instance.

## Adding an Input

Using an input, you can read single data values from an external input source. Often, that source will be a `DeviceIntegration` object attribute that represents a register or piece of data in a field device, but you can configure any attribute of any `AutomationObject` as the input source. The actual reference is usually configured by the end user.

If you know the expected data type, specify it and lock it in the Object Designer after you add the input. Give the input a useful external name that indicates its purpose to the end user.

When you add a static (non-virtual) input primitive, wrapper classes are added automatically. Use these wrappers to access the input values, quality, and status at run time. For example:

```
if (Input1.Value.Quality ==  
    DataQuality.DataQualityGood)  
{  
    double myValue = Input1.Value;  
}
```

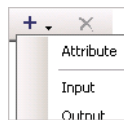
For virtual input primitives, you can use the primitive's attributes instead. For more information, see *Attributes of the Input Primitive* on page 91.

For more information on input primitive wrappers and using the `InputPrimitive` wrapper to create instances of a virtual input primitive, see the *ArchestrA Object Toolkit Reference Guide*.

### To add an input



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the input. For example, if you want to add an input to a local primitive, select that primitive or one of its attributes.
- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Input**. A primitive node for the input is added to the object tree in the **Shape** pane. The properties of the new input are shown in the **Configuration** pane of the Object Designer.
- 5 In the **External Name** box, enter a unique external name for the input. This is the name by which other objects can access the input. The name must be ArchestrA compliant.



6 In the **Internal Name** box, enter a unique internal name for the input. This is the name by which you can refer to the input in the object's code. The name must be C# compliant.

7 Select the **Virtual** check box if the new input primitive should be virtual.



8 In the **Shape** pane, click the **DataType** item underneath the new primitive node. The **Configuration** pane now shows the data type properties.

9 In the **Value** list, select the data type for the input.

10 In the **Shape** pane, click the **InputSource** item underneath the new primitive node. The **Configuration** pane now shows the input source properties.

11 In the **Value** box, enter the input source reference.

12 Click **OK**, or go back to step 2 to add more inputs.

### Attributes of the Input Primitive

You can use the following attributes of the Input primitive to monitor or re-configure it at config time and run time.

Name	Type	Category	Description
DataType	Data Type	Writeable_C _Lockable	Specifies the expected data type of the Value attribute. If you know the data type in advance, you'll probably lock this attribute in the primitive. <b>Config time set handler:</b> Sets the Type property of the Value attribute to the matching type. Only can be done on templates, not instances.
InputSource	Reference	Writeable_U SC_Lockable	Identifies the target attribute from which the value and quality are to be read. <b>Run time set handler:</b> Unregisters the old reference. Registers the new reference, sets Value.Quality to "Initializing," and ReadStatus to "OK" (if the object is off scan) or "Pending" (if on scan).

Name	Type	Category	Description
ReadStatus	Status	Calculated	Indicates the cause of any errors while reading data from the input reference. This is the Message Exchange status, not the status of communication to external devices such as PLCs. The status is updated on every execution. If data is successfully received, but cannot be coerced to the specified data type, ReadStatus is set to “Configuration Error.” ReadStatus is set to “OK” when the object is off scan, and to “Pending” when it goes on scan. When Quality is “Bad,” ReadStatus can be OK or in error.
Value	Variant	Calculated	<p>The value received from the input reference. The Value attribute is “calculated” using data received by a Message Exchange GetAttribute call. Quality can be one of the following:</p> <ul style="list-style-type: none"><li>• “Initializing” when the object goes on scan.</li><li>• “Bad” if data is successfully received, but cannot be coerced to the specified data type.</li><li>• “Bad” when the object goes off scan.</li><li>• The reported quality of the external data in all other cases.</li></ul>

## Adding an Output

Using an output, you can write single data values to an external output destination. Often, that destination will be a DeviceIntegration object attribute that represents a register or piece of data in a field device, but you can configure any attribute of any AutomationObject as the output destination. The actual reference is usually configured by the end user.

If you know the expected data type, specify it and lock it in the Object Designer after you add the output. Give the output a useful external name that indicates its purpose to the end user.

When you add a static (non-virtual) output primitive, wrapper classes are added automatically. Use these wrappers to write output values and monitor the write status at run time. For example, to write an output value:

```
Output1.Value = myValue;
```

Or, to check the write status in a subsequent scan cycle:

```
MxStatus stat = Output1.WriteStatus;
if (stat.Category == MxStatusCategory.MxCategoryOk)
{
    (... any required code ...)
}
```

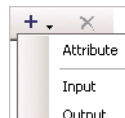
For virtual output primitives, you can use the primitive's attributes instead. For more information, see *Attributes of the Output Primitive* on page 94.

For more information on output primitive wrappers and using the `OutputPrimitive` wrapper to create instances of a virtual output primitive, see the *ArchestrA Object Toolkit Reference Guide*.

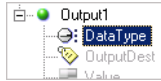
### To add an output



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the output. For example, if you want to add an output to a local primitive, select that primitive or one of its attributes.



- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Output**. A primitive node for the output is added to the object tree in the **Shape** pane. The properties of the new output are shown in the **Configuration** pane of the Object Designer.
- 5 In the **External Name** box, enter a unique external name for the output. This is the name by which other objects can access the output. The name must be ArchestrA compliant.
- 6 In the **Internal Name** box, enter a unique internal name for the output. This is the name by which you can refer to the output in the object's code. The name must be C# compliant.



- 7 Select the **Virtual** check box if the new output primitive should be virtual.
- 8 In the **Shape** pane, click the **DataType** item underneath the new primitive node. The **Configuration** pane now shows the data type properties.
- 9 In the **Value** list, select the data type for the output.
- 10 In the **Shape** pane, click the **OutputDest** item underneath the new primitive node. The **Configuration** pane now shows the output destination properties.
- 11 In the **Value** box, enter the output destination reference.
- 12 Click **OK**, or go back to step 2 to add more outputs.

### Attributes of the Output Primitive

You can use the following attributes of the Output primitive to monitor or re-configure it at config time and run time.

Name	Type	Category	Description
DataType	Data Type	Writeable_C_Lockable	Specifies the expected data type of the Value attribute. If you know the data type in advance, you'll probably lock this attribute in the primitive.
OutputDest	Reference	Writeable_USC_Lockable	Identifies the target attribute to which the value is to be written. <b>Run time set handler:</b> Unregisters the old reference. Registers the new reference and sets WriteStatus to "OK" (if the object is off scan) or "Pending" (if on scan).
Value	Variant	Calculated	The value to be written to the output destination. <b>Run time set handler:</b> Caches the new value and initiates a SupervisorySetAttribute call to the output destination on the object's next execution.

Name	Type	Category	Description
WriteStatus	Status	Writeable_S	Indicates the cause of any errors while writing data to the output destination. This is the Message Exchange status and also, if the output destination is a DeviceIntegration object, the status of communication to the external device (such as a PLC). Updated on each attempt to write a new value only. If data is successfully sent, but cannot be coerced to the specified data type, WriteStatus is set to “Configuration Error.” WriteStatus is set to “OK” when the object is off scan. On a new write attempt, WriteStatus is initially set to the temporary value “Pending” until the write either succeeds or fails. If the output destination is in a DeviceIntegration object, the “Pending” state remains until the DeviceIntegration object returns (asynchronously) the actual completion status to its target, usually an external field device.

## Adding an Input/Output

Using an input/output, you can read and write single data values to and from an external location. You can specify an input source that is different from the output destination. This will be the case when the input is read back from a secondary source location that is different from the output destination. Some field devices may be set up with separate input and output locations for security or robustness purposes.

The actual references are usually configured by the end user. If you know the expected data type, specify it and lock it in the Object Designer after you add the input/output. Give the input/output a useful external name that indicates its purpose to the end user.

When you add a static (non-virtual) input/output primitive, wrapper classes are added automatically. Use these wrappers to read/write I/O values and monitor the I/O status at run time.

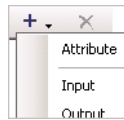
For virtual input/output primitives, you can use the primitive's attributes instead. For more information, see *Attributes of the Input/Output Primitive* on page 97.

For more information on input/output primitive wrappers and using the `InputOutputPrimitive` wrapper to create instances of a virtual input/output primitive, see the *ArchestrA Object Toolkit Reference Guide*.

### To add an input/output



- 1 Open the Object Designer.
- 2 In the **Shape** pane, select the location where you want to add the input/output. For example, if you want to add an input/output to a local primitive, select that primitive or one of its attributes.
- 3 In the **Shape** pane, click the down arrow next to the **Add** icon.
- 4 Click **Input/Output**. A primitive node for the input/output is added to the object tree in the **Shape** pane. The properties of the new input/output are shown in the **Configuration** pane of the Object Designer.
- 5 In the **External Name** box, enter a unique external name for the input/output. This is the name by which other objects can access the input/output. The name must be ArchestrA compliant.
- 6 In the **Internal Name** box, enter a unique internal name for the input/output. This is the name by which you can refer to the input/output in the object's code. The name must be C# compliant.







- 7 Select the **Virtual** check box if the new input/output primitive should be virtual.
- 8 In the **Shape** pane, click the **DataType** item underneath the new primitive node. The **Configuration** pane now shows the data type properties.
- 9 In the **Value** list, select the data type for the input/output.
- 10 In the **Shape** pane, click the **Reference** item underneath the new primitive node. The **Configuration** pane now shows the input/output reference properties.
- 11 In the **Value** box, enter the output destination reference. This reference is also used as the input source unless you configure a separate input source.
- 12 To configure an input source that is different from the output destination:
  - a In the **Shape** pane, click the **SeparateFeedbackConfigured** item underneath the new primitive node. In the **Configuration** pane, select the **true/false** check box.
  - b In the **Shape** pane, click the **ReferenceSecondary** item underneath the new primitive node. In the **Configuration** pane, enter the input source reference in the **Value** box.
- 13 Click **OK**, or go back to step 2 to add more inputs/outputs.

### Attributes of the Input/Output Primitive

You can use the following attributes of the Input/Output primitive to monitor or re-configure it at config time and run time.

Name	Type	Category	Description
DataType	Data Type	Writeable_C_Lockable	Specifies the expected data type of the ReadValue and WriteValue attributes. If you know the data type in advance, you'll probably lock this attribute in the primitive.

Name	Type	Category	Description
ReadStatus	Status	Calculated	<p>Indicates the cause of any errors while reading data from the input reference. This is the Message Exchange status, not the status of communication to external devices such as PLCs. The status is updated on every execution. If data is successfully received, but cannot be coerced to the specified data type, ReadStatus is set to “Configuration Error.” ReadStatus is set to “OK” when the object is off scan, and to “Pending” when it goes on scan. When Quality is “Bad,” ReadStatus can be OK or in error.</p>
ReadValue	Variant	Calculated	<p>The value received from the input reference. The ReadValue attribute is “calculated” using data received by a Message Exchange GetAttribute call. Quality can be one of the following:</p> <ul style="list-style-type: none"><li>• “Initializing” when the object goes on scan.</li><li>• “Bad” if data is successfully received, but cannot be coerced to the specified data type.</li><li>• “Bad” when the object goes off scan.</li><li>• The reported quality of the external data in all other cases.</li></ul>

Name	Type	Category	Description
Reference	Reference	Writeable_US C_Lockable	<p>Identifies the target attribute to which the value is to be written. If no separate input reference is specified, this also serves as the input reference.</p> <p><b>Run time set handler:</b> Unregisters the old reference. Registers the new reference and sets WriteStatus to “OK” (if the object is off scan) or “Pending” (if on scan).</p>
ReferenceSecondary	Reference	Writeable_US C_Lockable	<p>If the SeparateFeedbackConfigured attribute is set to TRUE, ReferenceSecondary identifies the source attribute from which the value and quality are to be read. If left empty, both the input and output use the single location specified in the Reference attribute.</p> <p><b>Run time set handler:</b> Unregisters the old reference. Registers the new reference, sets Value.Quality to “Initializing,” and ReadStatus to “OK” (if the object is off scan) or “Pending” (if on scan).</p> <p><b>Config time set handler:</b> Only allows this attribute to be set if SeparateFeedbackConfigured is TRUE.</p>
SeparateFeedbackConfigured	Boolean	PackageOnly_Lockable	<p>Specifies whether the primitive receives input data from a source address that is different from the output destination.</p>

Name	Type	Category	Description
WriteStatus	Status	Writeable_S	<p>Indicates the cause of any errors while writing data to the output destination. This is the Message Exchange status and also, if the output destination is a DeviceIntegration object, the status of communication to the external device (such as a PLC). Updated on each attempt to write a new value only.</p> <p>If data is successfully sent, but cannot be coerced to the specified data type, WriteStatus is set to “Configuration Error.” WriteStatus is set to “OK” when the object is off scan. On a new write attempt, WriteStatus is initially set to the temporary value “Pending” until the write either succeeds or fails. If the output destination is in a DeviceIntegration object, the “Pending” state remains until the DeviceIntegration object returns (asynchronously) the actual completion status to its target, usually an external field device.</p>
WriteValue	Variant	Calculated	<p>The value to be written to the output destination.</p> <p><b>Run time set handler:</b> Caches the new value and initiates a SupervisorySetAttribute call to the output destination on the object’s next execution.</p>

## Configuring “Advise Only Active” Support for an Attribute

You can implement “Advise Only Active” support for attributes in your objects and primitives. This allows you to configure individual attributes to stop updating if noone is subscribing to them. This reduces the processing and network load.

When you enable “Advise Only Active” support for an attribute, the Application Server infrastructure continually monitors whether there are active subscriptions to that attribute. When there are no (or no more) subscriptions, it calls a special, attribute-specific method to notify the attribute that it should suspend updates. When the first subscription starts, Application Server calls the same method again to notify the attribute that it should resume updates. You can customize this method for each attribute to suspend or resume subscriptions to any data sources that the attribute uses.

Typically, you would only implement “Advise Only Active” support for attributes that are associated with “live” updates from external sources, e. g. an input or an attribute in another object that your object subscribes to via Message Exchange. For example, if noone is polling the value of a calculated attribute that uses an value from an input primitive, you could stop requesting the value so as to reduce the load on the associated I/O server and network.

You can implement “Advise Only Active” support for all attribute types and categories that are available at run time.

Note the following:

- Before you can implement “Advise Only Active” support for individual attributes, you must enable it on the `ApplicationObject` level. For more information, see [Enabling “Advise Only Active” Support for the Object](#) on page 60. To use “Advise Only Active” in a reusable primitive, it must be enabled in the containing object.
- When you enable “Advise Only Active” support for an `ApplicationObject`, all `Input` and `InputOutput` primitives within the object (including its child primitives) are suspended automatically on startup. In most cases, this should fit your needs. However, if you have any attributes that are not configured for “Advise Only Active” and that require data from these `Input` or `InputOutput` primitives at each scan, you must activate these inputs at startup time by calling their `ActivateUpdatesList()` methods. For example,  

```
Input1.ActivateUpdatesList();
```

#### To implement “Advise Only Active” support for an attribute

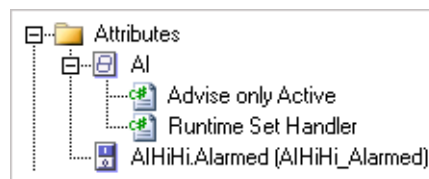
- 1 Enable the “Advise Only Active” option for the attribute:



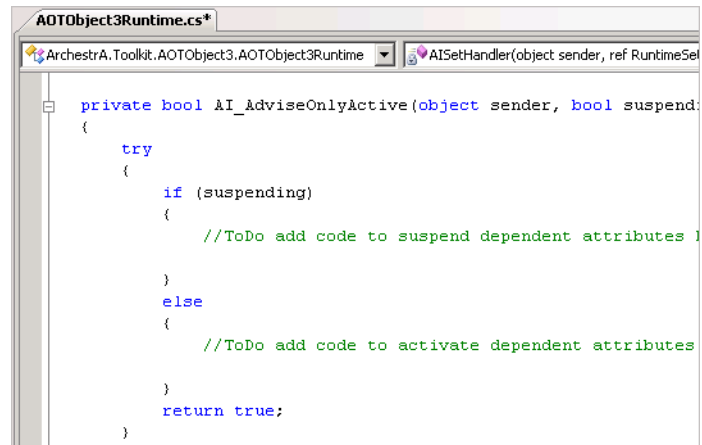
- a Open the Object Designer.
- b In the **Shape** pane, click the attribute name.
- c In the **Configuration** pane, select the **Advise only Active** check box.
- d Click **OK**.

