DotNET Connect User's Guide

VisualWorks 7.9

P46-0144-05

SIMPLIFICATION THROUGH INNOVATION

Copyright © 2003-2012 Cincom Systems, Inc.

All rights reserved.

This product contains copyrighted third-party software.

Part Number: P46-0144-05

Software Release 7.9

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

CINCOM, CINCOM SYSTEMS, and the Cincom logo are registered trademarks of Cincom Systems, Inc. ParcPlace and VisualWorks are trademarks of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. ObjectLens, ObjectSupport, ParcPlace Smalltalk, Database Connect, DLL & C Connect, COM Connect, and StORE are trademarks of Cincom Systems, Inc., its subsidiaries, or successors. ENVY is a registered trademark of Object Technology International, Inc. GemStone is a registered trademark of GemStone Systems, Inc. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 2003-2012 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

About This Book

Overview

This document describes the VisualWorks DotNETConnect package, which allows accessing Microsoft® .NET assemblies from a VisualWorks application.

Audience

This document is intended for new and experienced developers to quickly become productive developing applications in VisualWorks accessing .NET assemblies.

In general, it is assumed that you have some understanding of .NET technologies and terminology, and have access to specific information about the assemblies you will be using. It is also assumed that you have a basic understanding of working in the Microsoft Visual Studio development environment.

It is assumed that you have a beginning knowledge of programming in a Smalltalk environment, though not necessarily with VisualWorks. .

Organization

This document is organized into five chapters.

Chapter 1, “Introduction.” This chapter provides a general overview .NET technology and the DotNETConnect facilities for invoking .NET assemblies from VisualWorks.

Chapter 2, “Adapting an Assembly.”. This chapter describes in more detail the tools and processes necessary for creating the proxy code and files necessary to access those assemblies from VisualWorks.

Chapter 3, “Using an Assembly in VisualWorks.” This chapter provides examples and guidelines for invoking .NET assembly functions from VisualWorks.

Chapter 4, “Debugging.” This chapter provides instructions for creating a debuggable DLL, and for connecting VisualWorks and Visual Studio process to debug a method invoking a .NET assembly.

Chapter 5, “Deploying a DotNET Application.” This chapter provides a few notes on deploying a DotNETConnect application.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
cover.doc	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.

Examples	Description
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
`Integer>>asCharacter`	Indicates an instance method defined in a class.
`Float class>>pi`	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to helpna@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help > About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help > About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to helpna@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe". You can then address emails to vwnc@cs.uiuc.edu.

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://www.cincomsmalltalk.com/CincomSmalltalkWiki

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Contents

About This Book iii

Overview ...iii
Conventions ... iv
Getting Help ... v
Additional Sources of Information ...vii

Chapter 1 Introduction 1-1

Overview ...1-1
What is .NET? ...1-1
What is DotNETConnect ...1-2
How does it work ...1-2
Getting Started ...1-3
Prerequisites ...1-3
Installation ...1-3
Components of DotNETConnect ...1-5
VisualWorks components ...1-5
DLLs ...1-6
DotNETConnect Tools ...1-7
Code Generator ...1-7
Global Assembly Cache Browser ...1-8
Developing with DotNETConnect ...1-9

Chapter 2 Adapting an Assembly 2-1

Creating Adaptors Using the Wizard ...2-1
Choose the Compiler ...2-3
Change Settings ...2-4
Check Types ...2-7
Review Kept/Deleted ...2-7
Generate C++ and Smalltalk Code ...2-9
Install C++ and Smalltalk Code ...2-11

Creating Adaptors Programmatically 2-12
Read the Assembly Members 2-13
Checking the assembly 2-13
Generate the Code 2-14
Compiling the Proxy Code 2-14
Eliminating Compiling Errors 2-15
Remove a Member Programmatically 2-16
Installing the Smalltalk Stubs 2-18
Resolving Method Signatures 2-18
Importing MSCorLib 2-20

Chapter 3 Using an Assembly in VisualWorks 3-1

Using a .NET Assembly Proxy 3-1
Supported Types 3-1
Base Types 3-2
Enumerations 3-2
Reference Types 3-3
Creating instances 3-4
Generic Types 3-5
Converting between .NET and Smalltalk types 3-6
Argument types 3-6
Using .NET Events and Delegates in VisualWorks 3-8
Delegates 3-8

Chapter 4 Debugging 4-1

Overview 4-1
Debugging Errors While Loading the Proxy 4-2
Create a Debug DLL 4-2
Connect VisualWorks and Visual Studio 4-3
Debugging a Proxy Method 4-3

Chapter 5 Deploying a DotNET Application 5-1

Index Index-1

1

Introduction

Overview

DotNETConnect is a VisualWorks add-on that allows you to access and use Microsoft® .NET components in a VisualWorks application. This additional capability makes it easier for VisualWorks programmers to develop applications that coordinate well with other .NET applications.