The ArchestraA Object Toolkit automatically adds a method named “*AttributeName\_AdviseOnlyActive()*” to the run time code. This is the method that the Application Server infrastructure calls to notify the attribute that it should suspend or resume updates.

- 2 In the Object Design View, expand the **Attributes** folder. Expand the attribute name.



- 3 Double-click the contained **Advise only Active** node. The “*AttributeName\_AdviseOnlyActive()*” method section of the run time code file appears in the Visual Studio code editor.



- 4 The `if (suspending)` branch is executed when Application Server determines that there are no more subscriptions to the attribute. Enter code here to suspend updates from any data sources that the attribute uses. For example:
  - If the data source is an Input or Input/Output primitive, call its `SuspendUpdatesList()` wrapper method.
  - If the data source is an attribute in another object, use the `CMxIndirect.Suspend()` method.
- 5 The `else` branch is executed when subscriptions to the attribute start again. Enter code here to re-activate updates from any data sources that the attribute uses. For more information on available methods, see the *Archestra Object Toolkit Reference Guide*.
- 6 When you are done, save your work.

## Renaming or Deleting Attributes

You can rename or delete attributes even if other places in your object already refer to them. Note the following:

- When you change the *internal name* of an attribute, references in your object are automatically updated. However, you must manually update any references where the internal name is passed as a string.
- When you change the *external name* of an attribute, you must manually update any references where the external name is passed as a string. This includes any references in the custom object editor or GetValue/SetValue calls.
- When you delete an attribute, references in your object are automatically checked and updated, but you must manually update any references where the internal or external name is passed as a string.



# Chapter 7

## Internationalizing Objects

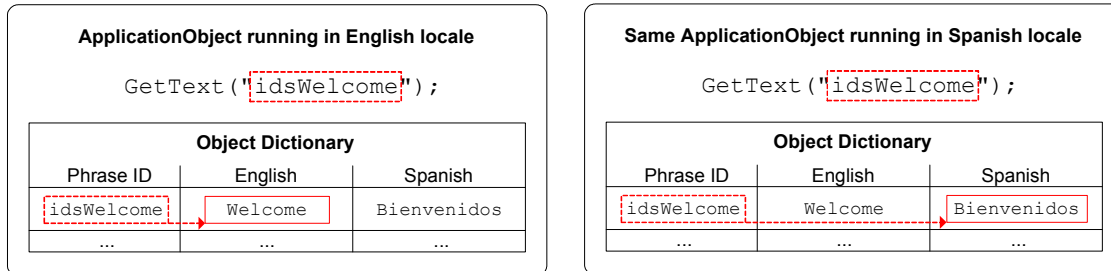
If your object will be used in localized environments, you can internationalize it by defining a multilingual dictionary that contains translated strings for the target locales. At run time, the object can retrieve the appropriate strings for the locale it is used in, and show those translated strings to the end user.

### About Internationalizing Objects

By internationalizing an object, you enable it to run in various language environments and use different translated text messages, prompts, etc. in each case. For example, assume that you develop a complex object with its own custom editor pages. Customers in North America would expect to see English editor pages. On the other hand, customers in South America might use the object on a Spanish operating system and expect to see the editor pages in Spanish.

One way to do this is to create and maintain separate language versions of the object. However, this makes it very hard to maintain and update your code. A better way is to maintain one object version that will use the correct translated strings depending on the locale it is used in.

To do this, you separate the object’s code (which works identically in all locales) from the translatable text phrases (which are different for each locale). The translatable content in the code is replaced with abstract phrase IDs. The object then retrieves the appropriate translated content for whatever phrase ID it needs, when it needs it.



If an ApplicationObject contains local primitives, all local primitives use the object’s main dictionary. Reusable primitives, on the other hand, have their own dictionary.

To internationalize your objects, the ArchestraA Object Toolkit provides:

- An object dictionary that stores translated strings for each resource ID
- A function to retrieve translated strings for a resource ID

For more information, see [Configuring the Object Dictionary and Retrieving Localized Dictionary Strings](#) on page 108.

## Configuring the Object Dictionary

For each translatable resource (“phrase”), the object dictionary defines an ID and translated strings for all locales that the object will be used in. You could visualize it as a table like the following:

Phrase ID	English	Spanish	German
idsWelcome	Welcome	Bienvenidos	Willkommen
idsValve1	Valve 1	Válvula 1	Ventil 1
...	...	...	...

When you request the phrase “idsWelcome” on an English operating system, you get the string “Welcome.” On a Spanish operating system, you get “Bienvenidos” instead, and so on.

The object dictionary is saved as an XML file with the .aaDCT extension. By default, it contains only a sample entry, and you have to add phrases and translations before you can use it.

You can edit the dictionary using any text editor or a special XML editor. For more information, see Dictionary File Format and Structure and Editing the Dictionary in Visual Studio on page 107.

## Dictionary File Format and Structure

The XML dictionary file has the following structure:

```
<Dictionary>
  <Phrase_Index PhraseID="SampleEntry">
    <Language LangID="1033">
      <VALUE>English string</VALUE>
    </Language>
    <Language LangID="1031">
      <VALUE>String translated into German</VALUE>
    </Language>
    ... more <Language> elements for other
    languages ...
  </Phrase_Index>
  <Phrase_Index PhraseID="Entry2">
    ... <Language> elements ...
  </Phrase_Index>
</Dictionary>
```

There is a single `Phrase_Index` element for every translatable string. Its `PhraseID` attribute defines the ID by which you can access the string.

Each `Phrase_Index` element contains one `Language` element for each language-specific translation of the string. The `LangID` attribute of the `Language` element specifies the locale that the translation applies to.

Each `Language` element contains a single `VALUE` element with the translated string for the specified locale.

## Editing the Dictionary in Visual Studio

You can directly edit the object dictionary using the built-in Visual Studio editor.

### To edit the dictionary in Visual Studio

- ◆ In the Object Design View, double-click **Dictionary**. Visual Studio opens a tab with the dictionary file.

You can now add or edit strings according to the dictionary XML structure.

## Retrieving Localized Dictionary Strings

To retrieve a localized dictionary string at config time or run time, simply use the `GetText` method. For example:

```
GetText("idsError");
```

This statement gets the translation for the dictionary entry with the ID “idsError” for the default locale of the process it is called from. For config time code, this is the default locale of the Galaxy, i. e. the OS locale at the time the Galaxy was created.

For more information, see the documentation on the `GetText` method in the *ArchestrA Object Toolkit Reference Guide*.

# Chapter 8

## Building and Versioning Objects

By building your object, you create an .aaPDF object file that you can import and use in Wonderware Application Server. You can:

- **Configure build options.** You can configure various options concerning the build process.
- **Validate your object.** This allows you to find errors that would cause problems when building or using the object but that can't be discovered by Visual Studio's standard checking process.
- **Manage object versions.** You can specify whether to increment the object's major or minor version with a build. You can also override the auto-generated version number in the object properties.
- **Start the build process.** The ArchestrA Object Toolkit can automatically import, instantiate and deploy the object as part of the build process.
- **Analysing migration requirements.** If you are developing a new version of an existing object, the ArchestrA Object Toolkit can help you to structure the code for migrating the existing object.

## Validating an Object

Validating an object allows you to find errors that would cause problems when building or using the object but that can't be discovered by Visual Studio's standard checking process. This is particularly important if you have edited the object's code directly. For example, you might try to assign a value of an invalid type to an attribute. This type of error is invisible to Visual Studio, but can be discovered by validating the object.

Your object is validated automatically when:

- You open the Object Designer.
- You build the object.
- You refresh the Object Design View.

You can also start validation manually.

The validation process reports any warnings and errors in the Logger view and, where appropriate, tries to fix the underlying issues. The warning and error messages are self-explanatory, so they are not duplicated here.

If the validation process detects any errors, you must fix them before you can build or debug the object, use the Migrate analysis, open the Object Designer, or update the Object Design View. These features are disabled until you fix the errors and revalidate the object.

### To start validation manually

- ✓ ◆ In the ArcestrA Object Toolkit toolbar, click the **Validate** icon.

## Configuring Build Options

You can configure build options for all ArcestrA Object Toolkit projects or just the current project.

- When you configure build options while no ArcestrA Object Toolkit project is opened, they apply as defaults for all ArcestrA Object Toolkit projects on that computer. When you move a project to a different computer, it uses the defaults configured on that computer.
- When you configure build options while an ArcestrA Object Toolkit project is opened, they apply to the current project and override the defaults. In this case, you can work with the defaults as follows:

- To restore a certain category of build options to its default values, click the **Default** button on its property page.
- To set the current values as the new default values for a certain category of build options, click the **Set Default** button on its property page.

You can configure the following build options:

- Output preferences to copy the build output to additional locations
- Galaxy preferences to specify the working Galaxy for the various build modes
- Additional search paths for reusable primitives and dependent files

## Configuring Output Preferences

By default, the build output (.aaPDF or .aaPRI file) is saved in the \Output subfolder of your project folder. Optionally, the ArchestrA Object Toolkit can copy the build output to a custom location. When building a reusable primitive, the .aaPRI file can be copied to the common ArchestrA folder for reusable primitives.

### To configure output preferences



- 1 In the ArchestrA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears with the **Build** category selected.
- 2 To copy the build output to a custom folder, select the **Copy output package to specified folder** check box and use the browse button to select the folder.
- 3 To copy reusable primitives to the common ArchestrA folder for reusable primitives after they have been built, select the **Copy reusable primitives to ArchestrA Common** check box. The base folder is always C:\Program Files\Common Files\ArchestrA\ReusablePrimitives. In that folder, the ArchestrA Object Toolkit creates a vendor subfolder based on the vendor name that you configured for the primitive.  
  
On a 64-bit operating system, the base folder is C:\Program Files (x86)\Common Files\ArchestrA\ReusablePrimitives.
- 4 Click **OK**.

## Configuring Galaxy Preferences

When you build an object, the Arcestra Object Toolkit can optionally import, instantiate and deploy the new object version in a Galaxy so that you can test it. For more information, see Building an Object on page 117. You can specify which Galaxy to use for this.

### To configure Galaxy preferences



- 1 In the Arcestra Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2 In the left pane, click **Galaxy**. The Galaxy options appear in the right pane.
- 3 In the **GR Node Name** box, enter the name of the Galaxy Repository node. In most cases, it will be best to use a Galaxy Repository on the local machine. Otherwise, the build process can't automatically restart the Application Server processes to make sure that the latest object version gets used.
- 4 In the **Galaxy** list, enter or select the name of the Galaxy to use.
- 5 If security is enabled for the Galaxy, enter the credentials in the **User Name** and **Password** boxes.
- 6 In the **Assign to Area** box, enter the name of the Area object that instances of your object should be assigned to.
- 7 To test the Galaxy connection, click **Test**.
- 8 Click **OK**.



## Configuring Additional Search Paths

You can configure additional search paths for dependent files. This gives you more flexibility because you can store your development files in multiple locations.

### To configure additional search paths



- 1 In the ArchestrA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2 In the left pane, click **Search Paths**. The **Locations** list appears in the right pane.
- 3 Edit the **Locations** list as follows:
  - To add the first entry, enter the search path in the text box, or click the browse button and select a path. Press Enter to confirm.
  - To add another entry, click the blank entry at the end of the list twice. The entry changes into editing mode. Enter the path as described above.
  - To edit an existing entry, click it twice to change into editing mode, then make your changes as described above.
- 4 Click **OK**.

## Managing an Object's Versions

An ApplicationObject has a version number that consists of a major version and a minor version. For example, “1.3” where 1 is the major version and 3 is the minor version. This version number helps Wonderware Application Server distinguish object versions and detect any migration requirements.

When you build your object, you can keep the current version number, or you can automatically increment the minor or major version. You can also specify the version numbers manually in the object properties.

While developing an object, it is safest to have the major version number increment automatically on new builds. This helps avoid problems if you change the object shape but forget to increment the major version accordingly.

## Building a New Minor Version of an Object

You usually increment an object's minor version after making small changes to the object code. For example, you should increment the minor version after fixing bugs or making optimizations. If you change the object shape in any way, you must increment the major version instead.

When you build a new minor version of an object, you can choose to automatically restart Application Server processes. This is necessary if you have already imported a previous version of the object with the same major version into your Application Server working Galaxy. If you don't restart the processes, Application Server continues to use the previous version even after you import the new minor version.

Depending on which components of your object have changed, you must restart different processes:


- After making changes to **config time** code, you must restart the **aaGR** and **IDE** processes.
- After making changes to **run time** code, you must restart the **Bootstrap** process.
- After making changes to **custom object editor** code, you must restart the **IDE** process.

When you build a new minor version and restart the processes, the Arcestra Object Toolkit performs the following steps in the order listed:

- 1 Undeploys existing instances of the object
- 2 Deletes existing instances of the object
- 3 Deletes the existing object template(s)
- 4 Stops the processes
- 5 Builds the object
- 6 Restarts the processes
- 7 Performs any other steps as defined by the build mode (import, instantiate, deploy)

The Arcestra Object Toolkit can only restart processes running on the local machine. For example, if you are using a remote Galaxy Repository (GR) machine or if you have deployed your object to a remote machine, you must restart the relevant processes manually. To specify the GR node, see [Configuring Galaxy Preferences](#) on page 112.

**To build a new minor version of an object**

-  **1** In the ArcestrA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2** In the left pane, click **Version**.
- 3** Select the **Increment Minor Version** option.
- 4** Select the **Restart Bootstrap**, **Restart aaGR** and **Restart IDE** check boxes as required (see above). If you select **Restart aaGR**, **Restart IDE** is always selected as well.
- 5** Click **OK**.
- 6** Build your object.


## Building a New Major Version of an Object

You increment an object's major version after making extensive changes to the object code, behavior and/or shape. For example, you should increment the major version after adding or renaming attributes. This alerts the user that the new version may not behave the same as previous versions, which might impact the user's application.

While you develop an object, we recommend that you use this setting to have the major version number increment automatically on new builds. This helps avoid problems if you change the object shape but forget to increment the major version accordingly.

When you build an object with a new major version, you can choose to automatically delete the old version's templates and instances from your working Galaxy. Alternatively, you can have the new version imported with the version number appended to the template name. This allows you to keep multiple versions of the same template in the Galaxy without having to manually rename existing templates.

**To build a new major version of an object**

-  **1** In the ArcestrA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2** In the left pane, click **Version**.
- 3** Select the **Increment Major Version** option.

- 4 Specify what to do with existing templates and instances if the Arcestra Object Toolkit automatically imports the new object version into the Galaxy.
  - To keep the old template version and import the new version with the version number appended to its name, select the **Append version number to template name** check box.
  - To delete all previous versions of the object template (as determined by its vendor and object name), select the **Delete all templates with the same vendor and object name** check box. Any instances of these templates are deleted too.
  - To delete all previous versions of the object template and import the new version with the version number appended to its name, select the **Delete all templates and append version number** check box.
- 5 Click **OK**.
- 6 Build your object.

## Creating a New Build without Incrementing the Version Number

You can create a new build without incrementing the object's current version number. For example, you would do this if:

- You are using an automated build system that only recompiles the project.
- You set the final version number manually before release and don't want the final build to increment that number.

When you build an object using this option, the only available build modes are **Build** and **Build & Swap**.

### To build an object without incrementing the version number




- 1 In the Arcestra Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears.
- 2 In the left pane, click **Version**.
- 3 Select the **Retain current version number** option.
- 4 Click **OK**.
- 5 Build your object.

## Manually Specifying the Version Number

If you want to reset the auto-generated version number, you can manually specify the object's version number.

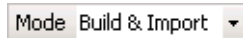
### To manually specify the version number

- 1  Open the Object Designer.
- 2 In the **Shape** pane, click the object name. The object properties appear in the **Configuration** pane on the right.
- 3 In the **Major Version** and **Minor Version** boxes, enter the object's major and minor version. If you increment the major version, you should reset the minor version to 1.
- 4 Click **OK**.

## Building an Object

After you have set all build and versioning options and validated your object, you can build it. This creates an .aaPDF object file that you can import and use in Wonderware Application Server.

The ArcestrA Object Toolkit can automatically import, instantiate and deploy the object as part of the build process.



To specify this, use the **Mode** list in the ArcestrA Object Toolkit toolbar.

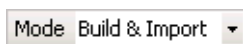
The following build modes are available:

Build Mode	Description
<b>Build</b>	Creates an .aaPDF file of the object in the <b>\Output</b> subfolder of the Visual Studio solution folder.
<b>Build &amp; Import</b>	Creates an .aaPDF file as outlined above, and then imports the template into the working Galaxy.
<b>Build &amp; Inst</b>	Creates an .aaPDF file as outlined above, imports the template into the working Galaxy, derives a new template from it, and creates an instance from that template. The instance is assigned to the area configured in the Galaxy preferences (if any).

Build Mode	Description
<b>Build &amp; Deploy</b>	Creates an .aaPDF file as outlined above, imports the template into the working Galaxy, derives a new template from it, creates an instance from that template, and deploys it. The instance is assigned to the area configured in the Galaxy preferences (if any).
<b>Build &amp; Swap</b>	<p>Allows you to quickly swap the existing object assemblies in the Windows Global Assembly Cache (GAC). This is handy for debugging as it saves you the time for undeploying and redeploying an existing object instance. Don't use this option if you have changed the object shape. Otherwise, unexpected results may occur.</p> <p>When you use this option, you must specify which processes to restart:</p> <ul style="list-style-type: none"> <li>• IDE if you have made changes to the custom object editor</li> <li>• IDE and aaGR if you have made changes to the config time code</li> <li>• Bootstrap if you have made changes to the run time code</li> </ul>

To specify the working Galaxy and Area object for importing, instantiating and deploying the object, see [Configuring Galaxy Preferences](#) on page 112.

### To build an object



- 1 In the Archestra Object Toolkit toolbar, click the desired build mode in the **Mode** list. See above for available options.



- 2 Click the **Build** icon.

The Archestra Object Toolkit now starts the build and performs any other actions specified by the build mode. Any errors or warnings are reported in the Logger pane. The build output (.aaPDF or .aaPRI file, aaDEF file) is stored in the Output subfolder of your project folder.

## Migrating Objects

When you import a new major version of an object template into Wonderware Application Server, existing instances of that object can be automatically migrated. This allows you to preserve their configuration in the new object version.

This is very easy if the new object version uses the same attributes as the previous version. In this case, all attribute values are automatically copied over from the old version's instances to the new version's instances.

However, you must create custom migration code if:

- The new version has attributes that the old version doesn't have, and vice versa; and/or
- The security classification of an attribute has changed in the new version.

In this scenario, the custom migration code handles the mapping of attribute values between the old and new version. For example, if you have changed an attribute's name from "Attribute1" to "AttributeA," but the attribute still has the same purpose, the migration code could copy the value of Attribute1 (in the old version) to AttributeA (in the new version).

The ArcestrA Object Toolkit can help you create the migration code by generating a list of which attributes were added, removed, or have changed. You simply select a previous object version, and the ArcestrA Object Toolkit inserts a code region containing the names of all attributes that have changed between the previous version and the current version you're developing. You can then add migration code for each attribute. For a short example, see [Example: Migrating a Previous Object Version](#) on page 121.

### To create a migration code section for a previous version.



- 1 In the ArcestrA Object Toolkit toolbar, click the **Migrate** icon. The **Browse for aaPDF and aaPRI Files** dialog box appears.
- 2 Select the .aaPDF file of the previous object version (or the .aaPRI file if you are developing a reusable primitive), and then click **Open**. You can now view the auto-generated migration analysis results.

- 3 In the Object Design View, expand the **Configtime** folder, and then double-click the **Migrate** item. The migration results section opens in the code editor.

```

#region Migrate Information version 1 to 2
/*
*** Matched Attributes ***
Name          Type      Security  RTSetHandler  CTS
-----
AIHiHi_Alarmed Boolean  FreeAccess  False         Fal

*** Unmatched Attributes ***
Name          Change      Was (V1)  Now (V2)
-----
AI            RTSetHandler  False     True
Attribute1    added         -----

*** Unmatched Primitives ***

```

Note that the Arcestra Object Toolkit has inserted a new code region showing the differences between the two versions. Enter any required migration code for the previous version here. For more information on available methods, see the *Arcestra Object Toolkit Reference Guide*.

The Arcestra Object Toolkit has also automatically updated the `ObjectAttributes.Migrate` property to include the version number of the object that you selected in step 2. This tells Application Server that your `ApplicationObject` supports migrating from that version. For more information on this property, see the *Arcestra Object Toolkit Reference Guide*.

You can repeat this process for multiple previous versions of an object. The Arcestra Object Toolkit generates a separate code region for each previous version. This allows you to have migration code for multiple previous versions in the same object.



## Example: Migrating a Previous Object Version

See the example code region of the Migrate config time event handler for a short example of migration code. In this example, major version 1 of the object had an attribute named “Eg\_001,” which was renamed to “Example\_001” in the current version. The migration code transfers the value and settings of the old attribute to the new attribute:

```
if (migrate.MajorVersion() == 1)
{
    //Transfer attribute value, lock and security
    classification
    Example_001 = migrate.GetValue("Eg_001");
    //Gets value
    Example_001.Locked = migrate.GetLocked("Eg_001");
    //Gets lock status
    Example_001.Security =
    migrate.GetSecurityClassification("Eg_001");
    //Gets Security Classification

    //Transfer primitive values
    SetValue("Example_001.TrendHi",
    migrate.GetValue("Eg_001.TrendHi"));
    Set("Example_001.TrendHi",
    EATTRIBUTEPROPERTY.idxAttribPropLocked,
    migrate.GetLocked("Eg_001.TrendHi"));

    //Automatically migrate all child primitives
    migrate.AutoMigrateChildPrimitives = true;
}
```

Note the “if” condition at the beginning of the code. Using similar conditions, you can have additional, separate migration code sections for other previous major versions of the object.

## Additional Guidelines for Migrating Objects

Note the following when developing migration code for your objects:

- Make sure that your migration code is aware of the presence or absence of child virtual primitives.
- When migrating attributes within a reusable primitive, the migration code must use the full primitive name to access the attribute in the original object being migrated from. You can use the `OriginalPrimitiveFullName` property in the `MigrateHandler` class for this purpose. For example:

```
migrate.GetValue(migrate.OriginalPrimitiveFullName
+ ".Attribute1")
```



# Chapter 9

## Debugging Objects

The ArcestrA Object Toolkit allows you to attach the Visual Studio debugger to the Application Server processes running your object's code. This allows you to troubleshoot and debug your objects. In order to use the debugging features, you must use the object on a computer that has Visual Studio installed.

---

**Caution** Never debug objects on a production system. Always use a development node with a local Galaxy for debugging.

---

---

**Caution** Never ship an object that was created from a debug build. These objects may require debug modules and may not function correctly in a production environment.

---

There are two ways for attaching the debugger:

- If you have already created a build with the required PDB files and instantiated or deployed the object on the local system, you can attach the debugger to the Application Server processes running the current object version and debug that version.
- If you've made changes to your object and want to debug the new version, you can attach the debugger as part of the build process and then debug the new version.

## Attaching the Debugger to the Processes Running the Current Object Version

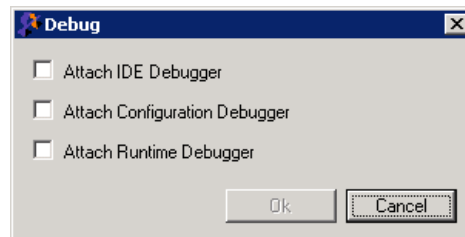
If you want to debug the current version of an object that you have already instantiated or deployed on the local system, you can attach the debugger to the Application Server processes running the current object version. There are two prerequisites for this:

- You must have created the required PDB (Program Database) files as part of the current build.
- Visual Studio must be able to find the PDB files. If necessary, set up the search paths in Visual Studio accordingly.

For help on these points, refer to the Visual Studio documentation.

### To attach the debugger to the processes running the current object version

- 1 In the Arcestra Object Toolkit toolbar, click the **Debug** icon. The **Debug** dialog box appears.






- 2 Select one or more of the following check boxes:
  - **Attach IDE Debugger** to debug custom editor code
  - **Attach Configuration Debugger** to debug config time code
  - **Attach Runtime Debugger** to debug run time code
- 3 Click **OK**. Visual Studio switches into debugging mode. You can now work with your object and use the debugging features as required.
- 4 To stop debugging, click the **Debug** icon in the Arcestra Object Toolkit toolbar again.

## Attaching the Debugger during the Build Process

If you've made changes to your object and want to debug the new version, you can attach the debugger as part of the build process and then debug the new version.

### To attach the debugger during the build process

-  1 In the ArcestrA Object Toolkit toolbar, click the **Options** icon. The **Options** dialog box appears with the **Build** category selected.
- 2 Select one or more of the following check boxes:
  - **Attach IDE Debugger** to debug custom editor code
  - **Attach Configuration Debugger** to debug config time code
  - **Attach Runtime Debugger** to debug run time code
- 3 Click **OK**.
- 4 In the ArcestrA Object Toolkit toolbar, select a build mode in the **Mode** list. For example, to debug config time or custom editor code, **Build & Inst** is convenient.
-  5 Click the **Build** icon to launch the build. Once the build is finished, Visual Studio switches into debugging mode. You can now work with your object and use the debugging features as required.
-  6 To stop debugging, click the highlighted **Debug** icon in the ArcestrA Object Toolkit toolbar.



# Appendix A

## Programming Techniques

Use the following workflow and programming techniques to code within the ArcestrA Object Toolkit (AOT).

### Programming Workflow

Using the AOT, you can seamlessly modify the object shape as the object is coded. The AOT also supports changing the ArcestrA attribute data type and renaming ArcestrA attributes and child primitives. This functionality, referred to as round-tripping, has been implemented by configuring the object shape in code. The code is parsed at build time to form the object's aaDEF file that defines its shape.

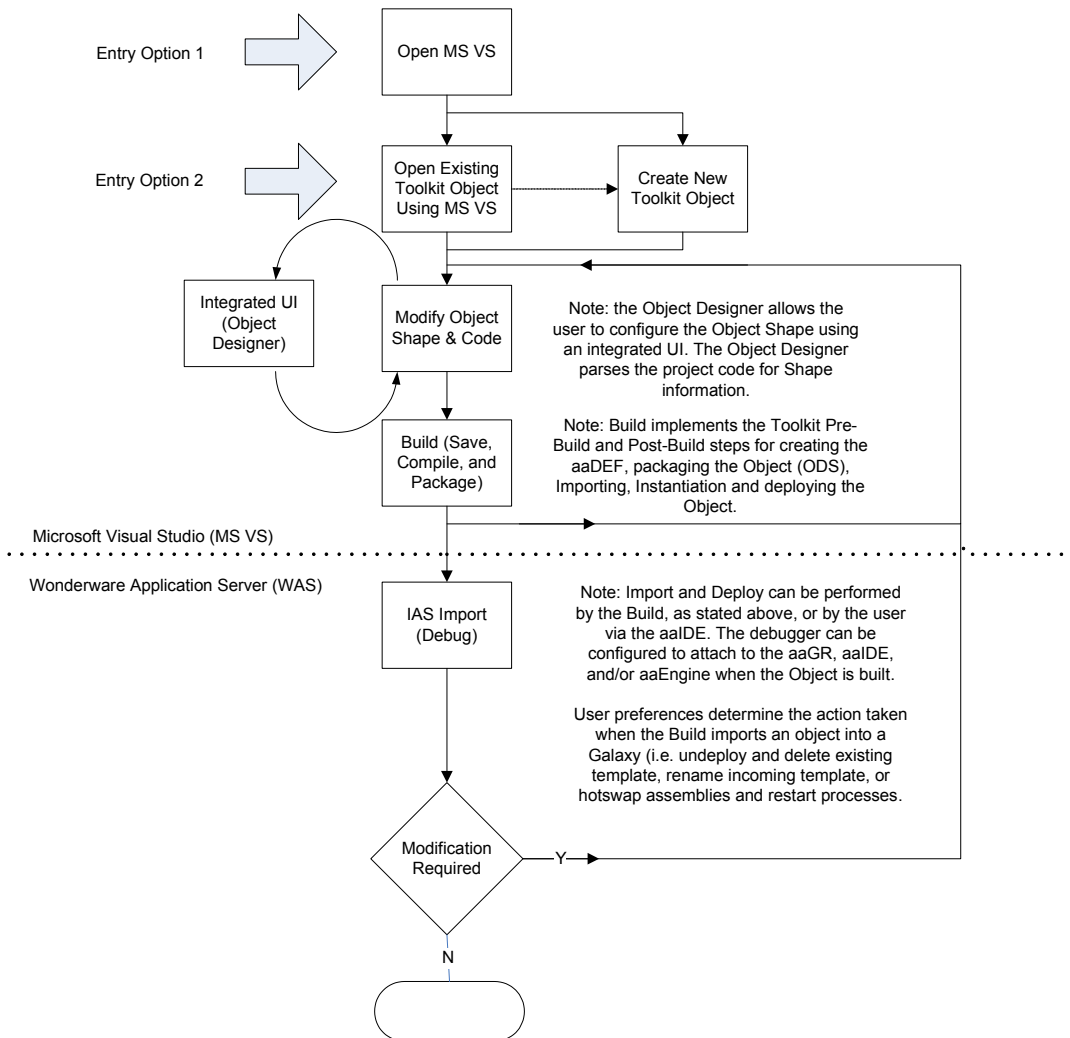
Use the Object Designer to modify the the object shape code. The Designer parses the object shape from the code. Modifications made in the Object Designer are then written back to the code. The Object Designer also enables you to perform tasks that impact multiple code sections simultaneously, such as renaming ArcestrA attributes, adding local primitives, and making changes associated with modifying an object's major version.

The basic steps of the workflow are:

- 1 Define the object shape using either the ArchestrA Object Toolkit Designer or in code directly.
- 2 Code the Configtime.
- 3 Code the Editor for the object.
- 4 Code the Runtime.
- 5 Build and import the object.
- 6 Test and debug the object.

**Note** We recommend that you define the object shape with the ArchestrA Object Toolkit Designer and use code only when necessary. Refer to Advanced Techniques on page 151 for information on coding the shape of the object.

The following is an overview of the AOT workflow from Object creation through to debugging.





## Configuring Internal and External Names

Internal names apply to objects, primitives, and attributes. Internal names are used in code and must be C# compliant.

Do not use .NET keywords, names of classes in the Arcestra Object Toolkit, or names used in other libraries that are used in your code as internal names for objects, primitives, and attributes.

Internal names are used in code to refer to the primitive or attribute in C# through provided wrapper classes.

The maximum length of an object's internal name is 255 characters.

The external name is used in the Arcestra IDE and can be used to create default names for object instances. Each name should be meaningful and suggest the meaning of the object, primitive, or attribute.

External names must be Arcestra compliant and cannot have any of the following characters: (space) + - \* / \ = ( ) ` ~ ! % ^ & @ [ ] { } | : ; " , < > ? "

---

**Note** For the sake of brevity, do not use the word "Object" or "Template" in an object's name.

---

## Providing Wrappers for Referencing Archestra Attributes

Attribute wrappers provide the following functionality for non-dynamic local attributes and non-virtual child primitive attributes:

- Strongly typed attribute references
- Automatic renaming of attribute references
- IntelliSense

---

**Note** Renaming an attribute property causes Visual Studio to rename all references to the attribute in code except for string based references. For example, renaming `Attribute1` does not modify `GetValue("Attribute1")`.

---

Modifying an attribute internal name using the Object Designer renames the Attribute property (known also as the attribute wrapper).

Using the Object Designer to modify the child primitive internal name, that is, the fully scoped name of the referenced attribute, also renames the Attribute property.

---

**Note** Wrappers are updated each time the solution is parsed, that is, when the Object Designer is opened, the object is built, the Object Design View is refreshed, or Code Validate is selected.

---

## Config Time Coding

Use the following techniques to code for the Configtime project.

The Configtime project is used to provide all logic related to configuring the attributes of the object within the Galaxy database.

### Config Time Set Handler

Use the config time set handler to implement any logic required for setting the value of an attribute at configuration time.

This logic could include, for example, range checking and adding or removing virtual primitives.

You can add a Configtime set handler to an attribute by adding a set handler delegate and associated method to the Configtime class.

The Configtime class template provides commented out examples of a set handler delegate and method.

The set handler registration and associated methods are not renamed when the attribute is renamed in code. To remove the set handler, comment out or delete the registration. The Set Handler method can exist without the set handler registration; however, the method is not called.