DotNETConnect was developed by Georg Heeg eK, and is provided with VisualWorks under license.

What is .NET?

.NET is a set of Microsoft technologies for developing and deploying web services. The technologies consist of a development tools and servers designed specifically to facilitate implementing web service solutions.

At the core of the development and deployment environment is Visual Studio .NET and the .NET Framework. Visual Studio .NET is Microsoft's suite of programming tools, redesigned to coordinate and interact through the .NET Framework. All .NET programming tools operate within the requirements of the .NET Framework.

The .NET Framework consists of two components: the common language runtime (CLR) and a large class library. The library is built on the common type system (CTS), according to which all objects are decendents of the root class `System.Object`. The CLR is a runtime environment for applications built according to the requirements of the .NET Framework.

What is DotNETConnect

DotNETConnect is a framework for integrating .NET components into VisualWorks applications, while maintaining full compliance with both environments. DotNETConnect allows objects in VisualWorks to access object inside the .NET CLR as if they were VisualWorks objects. Unlike other existing connection techniques (web services, COM), there is no additional programming necessary on the side of the .NET classes.

DotNETConnect is the ideal solution for the Smalltalk programmer who needs to access .NET libraries from a Smalltalk application.

How does it work

DotNETConnect enables VisualWorks objects to interact with .NET objects through a set of stubs and proxies, which are methods and classes in VisualWorks. The VisualWorks programmer completes the stub methods as necessary by doing standard Smalltalk programming. The stubs and proxies are connected to .NET using a standard C DLL through DLL and C Connect.

DotNETConnect supports lifecycle management using a combined garbage collection.

Assemblies are the unit of deployment and reuse in .NET, consisting of one or more physical files, at least one of which is either an EXE or a DLL, presented as a single component with a well-defined API. As such, an assembly contains definitions of classes (types) and their methods and properties (members) collected into a single implementation unit. An assembly represents a kind of “black box” functionality, where the signatures of the contained classes and methods are visible to the developer, but not its implementation. The .NET CLR searches for assemblies in several places:

- The global assembly cache
- The application configuration file
- The local directory
- The manifest in the first loaded assembly

.NET includes a reflection API to query the signatures of all public classes and their members in an assembly. DotNETConnect uses this API to query and collect the signatures of the public classes and

their members. Based on the signatures, DotNETConnect automatically creates the source code for .NET proxies and VisualWorks stubs.

DotNETConnect generates a mixed DLL (plain C connecting to VisualWorks and managed C++ connecting to the .NET CLR) that does not have a strong name and cannot be installed in the global assembly cache. Copying all necessary DLLs into the local directory containing the VisualWorks virtual machine is the approach taken.

Getting Started

This section describes the system requirements for using DotNETConnect, how to install it, and a simple example.

Prerequisites

The software required to use DotNETConnect in a runtime environment are:

- Microsoft .NET Framework 2.0 or higher
- VisualWorks 7.3 or higher

The prerequisites for developing applications using DotNETConnect are:

- One of these Microsoft development environments:
  - Microsoft Visual Studio 2005 (Enterprise or Professional Edition)
  - Microsoft Visual Studio 2008 (any edition)
  - Microsoft Visual Studio 2010 (any edition)
- VisualWorks 7.3 or higher

The standard edition of Visual Studio 2005 .NET does not support mixed mode DLLs, which DotNETConnect requires for development purposes.

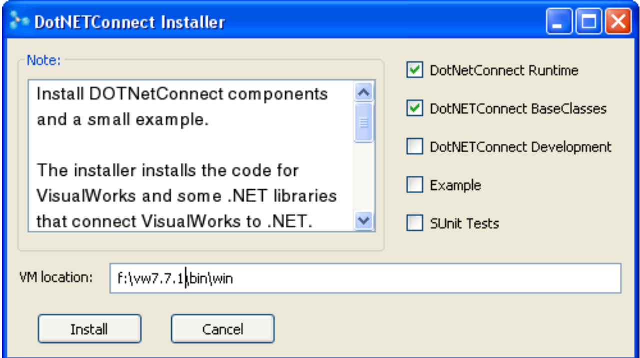
Installation

Installing DotNETConnect creates an environment in which VisualWorks has access to .NET libraries. It copies all necessary files to their appropriate locations and installs the parcels for the options selected.