In the following examples Attribute1 represents the internal and external name of the attribute.

The Configtime set handler is triggered when an attribute is changed at configuration time. It can be used for validation or to trigger a special action such as adding virtual primitives.

For additional information, see [Configuring Config Time Set Handlers](#) on page 74.

## Set Handler Code

`Attribute1.SetHandlerValue` enables the set handler code to appear the same for both array and non-array attributes. You can take more control of setting the value by using the following code examples:

- Non-array attribute:

```
Attribute1 = e.Value;
```

- Array attributes:

```
if (!e.IsArrayElement)
{
    Attribute1 = e.Value;
}
else
{
    Attribute1[e.attributeHandle.shIndex1] =
e.Value;
}
```

## Performing Config Time Validation with the `ConfigtimeValidate()` Method

Use this method to validate the entire object as a whole when it is being saved. The method can put the object in a warning or error state by using `EPACKAGESTATUS` enum.

Example:

```
private void
AOTObject4Configtime_ConfigtimeValidate(object
sender, ref EPACKAGESTATUS status)
{
    // By default set the object status to Good
    if (HiLimit < LoLimit)
    {
        status = EPACKAGESTATUS.ePackageBad;
        AddErrorMessage("Hi Limit must be greater
than or equal to Lo Limit");
    }
    else
    {
        status = EPACKAGESTATUS.ePackageGood;
    }
}
```

## Adding a Virtual Primitive at Config Time with AddPrimitive

You can test the ability to add an instance of a virtual primitive at config time using the following function (Boolean result):

```
bool CanAddPrimitive(string virtualPrimitiveName,
    string internalName, string externalName);
```

The following example uses CanAddPrimitive to check before adding a primitive:

```
if (CanAddPrimitive("c1", "MyCP1InternalName",
    "MyCP1ExternalName"))
{
    AddPrimitive("c1", "MyCP1InternalName",
        "MyCP1ExternalName");
}
```

Where:

- c1 is the internal name of the virtual primitive.
- MyCP1InternalName is the internal name of the Primitive Instance.
- MyCP1ExternalName is the external name of the primitive instance.

You can add an instance of a virtual primitive at config time and check the result using PrimitiveResult.message as shown in the following example:

```
private void Attribute1SetHandler(object sender, ref
    ConfigtimeSetHandlerEventArgs e)
{
    Attribute1.SetHandlerValue = e;

    if (e.Value)
    {
        if (!AddPrimitive("c1", "MyCP1InternalName",
            "MyCP1ExternalName"))
        {
            e.Message = PrimitiveResult.message;
            return; // Add failed
        }
    }
}
```

Where:

- `c1` is the internal name of the virtual primitive
- `MyCP1InternalName` is the internal name of the Primitive instance
- `MyCP1ExternalName` is the external name of the Primitive instance
- `Attribute1` is a Boolean Attribute with a config time set handler.

---

**Note** `PrimitiveResult.message` returns a message only on failure. To return status, use `PrimitiveResult.status` (`EPRIMITIVEOPSTATUS`).

---

Use the following example to iterate through the child primitives or child virtual primitives:

```
object primitives;
this.Site.ChildVirtualPrimitives(ThisPrimitive, out
    primitives);
foreach (IPrimitiveShape ips in
    (IEnumerable)primitives)
{
    LogInfo(ips.FullName);
}
```

## Removing a Virtual Primitive at Config Time with DeletePrimitive

You can test the ability to delete an instance of a virtual primitive at config time using the following function (Boolean result):

```
bool CanDeletePrimitive(string internalName);
```

---

**Note** `CanDelete` checks the lock status of the primitive being deleted.

---

Example:

```
if (CanDeletePrimitive("MyCP1InternalName"))
{
    DeletePrimitive("MyCP1InternalName");
}
```

Where:

MyCP1InternalName is the primitive instance being deleted.

You can delete an instance of a virtual primitive at config time and check the result using `PrimitiveResult.message` as shown in the following example:

```
private void Attribute1SetHandler(object sender, ref
    ConfigtimeSetHandlerEventArgs e)
{
    Attribute1.SetHandlerValue = e;

    if (!e.Value)
    {
        if (!DeletePrimitive("MyCP1InternalName"))
        {
            e.Message = PrimitiveResult.message;
            return; // Delete failed
        }
    }
}
```

Where:

- `c1` is the internal name of the virtual primitive.
- `MyCP1InternalName` is the internal name of the Primitive instance.
- `MyCP1ExternalName` is the external name of the primitive instance.
- `Attribute1` is a Boolean attribute with a config time set handler.

---

**Note** `PrimitiveResult.message` only returns a message on failure. To return status use `PrimitiveResult.status` (`EPRIMITIVEOPSTATUS`).

---

## Accessing Data in Attributes at Config Time

For static attributes, use attribute wrappers based on internal name to read/write value.

For dynamic attributes or attributes in a virtual primitive, use the `GetValue` and `SetValue` methods.

### Examples:

If the static float attribute is named `HiLimit`:

```
float myVal;  
myVal = HiLimit;
```

If the dynamic float attribute is named `HiLimit`:

```
float myVal;  
myVal = GetValue("HiLimit");
```

## Accessing Data in Other Primitives at Config Time

For static primitives, use primitives and attributes wrappers based on internal name to read/write value.

For attributes in a virtual primitive, use the `GetValue` and `SetValue` methods.

### Examples:

To access a float attribute with an internal name of `HiLimit`, of a static child primitive with the internal name `Limits`:

```
float myVal;  
myVal = Limits.HiLimit;
```

To access a float attribute with an external name of `HiLimit`, of an instance of a virtual child primitive with the external name `Limits`:

```
float myVal;  
myVal = GetValue("Limits.HiLimit");
```

## Adding and Deleting Dynamic Attributes at Config Time

Use the following code to add/remove a dynamic attribute.

```
bool status = AddAttribute("dynAttr1",  
    MxAttributeCategory.MxCategoryCalculated,  
    MxDataType.MxDouble, false);  
DeleteAttribute("dynAttr1");
```

For more information on these methods, see the *Archestra Object Toolkit Reference Guide*.



## Run Time Coding

Use the following techniques to code for run time. For more information on these methods, see the *ArchestraA Object Toolkit Reference Guide*.

### Runtime SetHandler

Use a run time set handler to implement any logic required to set an attribute value at run time, including range checking and accepting or rejecting the set.

You can add a run time set handler to an attribute by adding a set handler delegate and associated method to the Runtime class.

The Runtime class template provides commented out examples of a set handler delegate and method.

The set handler registration and associated methods are not renamed when the attribute is renamed in code. To remove the set handler, comment out or delete the registration. The Set Handler method can exist without the set handler registration; however, the method is not called.

In the following examples Attribute1 represents the attribute internal and external name.

The run time set handler registration appears in the Runtime class in the following region:

```
#region Runtime Set Handler Registration - Toolkit
    generated code
#endregion Runtime Set Handler Registration
```

The run time Set Handler registration for Attribute1 appears as:

```
this.RegisterRuntimeSetHandler("Attribute1.ex", new
    RuntimeSetHandlerDelegate(Attribute1SetHandler));
```

#### Remarks

The delegate is registered to the external name of the Attribute. In this example, the text "Attribute1.ex" represents the external name for Attribute1.

The run time set handler method for Attribute1 appears at the end of the Runtime class as:

```
private void Attribute1SetHandler(object sender, ref
    RuntimeSetHandlerEventArgs e)
{
    Attribute1.SetHandlerValue = e;
}
```

For additional information, see Configuring Run Time Set Handlers on page 76.

## Set Handler Code

Attribute1.SetHandlerValue enables the set handler code to appear the same for both array and non-array Attributes. You can take more control of setting the value by using the following code examples:

- Non-array attribute:

```
Attribute1 = e.Value;
```

- Array attributes:

```
if (!e.IsArrayElement)
{
    Attribute1 = e.Value;
}
else
{
    Attribute1[e.attributeHandle.shIndex1] =
e.Value;
}
```

### SetInfo Structure Event Arguments

Event arguments to provide the run time set handler event with the required data:

```
public class RuntimeSetHandlerEventArgs :
    SetHandlerEventArgs
{
    // an attribute to pass in the Set Info:
    SetInfo attributeInfo;

    // an attribute to pass out the status:
    MxStatus status;

    // Constructor
    RuntimeSetHandlerEventArgs(AttributeHandle
pAttributeHandle, SetInfo pInfo, MxStatus _status,
IMxValue pMxValue);
}
```

## Coding a RuntimeExecute() Method

Use this method to get inputs, perform calculations, set outputs, and set alarm Booleans.

---

**Note** This is the most important run time method, but it needs to be efficient and not time-consuming, or it could cause scan overruns.

---

### Example:

If the object is called Test:

```
Test_RuntimeExecute()
```

## Returning an Error Status String at Run Time

Use the `RuntimeGetStatusDesc` method to return an error message string associated with a previously returned error code from a sethandler.

### Example:

```
private void xxx_RuntimeGetStatusDesc(object sender,
    ref RuntimeGetStatusDescEventArgs e)
{
    //-----

    //  TODO: Runtime Event - GetStatusDesc
    //
    //  This routine provides a String for an
    //  error code when a client requests it.
    //  By default this method looks for an entry
    //  in the dictionary that has the
    //  DetailedErrorCode as the PhraseID.
    //
    //  You need to change this implmentation if
    //  you want to provide embedded values
    //  within your messages, or you want to use
    //  string PhraseIDs instead of integer
    //  PhraseIDs.
    //-----

    switch (e.detailedErrorCode)
    {
        default:
            e.status = GetText((int)e.detailedErrorCode);
            break;
    }
}
```

### RuntimeGetStatusDesc Event

Delegates added to this event are called when the event is fired by `ArchestraA`:

```
event RuntimeGetStatusDescDelegate
    RuntimeGetStatusDesc;
```

## Event Handler for Get Status Description

Use the following to provide the `GetStatusDescription` event with the required data:

```
class RuntimeGetStatusDescEventArgs : EventArgs
{
    // an attribute to pass out the detailed error code:
    short detailedErrorCode;

    // an attribute to pass out the status:
    string status;

    // Constructor:
    RuntimeGetStatusDescEventArgs();
}
```

## Manipulating Data Quality at Run Time

Use code to read data quality from and write data quality to attributes.

**Example:**

```
myAttribute = Input1.Value.Value;
myAttribute.Quality = Input1.Value.Quality;

if( myAttribute.Quality.isBad )
{
    // then do something like set alarm
    ...
}
```

## Manipulating the Timestamp at Run Time

Use code to read and write the time stamp at run time.

**Example:**

```
myAttribute = Input1.Value.Value;
myAttribute.Time = Input1.Value.Time;
```

## Getting Input (I/O) Values Using Utility Primitives at Run Time

Use code to read input value, status and quality from Input or InputOutput utility primitives. Use wrapper classes if the utility primitive is static; otherwise, use GetValue.

### Examples:

Input primitive:

```
myAttribute = Input1.Value.Value;
CMxStatus myStatus = Input1.ReadStatus;
```

InputOutput primitive:

```
myAttribute = InputOutput1.ReadValue;
CMxStatus myStatus = InputOutput1.ReadStatus;
myAttribute.Quality = InputOutput1.ReadValue.Quality;
myAttribute.Time = InputOutput1.ReadValue.Time;
```

## Setting Output (I/O) Values Using Utility Primitives at Run Time

Code utility primitives to write output value and read status from Output or InputOutput Utility Primitive. Use wrapper classes if the Utility Primitive is static; otherwise, use SetValue.

### Examples:

Output primitive:

```
Output1.Value = myValue;
Output1.Value.Time = myTime;
```

InputOutput primitive:

```
InputOutput1.WriteValue = myValue;
InputOutput1.WriteValue.Time = myTime;
```

Writing to the Quality of the Input, Output, or InputOutput primitive wrapper is not supported. The Quality of Value, WriteValue, and ReadValue is read-only. For example, attempting to set `InputOutput1.WriteValue.Quality = SomeAttribute.Quality` may result in an erroneous value (`InputOutput1.WriteValue.Value`) to be written to the output location.

## Accessing Data in Attributes at Run Time

For static attributes, use attribute wrappers based on the internal name to read or write values, quality, and time stamps.

**Example:**

If attribute is called `Attribute1`, just use `Attribute1` in code.

```
int i = Attribute1;
```

For dynamic attributes or attributes in a virtual primitive, use the `GetValue` and `SetValue` methods.

**Example:**

For virtuals and dynamics:

```
GetValue("attribute1");
```

or

```
GetValue( primitiveId, attributeId);
```

See the `AObjectBase` class definition for further details on the `GetValue` member.

## Accessing Data in Other Primitives at Run Time

For static primitives, use primitives and attributes wrappers based on the internal name to read/write value, quality, and time stamp.

For attributes in a virtual primitive, use the `GetValue` and `SetValue` methods.

**Examples:**

```
prim1.attribute1 = 23.0; // for static primitives  
and attributes
```

```
int i = GetValue("prim1.attribute1"); // for dynamic  
primitives and attributes
```

```
SetValue("prim1.attribute1",23.0); // for dynamic  
primitives and attributes
```

## Adding and Deleting Dynamic Attributes at Run Time

Use the following code to add/remove a dynamic attribute in run time code.

```
bool status = AddAttribute("dynAttr1",
    MxAttributeCategory.MxCategoryCalculated,
    MxDataType.MxDouble, false);

DeleteAttribute("dynAttr1");
```

## Supporting AdviseOnlyActive at Run Time

Application Server can suspend the input polling required for an attribute when that attribute is not currently being used, such as when it is not being viewed by a client, alarmed, script-referenced, or historized. For more information on the AddAttribute methods and overloads, see the *Archestra Object Toolkit Reference Guide*.

The Advise Only Active feature of the run time infrastructure determines whether an attribute is currently being used or not, and can suspend and activate that attribute at the appropriate time.

The ApplicationObject must determine which attributes are eligible candidates to be suspended when not being used. The ApplicationObject must also turn off the input polling required for an attribute when suspended, and turn the input polling back on when the attribute is activated.

To enable AdviseOnlyActive:

- 1 Determine whether or not the Object should support Advise Only Active functionality. If so, enable the **Advise Only Active supported** check box in the Object Editor.
- 2 Determine for each primitive being developed what attributes are eligible for Advise Only Active functionality. These typically will only be attributes that are updated or associated with “live” updates from external sources, usually from input type primitives. They can also be subscriptions using MX.

- 3 In either the code or the object editor, set “Advise Only Active” in the selected attributes to True.

---

**Note** In the Runtime Startup method, the auto-generated code checks whether AdviseOnlyActiveEnabled is enabled for the object. If AdviseOnlyActiveEnabled is enabled, the auto-generated code calls SuspendLocalAttribute() for each attribute supporting Advise Only Active.

---

- 4 Implement the body of the provided AttributeName\_AdviseOnlyActive() method for each attribute supporting Advise Only Active:
  - a The method shell is auto-generated.
  - b Fill in code in this method to take necessary actions to activate or suspend updates of polled data related to the specified attribute being activated. Typically, an Input primitive or InputOutput primitive wrapper is called, such as:

```
Input1.ActivateUpdatesList()
```

```
InputOutput1.ActivateUpdatesList()
```

- 5 You can also choose to take other actions, including:
  - a Activate/suspend updates on an attribute in another object using CMxIndirect.Activate() or CMxIndirect.Suspend().
  - b Activate/suspend updates on attribute in another primitive.

---

**Note** In Runtime Shutdown method, if AdviseOnlyActiveEnabled is enabled, the auto-generated code calls ActivateLocalAttribute() for each attribute supporting AdviseOnlyActive.

---

### AdviseOnlyActiveEnabled

Use this method or property to determine whether the object has AdviseOnlyActive functionality enabled. If disabled, the object must not call functions to use AdviseOnlyActive. The AOT prevents functions such as SuspendLocalAttribute() from being used if AdviseOnlyActive functionality is disabled. This method or property determines if AdviseOnlyActive functionality is enabled.



## Other AOT Wrappers for AdviseOnlyActive

In addition to the wrappers indicated in the previous section, the AOT adds the following wrapper function for AdviseOnlyActive.

### IO Utilities

Input and InputOutput utility primitives class wrappers provide methods to SuspendUpdatesList() and ActivateUpdatesList() that suspend and activate the input polling for the utility primitive.

## Triggering an Alarm at RunTime

Set the Boolean attribute representing the alarm to True to trigger the alarm. Set the attribute to False to clear the alarm condition.

### Example:

```
myCondition = true;
```

---

**Note** The Boolean attribute must have an alarm extension added to it in the Object Designer.

---

## Providing Access to External Attributes (BindTo)

The BindTo method of RuntimeBase provides a simplified method for accessing attributes in other objects at run time using CMxIndirect. For more information, see CMxIndirect on page 147.

You can use BindTo to:

- Read the value, time, and quality of an attribute or property.
- Set the value and time of an attribute. Quality cannot be set.

The Value, Quality, Datatype, Length, and Time of the external attribute can be accessed by the properties:

```
myIndirect.Value
myIndirect.DataQualityOfLastRead
myIndirect.Value.GetDataType
myIndirect.Value.Length
myIndirect.TimeStampOfLastRead
```

Like value, time can be set across object boundaries. You can access time without having to access value, but you must do this by binding to the Time property directly.

**Example:**

```
myIndirect = RuntimeBase.BindTo( "obj.attr.time", "" );
myIndirect.Value = DateTime.Now();
myTime = myIndirect.Value;      // where myTime is a
    System.DateTime variable
```

You can set both time and value together in one call:

```
myIndirect.Set( x, myTime );
```

Value and time can be set together as a pair.

**Example:**

```
private CMxIndirect myIndirect = null;
.
.
.
myIndirect = BindTo("MyTestObject.Attribute1", "",
    true);
if (myIndirect != null )
{
    myIndirect.Set( 180, DateTime.Now() );    // sets V,
    T in one call
}
```

The developer can get both Time and Value together in one call:

```
if (myindirect.StatusOfLastRead.success == -1 &&
    myindirect.StatusOfLastRead.Category ==
    MxStatusCategory.MxCategoryOk)
{
    myIndirect.Get( out x, out myTime, out myQuality );
}
```

After the BindTo operation, check the status before accessing the value as shown in the previous example.

---

**Note** Declaring the CMxIndirect in the Runtime Declarations Section makes the indirect available to all methods in the Runtime, that is, to Startup and Execute.

---

## CMxIndirect

Used for referencing external attributes. For more information, see [Providing Access to External Attributes \(BindTo\)](#) on page 145.

```
public class CMxIndirect
{
    public CMxIndirect(string _fullRefString, string
        _context, IMxSupervisoryConnection _superConn,
        RuntimeBase _rb, int _refHandle, short _statusId, int
        _statusIndex);

    public MxStatus CallBackStatus { get; }
    public string Context { get; }
    public short DataQuality { get; }
    public string FullReferenceString { get; }
    public bool IsGood { get; }
    public int RefHandle { get; }
    public MxStatus Status { get; }
    public short StatusId { get; }
    public int StatusIndex { get; }
    public CMxValue Value { get; set; }
    public CMxTime Time { get; set; }
    public void Set { CMxValue value, CMxTime time };
    public void Get { out CMxValue value, out CMxTime
        time, out short quality };
}
```

## Associating an ArcestraA Editor Control with an Attribute in Code

Normally, ArcestraA editor controls are statically configured to point to a single attribute to be configured on the editor tab.

However, on occasion, and especially with virtual primitives, it is useful to dynamically bind the editor control to a particular attribute by setting the `Attribute` property of the control.

For example, with a text box control, the following shows how to set the attribute name in code:

```
aaTextBox1.Attribute = "myAttribute1";
```

## Referencing Attributes Using GetValue and SetValue

The AOT enables the use of SetValue and GetValue to reference the following types of attributes:

- Child primitive attributes
- Dynamic attributes
- Virtual primitive attributes

You can reference these attributes using SetValue and GetValue with a relative string reference. The string reference is relative to the primitive that contains the code, and can be prefixed with the following modifiers:

- “me”
- “myparent”
- “myobject”

---

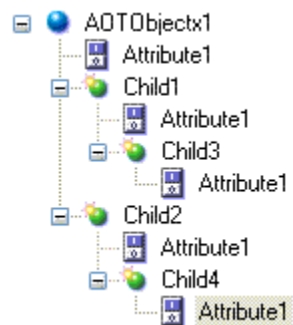
**Note** The prefix “me” is implied when no prefix is provided.

---

The GetValue and SetValue methods use the primitive external name and attribute external name as shown in the following example:

```
GetValue(PrimitiveExternalName.AttributeExternalName)
```

Examples included in this section are based on the following hierarchy:



## Local References

To reference any local attributes, use the attribute name with no prefixes or scope.

Using the *me* prefix explicitly sets the reference to local and can be used to make it clear what type of reference is required.

Either of the following statements in Child2 gets the value of AOTObjectx1.Child2.Attribute1:

```
GetValue("Attribute1");  
GetValue("me.Attribute1");
```

## Referencing Down (child)

To reference attributes of a child primitive, prefix the reference string with the name of the child primitives.

The following statement in Child2 gets the value of AOTObjectx1.Child2.Child4.Attribute1:

```
GetValue("Child4.Attribute1");
```

The following statement in AOTObjectx1 gets the value of AOTObjectx1.Child2.Child4.Attribute1:

```
GetValue("Child2.Child4.Attribute1");
```

## Referencing Up (parent)

To reference attributes of a parent primitive use the *myparent* prefix.

The following statement in Child2 gets the value of AOTObjectx1.Attribute1:

```
GetValue("myparent.Attribute1");
```

The following statement in Child4 gets the value of AOTObjectx1.Child2.Attribute1

```
GetValue("myparent.Attribute1");
```

The *myparent* prefix cannot be used more than once in a reference. To reference attributes of objects or primitives more than one level higher, you must use *myobject* to locate the reference to the top of the hierarchy and then work relative to that location.

The following statement in Child4 gets the value of AOTObjectx1.Attribute1:

```
GetValue("myobject.Attribute1");
```

The following statement in Child4 gets the value of AOTObjectx1.Child1.Child3.Attribute1:

```
GetValue("myobject.Child1.Child3.Attribute1");
```

## Array Usage

```
// Increment the 3rd element of an array
FloatArray1[3]++;

// Increment all the elements of an array
for (short counter = 1; counter <= FloatArray2.Length;
    counter++)

{
    FloatArray2[counter]++;
}
```

## The External Build Process

The build process is made up of many stages. In general, these stages all occur as a single event.

Sometimes you may want to execute only part of the build process or to add additional events in the middle of the process. To support this, you can start a solution build from the command line as well as repackage the object with the Packager application.

---

**Note** The project must be built before you execute the Packager, because the Packager requires the aaDEF file and associated assemblies create by the build.

---

## Command Line Recompile Object

Perform these processes from a command prompt with the Visual Studio Environment variables loaded.

### Build Process - Recompile (debug version)

```
C:\>devenv "C:\ Projects\AOTObject88\AOTObject88.sln"
/build Debug
```

### Build Process - Recompile (release version)

```
C:\>devenv "C:\ Projects\AOTObject88\AOTObject88.sln"
/build Release
```

## Command Line Repackage Object

The AOT includes a utility called Packager.

The Packager packages the object using the files created by the build, that is, it packages the aaDEF files and assemblies created by the build. This allows a build process to repackage the aaPDF after the object is rebuilt using the command line build.

You can start the Packager as a Windows Form application or execute it from the command line using the following switches:

/q - command line mode, no Form.

/f <filename> - The name of a text file containing information needed by DesignServer to repackage the Object.

The file is automatically generated in the Output directory when the AOT builds the object using the name DesignServerInfo.txt. It contains the name of the root aaDEF file and all of the paths to the core object dlls as relative paths, that is, the project is portable.

### Example

```
C:\Program Files\Archestra\Toolkits\AOT\Bin>packager /q
/f "C:\
Projects\AOTObject90\Output\designserverinfo.txt" /a
"C:\Utility Dlls\"
```

---

**Note** This feature allows you to modify the aaDEF file and repackage the Object.

---

## Advanced Techniques

Use these techniques to configure an object or primitive completely in code using C# attributes. Using the integrated Object Designer causes these C# attributes declarations to be automatically added to the code.

## Configuring an ArcestraA Attribute in Code

You can add ArcestraA attributes to the object. They can be used locally in code as C# variables.

The ArcestraA attributes are declared in the object project as C# variables using a CMx Data Type.

When the object is parsed, properties are automatically added to the Runtime and Configtime class for each ArcestraA attribute defined in the Object class. Parsing occurs while:

- Refreshing the AOT Object Design View
- Opening the AOT Object Designer
- Saving an object in the AOT Object Designer
- Executing build or code validation

The attribute properties allow you to access ArcestraA attributes as though they were strongly typed C# values.

When you reference the ArcestraA attribute in code, the property provides a C# typed wrapper to the attribute using SetValue and GetValue access. The wrapper provides access to Quality, Time, Data Type, and Array Length.

The following table lists the data types and associated C# conversions.

ArcestraA Data Type	C# Data Type
Boolean	bool
Integer	int
Float	float
Double	double
Time	DateTime
Elapsed Time	TimeSpan
String	string
Big String	string
Attribute Reference	string
Custom Enumeration	string array
DataType	ArcestraA.Core.MxDataType
Custom Structure	ArcestraA.Core.MxCustomStruct
MxStatus	ArcestraA.Core.MxStatus



ArchestrA Data Type	C# Data Type
Variant	CMxValue
InternalDumpLoadData	ArchestrA.Core.MxCustomStruct (special)
InternalFailoverData	ArchestrA.Core.MxCustomStruct (special)

**Note** There is no direct conversion from an internationalized string to a C# type. The internationalized string can be represented as an internationalized string structure (string value, locale) or by setting the locale for the attribute, which then allows you to reference the attribute as a C# string.

The variable declaration can be decorated with C# attributes to enable the configuration of the following attribute properties (excluding special data types):

- External Name
- Category
- Security
- Calculated Quality and Time
- Frequently Accessed
- Alarm Extension
- History Extension

**Note** The special data types of CMxInternalDumpLoadData and CMxInternalFailoverData are created and maintained by the toolkit. These data types are not intended for general use. These ArchestrA attributes store data for recreating dynamic attributes and child primitive instances on dump/load and failover.

### Specifying the Arcestra Attribute Array Length

You can set and get the array length of an attribute array (static) at config time or run time using the following syntax:

#### Syntax

```
AttributeName.Length = n;  
n = AttributeName.Length;
```

#### Parameters

##### *AttributeName*

Represents the array attribute name.

##### *n*

Represents an integer value.

#### Remarks

You can set the array length of a dynamic attribute array at config time or run time using one of the following Get and Set methods:

```
n = GetNumElements("AttributeName");  
  
n = GetNumElements(AttributeID, PrimitiveID);  
SetNumElements("AttributeName", n);  
  
SetNumElements(AttributeID, PrimitiveID, n);
```

Where, *AttributeName* is the array attribute name and *n* is an integer value.

You can apply the `GetNumElements` and `SetNumElements` methods to attributes. However, when you rename an attribute using the Object Designer, the attribute name referenced by these methods is not updated.

## Referencing Attributes from the Editor of the Object

To reference attributes directly from the Editor Project of the object, you must implement the `GetData()` and `SetData()` methods provided by the framework.

In the examples, `Attribute1` is a float array with four elements, `Attribute2` represents is a float.

#### Attribute Get Example:

```
//Get an Attribute value  
float MyData2 =  
    (float)Convert.ToDecimal(GetData("Attribute2"));
```

#### Attribute Set Example:

```
//Set an Attribute value  
SetData("Attribute2", 9.0);
```

**Array Attribute Get Example:**

```
//Get a single value of an Array Attribute
float MyData =
    (float)Convert.ToDecimal(GetData("Attribute1[1]"));
//Get an Array Attribute
object[] MyArrayValues = new object[4];
MyArrayValues = (object[])GetData("Attribute1");
```

**Array Attribute Set Example**

```
//Set an Array Attribute using a locally declared array
object[] MyArrayValues = new object[4];
MyArrayValues[0] = 1.0;
MyArrayValues[1] = 1.1;
MyArrayValues[2] = 1.2;
MyArrayValues[3] = 1.3;
SetData("Attribute1", MyArrayValues);
```

---

**Note** The array index is 1-based. There are no errors or warnings to indicate that a zero value has been passed to the array index from the editor of the object.

---

## Local Attribute Wrappers

Based on the attribute category, AOT adds an Attribute property to the Configtime class or the Runtime class for each attribute declared in the Object class. You can use the local attribute wrapper to access attributes using the attributes name `attribute_internalname`.

The following code example represents an auto-generated Attribute property added to the Configtime or Runtime class for an attribute that supports read and write. In the example, the Set statement would be excluded if the attribute were read-only.

```
private CMxBoolean Attribute1
{
    get { return InternalReferenceOnly.Attribute1; }
    set { InternalReferenceOnly.Attribute1.Set(value); }
}
```

The get property provides access to the attribute wrapper and allows you to access the features of the wrapper, such as quality and security.

The set property is limited to setting the value. This is how the property is used when the attribute is on the left side of an assignment operator, for example, `Attribute1 = 10`. You can assign values using this short method. It provides type checking and automatic type conversion of the value.

The property does not provide any other set access to the wrapper.

The following table shows the relationship between the attribute category and the attribute properties Get and Set added to the Configtime and Runtime classes.

Attribute Category	Configtime Class	Runtime Class
PackageOnly	Get and Set	- -
PackageOnly_Lockable	Get and Set	- -
Constant	Get Only	Get Only
Writeable_C_Lockable	Get and Set	Get Only
Writeable_UC	Get and Set	Get and Set
Writeable_UC_Lockable	Get and Set	Get and Set
Writeable_USC	Get and Set	Get and Set
Writeable_USC_Lockable	Get and Set	Get and Set
Calculated	- -	Get and Set
Calculated_Retentive	- -	Get and Set
Writeable_S	- -	Get and Set
Writeable_U	- -	Get and Set
Writeable_US	- -	Get and Set
SystemInternal	Unsupported	Unsupported
SystemSetsOnly	Unsupported	Unsupported
SystemWriteable	Unsupported	Unsupported

# Appendix B

## Development Best Practices

When developing your object, you should follow certain guidelines to ensure correct functionality and to avoid common pitfalls. See the following sections for guidelines and tips on developing config time code, run time code, and the custom object editor.

### General Guidelines

Use the following general guidelines when developing your object.

#### Naming Conventions

Use attribute and primitive names that are consistent within your object and with other objects in the ArcestrA environment. This makes it easier for operators and system engineers to browse the ArcestrA object namespace.

## Naming Restrictions

- The following characters are invalid in ArcestrA names: (space) . + - \* / \ = ( ) ` ~ ! % ^ & @ [ ] { } | : ; " , < > ?
- Non-English (“localized”) characters are supported in the external names of attributes, objects and primitives, but not in their internal names.
- You can use periods to create a logical naming hierarchy for attributes and primitives (see Creating a Logical Attribute Hierarchy on page 161). The maximum length of each identifier between periods is 32 characters. The maximum length of the entire name including all identifiers is 329 characters for attributes and 255 characters for primitives.

## ArcestrA Naming Standards and Abbreviations

The following table lists a set of standards for naming the attributes and primitives of ArcestrA objects. While you may not be able to apply it universally, you should follow it whenever possible to promote consistency across ArcestrA objects.

Instead of	Use	Comment
acknowledge	Ack	
acknowledged	Acked	
Address	Addr	
Alarm	Alarm	Don’t abbreviate.
Attribute	attr	
Automatic	auto	
Average	avg	
Cascade	casc	
command	cmd	OK to spell out “commandable” and “commanded.”
configuration	config	OK to spell out “configure” or “configured.”
connection	connection	Don’t abbreviate.
Control	ctrl	
controller	ctrlr	
Count	cnt	

Instead of	Use	Comment
dataaccess	DA	
description	desc	
destination	dest	
deviation	dev	
Different	diff	
Directory	dir	
dynamicdataexchange	DDE	
engineeringunits	EngUnits	
Enum	enumerationset	
EU	EngUnits	Use “EngUnits”
evaluation	eval	
External	external	Don’t abbreviate.
GloballyUniqueID	GUID	All uppercase
High	hi	
Identifier	id	
Interval	period	Don’t use “interval” to specify a time between cyclic events. Use “period” instead.
Low	lo	
Manual	man	
maximum	max	
message	msg	
Minimum	min	
mxreference	reference	
Number	cnt	“Cnt” is short for “count.” You can use “Number” if it refers to an index, not a count. For example, “TelephoneNumber” is OK because it specifies a literal number, not a count of telephones.
Object	object	Don’t abbreviate.
Output	OP	Abbreviation only used for PID controller

Instead of	Use	Comment
password	password	Don't abbreviate.
processvalue	PV	
Put	set	Use "set" instead of "put."
Queue	queue	Don't abbreviate.
randomaccessmemory	RAM	
rateofchange	ROC	
Received	rcvd	
reference	reference	Don't abbreviate.
Server	server	For attribute names, don't abbreviate. For file names, OK to abbreviate to "svr."
Setpoint	SP	Abbreviation only used for PID controller
Solicit	solicit	Don't abbreviate.
Statistics	stats	
userdefinedattribute	UDA	
Value	value	Don't abbreviate.