We recommend creating a dedicated project directory, for example, **c:\development\projects\dotnet**, as the target of generated code for imported assemblies. We will use this path in examples.

Copy a clean image into this directory and start it.

Using the Parcel Manager, load the DotNETConnectInstaller parcel, in the **OS-Windows** suggestions folder or the **DotNETConnect** directory. Loading the parcel opens an installation dialog.



Select the components to install, and confirm the directory where the virtual machine is located.

The first two components, **DotNETConnect Runtime** and **DotNETConnect Base Classes**, are required for both runtime and development installations. **DotNETConnect Development** adds development features, including tools for browsing assemblies and for generating the source code for .NET and VisualWorks. The **Example** option loads the Aladin DiveReader example, which is referred to later in this document to illustrate several operations.

The **SUnitTests** option loads SUnit tests which test the DotNETConnect Runtime.

The installer also copies several DotNETConnect DLLs into the virtual machine directory to enable the class loader inside the CLR to locate them.

Click **Install** to install the selected components.

DotNETConnect installs DLLs, some base source files for the .NET proxies, and two (deployment) or four (development) VisualWorks components.

For Microsoft Visual Studio 2005 you need **reshacker.exe**. The installer copies it from **packaging\win\reshack.zip** into the sources-DLL directory.

Components of DotNETConnect

The following paragraphs describe the components that are installed as part of the installation process.

VisualWorks components

The following components are loaded as bundles. Otherwise, they are loaded as collections of parcels with related, but sometimes different, names.

DotNETConnectRuntime

This bundle contains the base components needed for a project containing .NET-components. .NET objects are made accessible through stubs that are subclasses of DotNETObject. Automatic type conversion and the garbage collection of .NET components are handled in this bundle.

DotNETBaseClasses

This bundle contains stubs and wrappers for some selected classes of the .NET base assembly mscorlib. These classes (such as classes representing numbers, collections, I/O, etc.) are often needed in other projects. It connects to the mscorlib assembly using MscorlibProxy.DLL.

DotNETConnectDevelopment

This bundle contains the components necessary to make .NET assemblies available in VisualWorks. This includes the components for retrieving the assembly signatures, and tools for generating the proxy DLLs and the Smalltalk stubs.

Aladin DiveReader

Installing the example loads this additional component into the image. It contains the generated stubs for accessing the example assembly and a small VisualWorks application that uses the generated code.

Event Examples

Two packages demonstrate event usage:

- Simple Timer consists of the .NET class Clock, which

produces events, and the Smalltalk class Watcher, which consumes them.

- Simple Panel opens a window with a .NET form and processes its events in VisualWorks.

SUnit Tests

Tests all types in all possible situations (e.g., as parameters, results, as properties, in arrays, etc.).

DLLs

The following DLLs are installed in the directory containing the VisualWorks virtual machine.

DotNETBase.DLL

The DLL contains the DotNETMarshaller that is used to pass return values back to VisualWorks. It contains a registry that keeps objects in .NET from being garbage collected as long as they are referenced from VisualWorks stubs.

ReflectionProxy.DLL

The DLL contains the code that enables VisualWorks to access the reflection API of .Net.

MscorlibProxy.DLL

The DLL contains the proxy for selected classes from the .NET base assembly mscorlib. The stubs that use this proxy are organized in the bundle DotNETBaseClasses.

DotNETConnectTest.DLL and DotNETConnectTestProxy.DLL,

DotNETTests.DLL and DotNETTestsProxy.DLL

These DLLs are used by some of the SUnit tests.

Aladin.DLL and AladinProxy.DLL

The example adds these two DLLs to the VM directory. One DLL is the example assembly itself. The other DLL is the proxy for it.

EventClock.dll, EventClock.dll, EventExample_SystemProxy.dll and EventExample_SystemWindowsFormsProxy.dll

DLLs used by the event example parcel.

DotNETConnect Tools

In addition to the components described above, DotNETConnect includes two tools to assist in adapting .NET assemblies for use within VisualWorks: the Global Assembly Cache Browser, and the Code Generator.