### Additional Naming Guidelines

- When a name contains multiple words, **begin each word with a capital letter**. For example, "Average Page Faults" becomes "PageFaultsAvg." When one of the words itself is an acronym (e.g. "CPU"), still capitalize the word following the acronym. For example, "CPU Load" becomes "CPULoad," not "CPUload."
- **Place adjectives after the noun.** This causes objects of interest (typically, the noun) to be grouped together in an alphabetical list. For example, use "FlowAvg," "FlowMax," and "FlowMin" instead of "AvgFlow," "MaxFlow," and "MinFlow."
- Use **plural names for** attributes that are **arrays**.
- **Avoid unnecessary adjectives** when the noun itself is understood. For example, for an attribute that indicates the CPU load, don't use "CPULoadCurrent" or the abbreviated "CPULoadCur" when "CPULoad" is enough.



## Creating a Logical Attribute Hierarchy

An object's primitives naturally create a hierarchical namespace of attributes. Every attribute has a Hierarchical Name that includes the external name of the primitive that contains it. Without care, this namespace may expose the underlying primitive structure of the object to end users, which is usually undesirable from a useability standpoint.

You can use two strategies to address this issue: unnamed primitives, and periods in attribute names.

### Using “Unnamed” Primitives

When appropriate, primitives can be “unnamed,” that is, their external name is empty. This causes all of the primitive's attributes to appear to belong to the primitive's container (either the parent primitive or the object itself).

### Using Periods in Attribute Names

By using a period in an attribute name, you can create a hierarchy within the object namespace that is independent of the object's primitive structure.

This is recommended when an attribute is related to a contained primitive. In these situations, the name of the attribute should always be the same as the contained primitive's name, or extend the contained primitive's name using a period.

For example, if your object includes an alarm primitive named “AlarmHiHi,” you could create an object attribute named “AlarmHiHi.Condition” that sets the condition for the alarm. This allows the end user to refer to the alarm-related attributes in a consistent, intuitive way.

## Working with the Logger

Use the Logger only for tracing trapped software errors or diagnostics, and only use it sparingly in production objects. Do not use it to provide information that is intended for operators. Operators don't typically look at the Logger information, but rely on alarm and quality information instead.

If you use the Logger to trace diagnostic information, make sure that the logging does not continue indefinitely (e. g. on every Application Engine scan). Otherwise, performance issues occur.

If you use the Logger to provide debugging information during development, either remove the logging calls before releasing the object to production, or change them so that logging only occurs when a custom log flag is set.

For more information on the Logger APIs, see the *ArchestrA Object Toolkit Reference Guide*.

## Raising Data Change Events

Wonderware Application Server supports generating Application Data Change events to report significant or unexpected data value changes to the alarm and event sub-system. To generate a Data Change event, use the `SendEvent` method of the object's run time component.

These events are intended for data changes that occur during the `execute` method of the object. They can be used to record data changes in event history. However, do not use them for data changes initiated by a run time user ("user sets"). This causes duplication, because these data changes are already logged by the ArchestrA infrastructure.

If you implement these events, you may want to provide a configuration option to enable or disable them. Users may not always want them reported, especially in the case of "noisy" data.

## Changing or Enforcing the Length of an Array

ArchestrA array lengths are dynamic. Run time or config time clients can change the length of an array by writing a new set of values to the array. The array length can also be changed at any time by the object itself. To enforce a fixed array length, check incoming values by using a set handler.

## Guidelines for Config Time Code Development

Use the following guidelines for developing good config time code.

### Ensuring Galaxy Dump/Load Support

Make sure that your object can be processed by the IDE's Galaxy Dump/Load feature without generating warnings. This feature allows users of your object to dump object instances to a CSV file, modify their configuration, and then subsequently reload them. To ensure that this process works smoothly, you must follow certain rules:

- **Keep all validation rules in the config time code.** Do not rely on the custom editor code to maintain the integrity of the object (e. g. keeping two attributes consistent with each other). The Galaxy Load feature does not use the editor code when importing objects. It only calls the config time code's `OnValidate` method. Therefore, any validation rules in the editor code are ignored during a Galaxy Load operation.
- **Set handlers must quietly accept a new value equal to the current value.** An object should not reject a set to an attribute when the value being set is the same as the previous value, even if the object's configuration does not currently allow that attribute to be changed. Coding this way prevents "noise" when Galaxy Load is run.
- **Avoid "write-only" attributes that modify the object's namespace.** An example of this is to have a set handler add, remove, or rename a primitive whose name was passed in as the value of a "write-only" attribute. At first, this appears to be a sensible way for an editor to pass a parameter to a config time method. However, if that information is not subsequently exposed as a readable attribute, there is not enough exposed information in a dumped CSV file to recreate the object from its configurable attributes when it is loaded.

Instead, you could store the names of the desired primitives in an attribute containing an array of strings. This array can have an associated set handler that maintains the number of primitives and their names. In this case, the Galaxy Load feature can load the object successfully, because the exposed array contains all the information required for the config time logic to recreate the primitives.

## Determining the Configuration Status

Every ArcestrA object has an associated configuration status: Good, Bad, or Warning. This status is based on the individual statuses reported by the primitives within the object. To set the status, use the OnValidate config time event.

The object status reported in the ArcestrA IDE is based on the worst status reported by any primitive within the object.

- Only set the status to **Bad** to prevent an object from being deployed. In general, you should design an object so that it can be deployed successfully with minimal configuration, and only set an object's status to Bad if deploying it in its current configuration would be impossible or dangerous.
- Use **Warning** status to mark an object as having a potentially incorrect, but still deployable configuration. For example, an object that still uses its default settings.

## Changing an Attribute's Data Type at Config Time

Sometimes you may need to change an attribute's data type at configuration time. Normally, you will only do this for an attribute that you defined as a Variant (unspecified data type) in the Object Designer.

To change the attribute's data type, modify the attribute's data type property. For more information, see the *ArcestrA Object Toolkit Reference Guide*. For example, set the attribute's data type property to a value of MxDouble to indicate that the attribute's type is Double.

After changing the data type using the methods of the CMxVariant wrapper, the value is automatically initialized with the default value for that data type. If you change the data type using a Set call, you must initialize the new value manually.

# Guidelines for Run Time Code Development

Use the following guidelines for developing good run time code.

## Returning Warnings During Deployment

During deployment, objects can return a warning to the ArchestrA IDE user if the target environment is inconsistent with the object's configuration. The object continues to run despite the warning.

Returning warnings will rarely be necessary for ApplicationObjects, but if you want to do so, use the AddWarningMessage method. For more information, see the corresponding information in the *ArchestrA Object Toolkit Reference Guide*.

## Avoiding Application Engine "Overscans"

The Application Engine requires that runtime object method calls be nonblocking and relatively short in duration (on the order of 100 microseconds). You can create threads for slow or potentially blocking activities that would violate these requirements. However, make sure to terminate all threads when the object is shut down.

## OnScan/OffScan Behavior

You can define custom actions that are executed when your object goes OffScan. At a minimum, you should set the quality of any attributes that have the CalculatedQuality option enabled to Bad. When the object goes OnScan again, set the quality of these attributes back to Good.

## Dealing with Quality

Every attribute has an associated OPC-compliant data quality value that is a 16-bit word. The high-order byte is vendor-specific. In an ArchestrA environment, it is reserved for future use and currently always set to zero. The low-order byte specifies the OPC quality. It has three possible major quality states: Good, Uncertain, and Bad.

The ArchestrA environment additionally treats one substate of the OPC “Bad” state as the special quality of “Initializing.” Initializing quality is Bad quality with the Initializing bit set.

- If the quality of an attribute’s value is **Good**, the associated value can be trusted and used. However, the value could still be out of range or invalid (e. g. NaN). Your object must check for these conditions separately.
- If the quality is **Uncertain**, the associated value can be used, but there is some doubt about the integrity of the value. For example, this could be the case when manually overriding an attribute that is normally calculated automatically. When using an input with Uncertain quality, do it with care and mark the resulting (calculated) attribute as Uncertain also.
- If the quality is **Bad**, there are a number of possible reasons. These include:
  - The object that contains the attribute set its quality to Bad because insufficient or bad data was available.
  - The infrastructure returns Bad quality for an attribute when the attribute cannot be accessed within Message Exchange. For example, the target attribute does not exist or communication is faulty.
  - A field device may not be connected or accessible, resulting in Bad inputs that propagate through the system.
- **Initializing** quality is a form of Bad quality that requires special attention. It is temporary and only occurs while an object is initialized. It lasts until the object receives its first input data value. The quality then goes to Good, Bad (non-Initializing) or Uncertain.

Before you use data values in calculations and logic, always check their quality. For example, it does not make sense to calculate the average of two values if one or both values have Bad quality, since Bad quality indicates that the value is not to be used or trusted. Instead, in this case, you should skip the calculation of the average and set the resulting attribute to Bad quality itself.

The ArchestrA infrastructure does not automatically enforce a specific value (such as IEEE NaN) when quality is Bad, or a specific quality (such as Bad) when a value is NaN. Your object must check for these conditions before using any values in logic or calculations. For example, a float value read from a field device may have a value of NaN but Good quality. In that case, the object must be aware that the value may be unusable for a calculation. Conversely, a value read from a UDA attribute may be 4.3 but Bad quality. Again, the object must inspect the quality first, realize it is Bad, and take appropriate action.

### Best Practices for Dealing with Quality

Best practices for dealing with quality include:

- If an attribute's value is set by the object's run time logic, enable the **Supports Calculated Quality and Calculated Time** option for that attribute in the Object Designer.
- For static attributes (i. e. attributes that you didn't create programmatically), you can use the auto-generated wrapper to access the attribute's quality. For example:

```
Attribute1.Quality = DataQuality.DataQualityGood;
```

- Set such attributes to Bad quality when the object goes OffScan. Set them to Initializing quality when the object goes OnScan.
- Do not use an input value with Bad (including Initializing) quality in a calculation. Instead, set the result quality Bad or Initializing (if input was Initializing) and leave the value at the last value. (For a float or double result, consider setting the result to NaN.)
- Do not use a NaN (float or double) input in a calculation. Instead, set the result to Bad quality and leave the result value at the last value, or set it to NaN if it is a float or double.

- If an illegal combination of input values exists, set the resulting quality to Bad.
- Optionally, provide an option to report a “bad value” alarm when a result value has Bad quality. Do not report a “bad value” alarm when a value has Initializing. Otherwise, transient alarms occur when the object goes OnScan.
- Do not trigger any other alarms when the quality of an attribute goes Bad. For example, do not trigger a PV change-of-state alarm when the PV goes to some default state after its quality goes Bad. Instead, always use a separate alarm for bad value reporting.
- Inputs with Uncertain quality can be used with care. Set the result to Uncertain quality also to indicate its questionable status.
- Do not generate Logger messages when setting an attribute to Bad quality in the cases outlined above.
- Do not attempt to change the quality of an input, output, or input/output by using its wrapper. This is not supported and may result in unexpected I/O values being written.

## Dealing with Timestamps

Observe the following guidelines when dealing with calculated attributes:

- In most cases, it is appropriate to enable the **Supports Calculated Quality and Calculated Time** option for values whose value is calculated at run time.
- For attributes that are updated based on the value of an input or input/output, set the time of the attribute to the input value's time. This ensures that timestamps are propagated properly.
- When setting the value of a calculated attribute that is not connected to an input, it is usually best practice to set the time to the current time. For attributes that have the **Supports Calculated Quality and Calculated Time** option enabled, the system automatically does this when you set the value.
- When setting the value of an attribute based on the value from another object, make sure to set the time of the attribute to the time from the CMxIndirect value. This ensures that timestamps are propagated properly.



## Dealing with Outputs on Object Startup

When developing objects associated with field devices, such as a PLC, there are two main scenarios for what happens when the object starts executing at run time:

- In the more common case, the object mirrors the PLC's data. In this scenario, when starting or resuming run time execution, the object must initialize its own state to match the PLC data. The object only writes data to the PLC when an operator, script etc. requests such output. It must *not* automatically write any data to the PLC when it is started or shut down, set OnScan/OffScan, deployed/undeployed, etc. This should be the default scenario.
- Rarely, the inverse may be necessary, and the PLC should mirror the object's data. In this scenario, when the object starts or resumes run time execution, it writes to the PLC to force the PLC to match the object's data. For example, when resuming execution after a failure, the object might use checkpoint data to restore the state before the failure. This scenario is much less common since the PLC generally is in control upon restarts.

In keeping with these scenarios, the utility primitives that do outputs (Output and InputOutput primitives) never do an output unless the object itself requests it. The object is in complete control of when outputs occur. Therefore, if you want to implement the second scenario, you must implement custom code that performs the outputs to initialize the field device.

---

**Note** You can check the `ESTARTUPCONTEXT` input parameter to the Startup run time event handler to see why the object is starting up (deployment, etc.).

---

## Dealing with the Quarantine State

When an unhandled software error is detected in a primitive, the object is placed in a quarantine state indicating a bug in the primitive code. As a result, the primitive's set handlers, Execute method, and other methods are no longer called. The only remaining calls that the primitive can receive are those related to the shutdown or undeployment of its associated object. However, you can still read the object's attributes to gather troubleshooting information about the object state at the time of the failure, because this doesn't involve calling any methods.

When an object is quarantined, the hosting engine raises an alarm that remains active until the object is undeployed.

## Ensuring Failover Support for Run Time Dynamic Attributes

Note the following guidelines for run time code when working with failover/checkpointing support for dynamic attributes:

- Attribute information may become outdated if the dynamic attribute is modified after it is created. To ensure that attributes are re-created correctly after a failover, call the `UpdateDynamicAttributeData()` method immediately after changing an attribute's name, data type, category, security classification or set handler flag at run time. For more information, see the *Archestra Object Toolkit Reference Guide*.
- After you change the value of a dynamic attribute, call the `CheckpointDynamicAttributeData()` method either immediately or during the next scan cycle. This ensures that the attribute's values are kept current in the failover environment. For more information, see the *Archestra Object Toolkit Reference Guide*.
- To restore dynamic attributes and their values at run time startup, call the `RestoreDynamicAttributes()` method from the object's Startup event handler. For more information, see the *Archestra Object Toolkit Reference Guide*. You can check the `ESTARTUPCONTEXT` input parameter to the Startup event handler to see why the object is starting up (deployment, failover, etc.).

## Guidelines for Custom Editor Development

Use the following guidelines for developing good code for your custom object editor.

### Keeping Validation Rules out of the Editor Code

Do not rely on the custom editor code to maintain the integrity of the object (e. g. keeping two attributes consistent with each other). It should always be possible to create an object using a standalone configuration utility which configures the object's attributes without any involvement by the object's custom editor. Therefore, don't put validation rules in the custom object editor code. Instead, put them in the OnValidate config time event that is provided for this purpose.

### Creating a Complete Editor

Make sure that your custom object editor allows the user to edit every non-hidden configurable attribute of your object. Remember that you may even have to add non-configurable attributes to the editor, because their security classification might still be editable.



---

# Appendix C

## Sample Projects

The ArchestrA Object Toolkit comes with two sample ApplicationObjects:

- Monitor object
- WatchDog object

By default, the projects for these objects are installed in the C:\Program Files\Wonderware\Toolkits\ArchestrA Object\Samples folder. You can examine these objects to learn more about the C# code generated by the ArchestrA Object Toolkit. This appendix gives a short overview of what these objects do and what their structure looks like.

---

**Note** On a 64-bit operating system, projects for these objects are installed in C:\Program Files (x86)\Wonderware\Toolkits\ArchestrA Object\Samples.

---

### The Monitor Object

The Monitor object is a very simple ApplicationObject that reads an external input value and calculates its average. It also allows the user to output a new value that is below a configurable limit. It has no custom config time code and no alarm or history settings.

## Object Structure

The Monitor object uses the following primitives:

- InputOutput primitive to read and write the external value; external name: PVInputOutput

The Monitor object has the following custom attributes:

Name	Type	Category	Description	Additional Settings
PV	Float	Writeable _US	Process value	Calculated Quality, Frequently Accessed, Run Time Set Handler
PVHiLimit	Float	Writeable _USC	Limit value for PV output	
PVInputAvg	Double	Calculated Retentive	Average value	Calculated Quality
ResetInputAvg	Boolean	Writeable _US	Resets the average value	Run Time Set Handler

## Custom Object Editor

The custom editor of the Monitor object has only one custom tab with controls to configure the following attributes:

- PVHiLimit
- PVInputOutput.Reference
- PVInputOutput.SeparateFeedbackConfigured
- PVInputOutput.ReferenceSecondary

## Run Time Code

The Monitor object has the following custom run time code:

- **SetScanState event:**
  - When going OnScan, set the quality of calculated attributes to Initializing.
  - When going OffScan, set the quality of calculated attributes to Bad.

- **Execute event:**
  - Get the new input value and write it to the PV attribute.
  - Set the PV attribute's quality to the quality of the new input value.
  - Calculate the new average value and write it to the PVInputAvg attribute.
- **GetStatusDesc event:** Return messages for custom error codes.
- **Set handler for PV attribute:** Check that new value is less than PVHiLimit.
- **Set handler for ResetInputAvg:** Reset the average calculation.

## The WatchDog Object

The WatchDog object demonstrates basic input/output, alarming, and historization. It also shows how to use virtual primitives. The object:

- Monitors whether an input bit has changed.
- Calculates the time since the bit last changed.
- Raises an alarm if this time exceeds a timeout limit.
- Historizes this time.
- Provides optional statistics via a virtual primitive: average and maximum time since last change, time of last timeout, total number of timeouts.

## Object Structure

The WatchDog object uses the following primitives:

- Input primitive to read the external bit that should be monitored; external name: MonitoredBit
- Alarm primitive
- History primitive
- Custom virtual local primitive to calculate statistics; external name: Stats

The WatchDog object has the following custom attributes:

Name	Type	Category	Description	Additional Settings
TimeSinceChange	Elapsed Time	Calculated	Time since the MonitoredBit value last changed state	Historizable
Timeout.Limit	Elapsed Time	Writeable_U SC_Lockable	Limit value for timeout alarm	Frequently Accessed, Run Time and Config Time Set Handlers
Timeout	Boolean	Calculated	Set when timeout has occurred	Alarmable
Stats.Enable	Boolean	PackageOnly _Lockable	Enable/disable Stats virtual primitive	Config Time Set Handler

The Stats virtual primitive has the following custom attributes:

Name	Type	Category	Description	Additional Settings
Stats.DelayAverage	Elapsed Time	Calculated	Average time since last change	
Stats.DelayMax	Elapsed Time	Calculated	Maximum time since last change	
Stats.TimeoutCnt	Integer	Calculated	Timeout count	Historizable
Stats.LastTimeout	Time	Calculated	Time of last timeout	
Stats.Reset	Boolean	Writeable_U	Reset statistics	Run Time Set Handler



## Custom Object Editor

The custom editor of the WatchDog object has two custom tabs with controls to configure the following:

- **General tab:**
  - Input bit (MonitoredBit.InputSource)
  - Enable statistics (Stats.Enable)
  - Enable history and alarms for attributes
- **Advanced tab:**
  - History and alarm settings for attributes

## Config Time Code

The WatchDog object has the following custom config time code:

- **Set handler for Stats.Enable attribute:**  
Enable/disable the Stats virtual primitive.
- **Set handler for Timeout.Alarmed attribute:**  
Enable/disable the timeout alarm primitive.
- **Set handler for Timeout.Limit attribute:** Check that the new value is positive.
- **Set handler for TimeSinceChanged.Historized attribute:** Enable/disable the history primitive for the TimeSinceChanged attribute.

## Object Run Time Code

The WatchDog object has the following custom run time code:

- **Startup event:** Initialize the time of last change.
- **Execute event:**
  - Get the new input value.
  - Calculate the time since the last change and write it to the TimeSinceChanged attribute.
  - If the time exceeds the timeout limit, raise an alarm by setting the Timeout.Condition attribute to true.
- **GetStatusDesc event:** Return messages for custom error codes.
- **Set handler for Timeout.Limit attribute:** Check that the new value is positive.

## Stats Primitive Run Time Code

The Stats virtual primitive has the following custom run time code:

- **Execute event:** Calculate statistics (average/maximum time since last change, timeout count, last timeout time) and write them to the appropriate attributes.
- **Set handler for Reset attribute:** Reset all statistics attributes.

# Appendix D

## ArchestrA Data Types

Objects that you create using the ArchestrA Object Toolkit can have attributes of any standard data type that is supported in the ArchestrA environment. This appendix describes the available data types and provides some notes on their recommended use.

### List of ArchestrA Data Types

The ArchestrA environment supports the following data types. The defaults are used at startup time or when there is no data available. For notes on using each type correctly, see *Using Data Types Correctly* on page 185.

For additional information on the operations supported by each data type, see the class documentation in the *ArchestrA Object Toolkit Reference Guide*.

Data Type	Valid Values	Notes
Boolean	True, False (default: False)	
Integer	-2147483648 to 2147483647, signed (default: 0)	
Float	3.40282 E+38 to -3.40282 E+38, signed (default: NaN)	32-bit IEEE single-precision floating point, used when 6-7 significant digits are needed. Smallest representable absolute value is 1.175 E-38.

Data Type	Valid Values	Notes
Double	1.79769 E+308 to -1.79769 E+308, signed (default: NaN)	64-bit IEEE double-precision floating point, used when 15-16 significant digits are needed. Smallest representable absolute value is 2.23 E-308.
String	0 to 1024 characters, default: empty string	Variable-length Unicode string, size: 4 + 2*n bytes (n = number of characters)
Time	Microsoft FILETIME values (default: “zero time”)	64-bit FILETIME value in UTC (Coordinated Universal Time). Represents the number of 100-millisecond ticks since January 1, 1601, 00:00:00 (“zero time”).
Elapsed Time	Number of 100-ms ticks, signed (default: 0)	Stored as a 64-bit FILETIME structure. For example, -1 corresponds to a duration of “-00:00:00.0000001”.
Attribute Reference	Valid reference strings (default: empty string and null handle)	Standard structure containing a reference string and MxHandle (bound or unbound). A string of “---” results in a null handle, and no warning is generated when the object is validated (i. e. when the user saves the object configuration). A string of “---.---” results in a null handle, but a warning is generated on validation.
MxStatus	Default: Success	Standard structure containing access status information for a Message Exchange call.
Data Type	Enumeration, see notes (default: MxNoData)	Data type of an attribute. Valid enumeration values are: MxNoData, MxBoolean, MxInteger, MxFloat, MxDouble, MxString, MxTime, MxElapsedTime, MxReference, MxStatus, MxDataType, MxSecurityClassification, MxQuality, MxCustomEnum, MxCustomStruct, MxInternationalizedString and MxBigString.
Custom Enumeration	Default: ordinal=1, String=String1	Enumerations start at the value 1. Zero is not a valid ordinal value for an enumeration.

Data Type	Valid Values	Notes
Custom Structure	Default: GUID = 0, length = 0.	Provides support for custom data in the form of a GUID and byte array.
Internationalized String	0 to 1024 characters, default: empty string	A vector of strings and corresponding locale IDs in the configuration database. An MxString at runtime.
Big String	0 to 2147483647 characters, default: empty string	Variable-length Unicode string, size: 4 + 2*n bytes (n = number of characters)
Variant	N/A	Use this data type if the actual type of an attribute cannot be determined in advance.

## Coercion Rules for ArcestrA Data Types

In some cases, ArcestrA data types can be “coerced” when reading from or writing to an attribute. This means that the client can specify a different data type than the attribute actually has. The value is implicitly converted from the specified data type to the data type required by the attribute.

For example, if an output configured for Boolean values sends a value to an Integer attribute, the write operation succeeds and the Boolean value is automatically converted to a 0 or 1.

Coercion is only supported for some combinations of data types. Trying to use coercion for unsupported combinations results in an exception being thrown. Also, coercion generally fails in case of an overflow, i. e. if the value is outside the valid range of the target type.

The following tables list the supported combinations and required value formats.

### Coercion from Boolean Values

To ...	Values / Notes
Integer	False = 0, True = 1
Float, Double	False = 0.0, True = 1.0
String, Big String	“false” or “true”

### Coercion from Integer Values

To ...	Values / Notes
Boolean	0 is False, non-zero is True.
Float, Double	Value is preserved as is.
String, Big String	Value is formatted as string.
Elapsed Time	Interpreted as number of seconds.
Enumeration types	Interpreted as ordinal value of enumeration.

### Coercion from Float or Double Values

To ...	Values / Notes
Boolean	0.0 is False, non-zero is True.
Double (from Float)	Value is preserved as is.
Float (from Double)	Values less than the minimum absolute Float value of 1.17549E-38 result in a value of Float 0.0, i. e. precision may be lost.
Integer	Value is rounded.
String, Big String	Value is formatted as string.
Elapsed Time	Value is interpreted as number of seconds and rounded.
Enumeration types	Interpreted as ordinal value of enumeration.

## Coercion from String or Big String Values

To ...	Values / Notes
Boolean	“False” (any case) is False, “True” (any case) is True. All other values result in an error.
Float, Double	String must use the following format: [whitespace][sign][digits][.digits] [d D e E][sign][digits]. Precision may be lost.  A string of “NaN” (any case) results in an IEEE NaN value.
Integer	String must represent a valid signed or unsigned Integer.
Elapsed Time	String must use the following format: [-[DDDDDD.] [HH:MM:]SS[.ffffff]], where DDDDDD is from 0 to 999999, HH is from 0 to 23, MM is from 0 to 59, SS is from 0 to 59, fffffff is fractional seconds (one through seven digits). Parts in brackets are optional.
Time	String must use correct date/time syntax for the current locale.
Data Type	String must be a valid enumeration label (“MxInteger”, “MxFloat”, etc.)
Custom Enumeration	Interpreted as string part of enumeration. No checking is done to determine if the string is valid.
Reference	String is set as reference string. No syntax checking is done.

## Coercion from Time Values

To ...	Values / Notes
String, Big String	Value is formatted according to the time format specified by the current locale. “Zero time” (1/1/1601, 00:00:00) results in a blank string.

### Coercion from Elapsed Time Values

To ...	Values / Notes
Integer	Converted to number of seconds.
Float, Double	Converted to number of seconds.
String, Big String	String uses the following format: [-]DDDDDD HH:MM:SS.ffffff

### Coercion from MxStatus Values

To ...	Values / Notes
String, Big String	String is generated from the Category and Detail information of the MxStatus value. If Category is OK, the string is empty.

### Coercion from Data Type Values

To ...	Values / Notes
Integer, Float, Double	Converted to ordinal value.
String, Big String	Converted to type label, e. g. "MxInteger"
Elapsed Time	Value is interpreted as number of seconds and rounded.
Enumeration types	Interpreted as ordinal value of enumeration.

### Coercion from Custom Enumeration Values

To ...	Values / Notes
Integer, Float, Double	Converted to ordinal value.
String, Big String	Converted to string value, e. g. "MyEnum1"



## Coercion from Custom Structure Values

To ...	Values / Notes
String, Big String	Qualifier (GUID) is converted to string.

## Using Data Types Correctly

Follow these guidelines on using specific data types.

### Custom Enumeration vs. Integer

When defining an attribute that contains enumeration values, use the Custom Enumeration data type, not Integer.

There are two criteria to distinguish an enumeration attribute from a simple Integer attribute:

- In an enumeration, each possible value has a specific meaning and represents a mode, state, etc.
- The values of an enumeration cannot be meaningfully compared using comparison operators (>, <, <=, >=).

For each Custom Enumeration attribute, you must define a second attribute containing an array of strings that defines the possible enumeration values. Often, these values shouldn't be changed by the end user. If this is the case, set the array attribute's category to Constant so that users can't modify it.

### Absolute and Elapsed Times

Use the Elapsed Time data type for storing an “elapsed time,” that is, an amount of time. Avoid using Integer or Float attributes with associated units of measure (like seconds, minutes...) for this purpose. The only exception is very short amounts of time expressed in milliseconds. These can be stored as an Integer value.

Elapsed Time and Time attributes have a standard string representation. The ArchestrA framework automatically converts them to and from strings. They do not require an associated engineering unit.

## Internationalized String

Use the Internationalized String data type to define string attributes that contain translations of a string for multiple target languages. A good example would be attributes containing an Engineering Unit name.

Use the Object Designer to define the default value for US English.

At config time, you can use the methods of the `CMxInternationalizedString` class to get and set the string values for each locale. For more information, see the *Archedra Object Toolkit Reference Guide*. For example:

```
//Get a string value using locale 1033
string temp = Attribute1.GetString(1033);
//Write a string value using locale 1033
Attribute1.SetString(1033, "MyString");
//Create a local copy of an Internationalized String
InternationalizedString[] temp =
    Attribute1.GetInternationalizedStrings();
```

You can't modify `InternationalizedString` attributes at run time.

## Big String

The Big String data type is designed to let you create reasonably large strings beyond the 1,024 character limitation of the String data type. Theoretically, it allows you to create strings up to 2,147,483,647 characters in length. In practice, available system memory and system performance impose much lower limits. A reasonable practical maximum for a single attribute of this type would be around 10 MB.

## Attribute References

Use the Attribute Reference data type to store the fully qualified name of an attribute. For example, the Input and Output primitives use attributes of this type to hold the input source or output destination.

You can read and write attribute references as strings. However, they are different from strings. They also include binding (location) information to improve the startup performance of the messaging system. Therefore, don't use simple strings to store reference information.

You can use two default values for an attribute reference to indicate that the reference is unspecified:

- Use “---.---” to indicate that the user must specify a reference. If the user doesn’t specify a reference, this default value causes a warning when the object’s configuration is validated.
- Use “---” to indicate that the reference is optional. This default value does not cause a warning when the object’s configuration is validated.

You will rarely need to create attributes of this data type. In most cases, you will simply use the existing Input and Output primitives for communicating with other objects.

## Variant (Unspecified) Data Type

Use the Variant data type if the actual type of an attribute cannot be determined in advance. You can then include custom config time code that lets the user select the actual data type while configuring the object. The data type can also be modified when the object starts up at runtime.

An example of this is the Input primitive. This primitive gets data from another object, regardless of its type, and stores the value in its “Value” attribute. The data type of the value attribute therefore depends on the type of data that the input primitive is being used to retrieve.

In certain cases, it may be helpful to configure and persist the datatype itself (Boolean, Float, String, etc.) as an attribute. To do this, there is a special data type called “Data Type.” For example, the Input and Output primitives include an attribute of this type to configure the desired type of their value attribute.

## Arrays

You can configure attributes of any data type as arrays. When defining your object in the Object Designer, you can specify the initial number of elements and the initial value of each element.

The size of an array can be changed at config time or run time. If you want to fix the size of an array, you must create config time and run time set handlers that ensure that the size of the array is not changed.



---

# Appendix E

## ArchestrA Attribute Categories

An attribute's category determines which namespaces an attribute appears in, whether the attribute can be written to, and what type of client (users, scripts, etc.) can write to it.

You should only allow as much access to an attribute as necessary. For example:

- An attribute that is used only by config time logic to add or remove Virtual Primitives should be given a category that prevents it from becoming part of the run time namespace (e. g. `PackageOnly`).
- An attribute that doesn't make sense to configure in an editor should be given a category (e. g. `Calculated`) that prevents it from becoming part of the config time namespace (and the custom object editor).
- An attribute that needs to be configured and deployed but is not allowed to be modified at runtime should be given a category that prevents users or other objects from writing to it (e. g. `Writeable_C_Lockable`).

Some attribute categories allow an attribute to be *locked*. This means that IDE users can lock the attribute in a template to protect its configured value from being changed in derived instances or templates. The value of a locked attribute cannot be modified, not even internally by the object's code.

In general, you should make attributes lockable whenever possible. Lockable attributes allow users to enforce standards and can simplify system maintenance. Locking data also helps minimize the size of a system's configuration database, which improves the speed of configuration tasks.