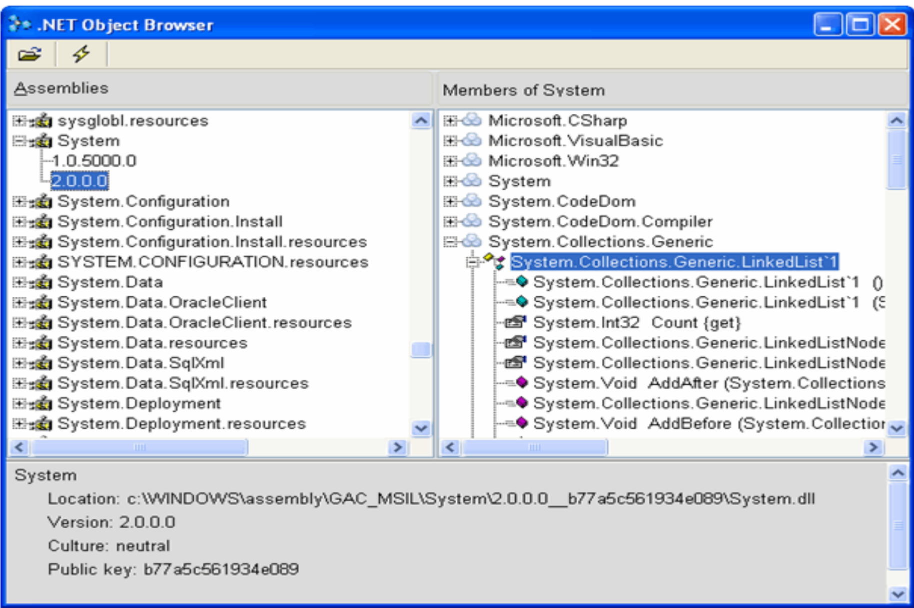
Code Generator

The DotNETConnect Code Generator is a wizard that walks you through the steps of generating the necessary C++ and Smalltalk code for adapting a .NET assembly. It also simplifies certain of the steps. Use of the Code Generator is described in detail in the next chapter, Adapting an Assembly.

To launch the Code Generator, select **Tools > DotNETConnect Code Generator** in the main VisualWorks window. Alternatively, select the assembly to import in the Global Assembly Cache Browser and click the “lightening” (launch wizard) button.

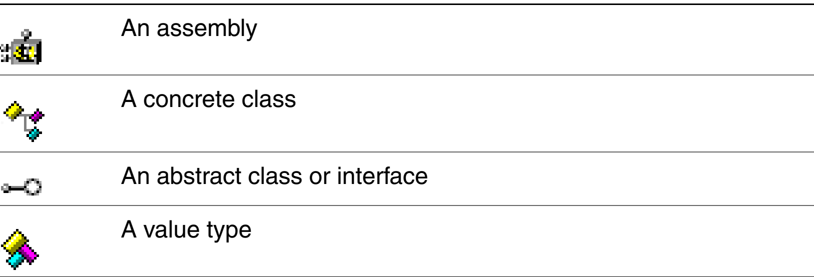
Global Assembly Cache Browser

DotNETConnect includes this browser for exploring the global assembly cache, a special directory for shared assemblies. To open the browser, select **Tools > DotNETConnect Browser** in the VisualWorks main window.



The left list pane shows all of the assemblies and their versions installed in the .NET global assembly cache. The right list pane shows the types (classes, interfaces and value types) in an assembly and their members (methods) grouped by name space. The text pane shows the signature of the selected type or member.

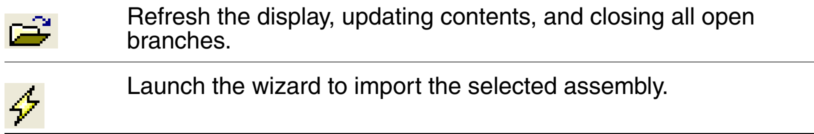
The icons in the list panes describe the type of the item:



	An assembly
	A concrete class
	An abstract class or interface
	A value type

	A constructor
	A property
	A method
	A name space

The toolbar provides two buttons:



	Refresh the display, updating contents, and closing all open branches.
	Launch the wizard to import the selected assembly.

This is a display-only browser that allows you to examine the contents of the global assemblies cache. Use the Launch button to open the Import Wizard on the selected assembly.

Developing with DotNETConnect

Developing an application to use .NET components is simple, once the assemblies have been adapted for use in VisualWorks. In this section we illustrate a very simple application using a .NET class and methods.

Most of the real work is in adapting an .NET assembly, prior to developing the application. This is described in detail in the next chapter, Adapting an Assembly

For this example we use only features that are available in the mscorlib assembly, which is already adapted and ready for use in DotNETConnect. In particular, we will replace the use of the VisualWorks `Random` class with the mscorlib `Random` class.