**Note** An exception is raised if an object's logic attempts to modify a locked attribute. Therefore, before modifying a lockable attribute in your code, check whether it is locked.

The following table describes each attribute category.

Category Name	Description
PackageOnly	Only exists at config time. Not deployed.
PackageOnly_Lockable	Only exists at config time. Not deployed. Can be locked.
Calculated	Only exists at run time. Not externally writeable by users or other objects. Run time changes are not persisted to disk by the AppEngine.
Calculated_Retentive	Only exists at run time. Not externally writeable by users or other objects. Run time changes are persisted to disk by the AppEngine.
Constant	Defined by an object developer. Never changes. Exists at config time and run time.
Writeable_U	Exists at config time and run time, but only the Security Classification is configurable. Only externally writeable by users at run time.
Writeable_S	Only exists at run time. Only externally writeable by other objects at run time.
Writeable_US	Exists at config time and run time, but only Security Classification is configurable. Externally writeable by users or other objects at run time.
Writeable_UC	Exists at config time and run time. Only externally writeable by users at run time.
Writeable_UC_Lockable	Exists at config time and run time. Only externally writeable by users at run time. Can be locked.
Writeable_USC	Exists at config time and run time. Externally writeable by users or other objects at run time.
Writeable_USC_Lockable	Exists at config time and run time. Externally writeable by users or other objects. Can be locked.
Writeable_C_Lockable	Exists at config time and run time. Not writeable at run time, even by the object itself. Can be locked.

An attribute's category also determines whether the attribute supports various other options, such as a default value or historization. See the following table for details.

Attribute Category	Can set security classification	Can have config time set handler	Can have run time set handler	Can be marked as "Frequently Accessed"	Supports "Calculated Quality and Time"	Default value can be set	Supports alarms and history <sup>1</sup>	Supports "Advise Only Active"
PackageOnly	N	Y	N	N	N	Y	N	N
PackageOnly_Lockable	N	Y	N	N	N	Y	N	N
Constant	N	N	N	Y	N	Y	N	N
Writeable_C_Lockable	N	Y	N	Y	N	Y	N	N
Writeable_UC	Y	Y	Y	Y	Y	Y	Y	Y
Writeable_UC_Lockable	Y	Y	Y	Y	Y	Y	Y	Y
Writeable_USC	Y	Y	Y	Y	Y	Y	Y	Y
Writeable_USC_Lockable	Y	Y	Y	Y	Y	Y	Y	Y
Calculated	N	N	Y	Y	Y	N	Y	Y
Calculated_Retentive	N	N	Y	Y	Y	N	Y	Y
Writeable_S	N	N	Y	Y	Y	N	Y	Y
Writeable_U	Y	N	Y	Y	Y	N	Y	Y
Writeable_US	Y	N	Y	Y	Y	N	Y	Y
SystemInternal	N	N	Y	N	N	Y	N	N
SystemSetsOnly	N	N	Y	N	N	N	N	N
SystemWriteable	N	N	Y	N	Y	Y	Y	Y

<sup>1</sup> Only non-array Boolean attributes can be alarmed. Only non-array attributes of the following types can be historized: Double, Float, Integer, Boolean, String, Custom Enumeration, and ElapsedTime,





# Appendix F

## ArchestrA Security Classifications

By default, new attributes are created with the “Free Access” security classification, which means that any user can write to them. You can restrict write access to an attribute by selecting a different security classification. For example, you can specify that the user must have a certain permission in order to write to the attribute, or that the write operation must be verified by a second user.

---

**Important** Security classifications are only effective if security is enabled in the Galaxy.

---

The ArchestrA infrastructure supports the following security classifications:

---

Security Classification	Description
FreeAccess	Any user can write to these attributes. Use this classification for attributes that trigger safety or time critical tasks that could be hampered by an untimely logon request. For example, halting a failing process.
Operate	Users need Operate permissions to write to these attributes.  Use this classification for attributes that operators write to during normal day-to-day operations.

---

Security Classification	Description
SecuredWrite	<p>When writing to these attributes, users must re-enter their logon information. The new value is only written if the logon information is correct and the user has Operate permissions for the attribute.</p> <p>Use this classification for attributes that operators write to during normal day-to-day operations, but that require an extra level of security.</p>
VerifiedWrite	<p>When writing to these attributes, users must re-enter their logon information, and another user must confirm the write by entering his or her logon information as well. The new value is only written if the two users are different, the logon information for both users is correct, and both users have Operate permissions for the attribute.</p> <p>Use this classification for attributes that require very tight security and whose values should not be changed based on the decision of one person alone.</p>
Tune	<p>Users need Tune permissions to write to these attributes.</p> <p>Use this classification if an attribute is a configuration parameter that might be tuned by an engineer during normal system operations. For example, an alarm setpoint, PID sensitivity, etc.</p>
Configure	<p>Users need Configure permissions to write to these attributes, and the object must be OffScan for the write to succeed.</p> <p>Use this classification if a change to the attribute would be considered a significant configuration change. For example, the I/O addresses of an object.</p>
ReadOnly	<p>These attributes can not be written to at run time at all, regardless of the user's permissions.</p>

# Index

## A

- aaDEF file
  - build output 118
  - importing 67
- aaPDF file 109
  - build output 118
- aaPRI file 70
  - build output 118
- acknowledgement (alarms) 87
- add-in, Visual Studio 19
- Advise Only Active
  - at run time 143
  - enabling for attributes 101
  - enabling on object level 60
- alarm 145
- alarms 138
  - acknowledgement 87
  - alarm primitive attribute list 87
  - alarming attributes 83
  - category 84, 88
  - inhibition 88
- ApplicationObjects
  - assemblies, configuring 64
  - building 117
  - creating (workflow) 17
  - debugging
    - current version 124
    - new version 125
  - defining 35
  - definition 14
  - description, configuring 36
  - design guidelines 25
  - dictionary 106
    - editing 107
    - retrieving strings 108
    - structure 107
  - differences to editing reusable primitives 70
  - event handlers, configuring 37
  - execution group, configuring 63
  - external name 36
  - help, adding 65
  - IDE behavior, configuring 61
  - internal name 36
  - internationalization 105
  - migrating 119
  - minimum Application Server version, configuring 60
  - names, configuring 36
  - shape 16, 67, 114
  - switching to reusable primitive mode 69
  - toolset, configuring 62
  - validating 110
  - vendor name 36
- ArchestrA attributes 152
- ArchestrA UI controls

- adding to Visual Studio 48
- changing reference 49
- array attributes 132
- array length 154
- arrays
  - changing or enforcing length 162
  - defining 73
  - usage notes 187
- assemblies
  - for ApplicationObjects, configuring 64
  - for local primitives, configuring 41
- associated files 49
  - configuring manually 55
  - setting up rules 50
- attaching debugger
  - to current version 124
  - to new version 125
- Attribute Reference (data type)
  - usage notes 186
- attribute wrappers 130
- attributes
  - adding to an ApplicationObject or primitive 72
  - Advise Only Active, enabling 101
  - alarming 83
    - alarm primitive attribute list 87
  - arrays, configuring 73
  - calculated quality and time 73
  - category
    - list 189
    - setting 73
  - config time set handlers, configuring 74
  - configuring 71
  - data type
    - changing at config time 164
    - coercion rules 181
    - list of data types 179
    - setting 73
    - usage guidelines 185
  - default attribute, creating 74
  - default value, setting 73
  - defining optional attributes via primitives 45
  - definition 14
  - deleting 104
  - dynamic. See "dynamic attributes"
  - extensions, configuring 79
  - external name 72, 104
  - hidden attribute, creating 74

- historizing 79
  - history primitive attribute list 81
- internal name 72, 104
- logical hierarchy, creating 161
- naming conventions 157
- naming restrictions 158
- overriding in reusable primitives 43
- planning usage 28
- renaming 104
- run time set handlers, configuring 76
- security classification 73
  - list 193
- set handlers
  - config time, configuring 74
  - run time, configuring 76

## B

- Big String (data type)
  - coercion rules 183
  - usage notes 186
- Boolean (data type)
  - coercion rules 181
- build modes 117
- build options
  - Galaxy preferences 112
  - output preferences 111
  - search paths 113
- build process 150
- building an ApplicationObject or reusable primitive 117

## C

- calculated quality
  - enabling for attribute 73
- calculated time
  - best practices 168
  - enabling for attribute 73
- calculations 138
- category
  - of an alarm 84, 88
  - of an attribute
    - list 189
    - setting 73
- checkpointing support for dynamic attributes 170
- child primitive 149
- CLSIDs, configuring 64
- coercion rules for data types 181

- Common primitive 15
- config time coding 131
- config time event handlers 37
- config time set handlers, configuring for attributes 74
- Configtime (subfolder) 33
- configuration status, determining 164
- Custom Enumeration (data type)
  - coercion rules 184
  - usage notes 185
- Custom Structure (data type)
  - coercion rules 185

## D

- Data Change events 162
- data type
  - coercion rules 181
  - list 179
  - setting for attribute 73
  - usage guidelines 185
- Data Type (data type)
  - coercion rules 184
- debugging
  - current version of an object 124
  - new version of an object 125
- default attribute, creating 74
- Dependent File Manager 50
- dependent files. See "associated files"
- description
  - alarm 85, 88
  - setting for historized attribute 80, 81
- design guidelines 25
- determining configuration status 164
- dictionary 106
  - editing 107
  - retrieving strings 108
  - structure 107
- documentation conventions 11
- Double (data type)
  - coercion rules 182
- dump/load support
  - for dynamic attributes and virtual primitives 58
  - guidelines for config time code 163
- dynamic attribute 136, 143
- dynamic attributes
  - checkpointing support 170

- dump/load support 58
- failover support 59, 170
- set handler, configuring 78

## E

- editor
  - adding to object 47
  - guidelines 171
- Editor (subfolder) 33
- Elapsed Time (data type)
  - coercion rules 184
  - usage notes 185
- engineering units
  - alarms 85, 87
  - historization 80, 81
- error status 139
- event handlers 37
  - config time 37
  - run time 39
- events, raising 162
- Execute (event handler) 39
- execution group
  - ApplicationObjects 63
  - local primitives 41
- extensions (attributes) 79
- external attributes 145
  - referencing 147
- external name
  - ApplicationObjects 36
  - attributes 72, 104
  - configuring in code 129
  - empty (for primitives) 161
  - general conventions 157
  - Input/Outputs 96
  - Inputs 90
  - Outputs 93
  - primitives 46
  - restrictions 158

## F

- failover support for dynamic attributes 59, 170
- feature overview 13
- Float (data type)
  - coercion rules 182
- forced storage period (history option) 80, 82

## G

Galaxy preferences, configuring 112  
GetStatusDesc (event handler) 39  
getting inputs 138

## H

help, adding to ApplicationObject 65  
hidden attributes, creating 74  
historizing attributes 79  
    history primitive attribute list 81

## I

IDE behavior, configuring 61  
importing aaDEF files 67  
inhibition (alarms) 88  
Initialize (event handler) 39  
input value 141  
Input/Outputs  
    adding 95  
    external name 96  
    Input/Output primitive attributes 97  
    internal name 96  
    references 97  
Inputs  
    adding 90  
    external name 90  
    Input primitive attributes 91  
    internal name 91  
    source reference 91  
Integer (data type)  
    coercion rules 182  
internal name  
    ApplicationObjects 36  
    attributes 72, 104  
    configuring in code 129  
    Input/Outputs 96  
    inputs 91  
    Outputs 93  
    primitives 46  
    restrictions 158  
internationalization 105  
Internationalized String (data type)  
    usage notes 186  
interpolation type (history option) 80, 82  
Intialize (event handler) 37

## L

limit (alarm setting) 85, 87  
limitations  
    complexity 27  
    performance 29  
local attribute wrapper 155  
local primitives  
    adding 40  
    assemblies, configuring 41  
    definition 15  
    execution group 41  
    limitations to complexity 27  
    subfolders in project 33  
local reference 149  
Logger  
    usage guidelines 162  
Logger View 21  
    hiding/showing 21

## M

major version 115  
Migrate (event handler) 37  
migrating ApplicationObjects 119  
minimum Application Server version,  
    configuring 60  
minor version 114  
Mode list (build modes) 117  
Monitor (sample object) 173  
MxStatus (data type)  
    coercion rules 184

## N

naming  
    conventions 157  
    restrictions 158  
non-array attributes 132

## O

Object Design View 20  
    hiding/showing 21  
    refreshing 21  
Object Designer 22  
    opening 22  
    panes 23  
    synchronization with code 22  
object dictionary 106  
    editing 107  
    retrieving strings 108

- structure 107
- object editor (custom) 47
- object help, adding 65
- object shape 16, 67, 114
- objects. See ApplicationObjects
- Output (subfolder) 33
- output preferences, configuring 111
- output value 141

## Outputs

- adding 92
- destination reference 94
- external name 93
- guidelines for object startup 169
- internal name 93
- Output primitive attributes 94
- overriding attributes in reusable primitives 43
- overscans, avoiding 165

## P

- packaging 151
- parent primitive 149
- performance considerations 29
- polling 143
- PostCreate (event handler) 38
- PreValidate (event handler) 38
- primitives
  - "arrays" using virtual primitives 45
  - adding 40
  - Common 15
  - deleting 43
  - execution group, configuring 63
  - external name 46
  - Inputs and Outputs 89
  - internal name 46
  - limitations to complexity 27
  - local. See local primitives
  - naming considerations 46
  - naming conventions 157
  - naming restrictions 158
  - reusable. See reusable primitives
  - virtual. See virtual primitives
- priority (alarm setting) 85, 89
- programming techniques 127
- projects
  - ArchestrA Object Toolkit 31
  - creating 32
  - deleting 34

- editing 34
- folder contents 33
- moving 34
- opening 33
- overview 31
- synchronization between code and Object Designer 34
- Visual Studio 31

## Q

- quality 140
  - calculated 73
- quarantine state 170

## R

- rate deadband (history option) 80, 82
- references
  - Input source 91
  - Input/Output 97
  - Output destination 94
- relative string reference 148
- renaming attributes 104
- retrieving localized strings 108
- reusable primitives
  - adding 41
  - creating (workflow) 17
  - defining 69
  - definition 15
  - design guidelines 25
  - differences to editing
    - ApplicationObjects 70
  - event handlers 43
  - execution group, configuring 63
  - overriding attributes 43
  - switching to ApplicationObject mode 69
- rollover value (history option) 80, 82
- rules for associated files 50
- run time coding 137
- run time event handlers 39
- run time set handlers, configuring for attributes 76
- Runtime (subfolder) 33

## S

- sample count (history option) 80, 82
- sample objects
  - Monitor 173

- WatchDog 175
- search paths, configuring 113
- security classification
  - list 193
  - setting for attribute 73
- set handler code 138
- set handlers
  - config time, configuring for attributes 74
  - for dynamic attributes, configuring 78
  - run time, configuring for attributes 76
- SetScanState (event handler) 39
- setting outputs 138
- shape of an object 16, 67, 114
- Shutdown (event handler) 39
- solution folder 31
  - contents 33
- Startup (event handler) 39
- static attributes 136, 142
- static primitives 136, 142
- String (data type)
  - coercion rules 183
- swinging door (history option) 80, 81
- switching between object/primitive mode 69

## T

- technical support, contacting 12
- time
  - best practices 168
  - calculated 73
- Time (data type)
  - coercion rules 183
  - usage notes 185
- time stamp 140
- toolbar 19
- toolset
  - configuring for ApplicationObject 62
  - names, setting up 62
- trend scale (history option) 80, 82

## U

- user interface

- additions to Visual Studio 19
- Logger view 21
- Object Design View 20
- Object Designer 22
- overview 18
- toolbar 19

## V

- Validate (event handler) 38
- validating an object 110
- validation 132
- value deadband (history option) 80, 83
- Variant (data type)
  - usage notes 187
- vendor name 36
- versioning 113
- versions
  - major 115
  - minor 114
  - of an object, managing 113
  - specifying manually 117
- virtual primitives
  - adding 133
  - and object design 26
  - definition 15
  - deleting 134
  - dump/load support 58
  - overview 43
- Visual Studio
  - adding ArchestrA controls to toolbox 48
  - additions to interface 19
  - projects 31
  - references, configuring as associated files 50
  - solution folder 31

## W

- WatchDog (sample object) 175
- workflow 128
  - creating an ApplicationObject or reusable primitive 17