Using the Parcel Manager, load the WalkThru example (in the **Directories** view, in the **examples** directory). Find the parcel/package in a System Browser, and browse the RandomNumberPicker class. There are three methods that invoke the Random class or its methods: initialize, nextRandom, and resetSequences. We'll look at each of these.

```
initialize
"Create the initial random number generator for this application"
myRandom := Random new.
```

In this method the instance creator new is invoked, and might be different for the corresponding mscorlib instance creator, as might be the class name itself (though it is not). For methods, even if the names are the same, the naming convention in .NET is to begin a method name with an uppercase letter, contrary to the Smalltalk convention.

```
nextRandom
"Update the current random display with the Next button is clicked"

   currentRandomValue value: myRandom next
```

As was the case for new, the next message also might be different, and needs to be checked.

```
resetSequence
"Start a new Random, with a specified seed if seed is not 0"

   | seedValue |
   seedValue := seed value.
   seedValue = 0
      ifTrue: [myRandom := Random new]
      ifFalse: [myRandom := Random new seed: seedValue].
   currentRandomValue value: 0.
```

And in this method, again there is the instance creation method, but also the method for setting the seed, using the instance method seed:.

Compare these methods with those for Random in mscorlib. In a System Browser (it may be convenient to open a new one, for easy comparison), select the DotNETBaseClasses bundle or package, and find the Random class. Note that this is a VisualWorks class serving as a proxy for the mscorlib Random class.

Note that, although the class name is the same as the standard VisualWorks Random class, it is distinct, by virtue of being in a different name space. Look at the class definitions to see that the mscorlib proxy class is defined in the DotNET.System name space rather than Smalltalk.Core.

Browsing the proxy class, look at the instance creation method category, in the class methods view. There are two methods: New and New:. New: takes a seed as argument, and so will be useful in the resetSequence method. New will be used wherever RandomNumberPicker originally used new.

Switch to the instance method view, and select the methods method category. There are several methods available, but we only need Next, which is the same as the original next, except for starting with an uppercase letter.

Because there are obvious equivalent methods in the .NET library to those in the VisualWorks library, the rewrite is very simple. The three methods listed above can be simply rewritten as:

```
initialize
 "Create the initial random number generator for this application"

   myRandom := Random New.

nextRandom
"Update the current random display with the Next button is clicked"

   currentRandomValue value: myRandom Next

resetSequence
"Start a new Random, with a specified seed if seed is not 0"

   | seedValue |
   seedValue := seed value.
   seedValue = 0
      ifTrue: [myRandom := Random New]
      ifFalse: [myRandom := Random New: seedValue].
   currentRandomValue value: 0.
```

Make and save these changes.

We are almost done, but not quite. If we tried to open the modified application now, by evaluating RandomNumberPicker open, we would get a **Message Not Understood** walkback notifier saying that New is not understood. We need to adjust the name space.

In the System Browser, browse the WalkThru package/parcel and select the WalkThru name space. It imports Smalltalk.*, which was fine while we were using Random defined in Smalltalk.Core. But, we need the Random defined in DotNET.System. So, by changing the import line, as shown below, we can correct this and complete the application:

```
Smalltalk defineNameSpace: #WalkThru
    private: false
    imports: '
    private DotNET.System.*
    '
    category: 'Tutorial-WalkThru'
```

Launch the application now to verify that it works properly.

As mentioned above, however, using a .NET class is easy. The rest of this document will address issues of representing external .NET classes in VisualWorks.

The numeric values returned by the .NET and the VisualWorks classes are different, so simply observing the values displayed indicates that the program is operating correctly and invoking the .NET class as intended.

2

Adapting an Assembly

In order to make use of the classes and methods defined in a .NET assembly from within VisualWorks, the assembly must be adapted for access from VisualWorks. This is done by creating a proxy DLL to bridge between VisualWorks and the .NET CLR, and by generating Smalltalk classes and methods representing the corresponding .NET classes and methods.

This chapter describes how to create the adaptor DLL and Smalltalk code. Two procedures are described. The preferred procedure is to use the DotNETConnect Creation Wizard, which simplifies the process of identifying an assembly and the members to adapt. You can also create the necessary adaptor code programmatically, using Smalltalk messages.

Procedures for generating the Smalltalk and C++ source code are described first using the wizard, then using programmatic procedures. Then, procedures for compiling resulting the C++ code and for loading the Smalltalk code are described.

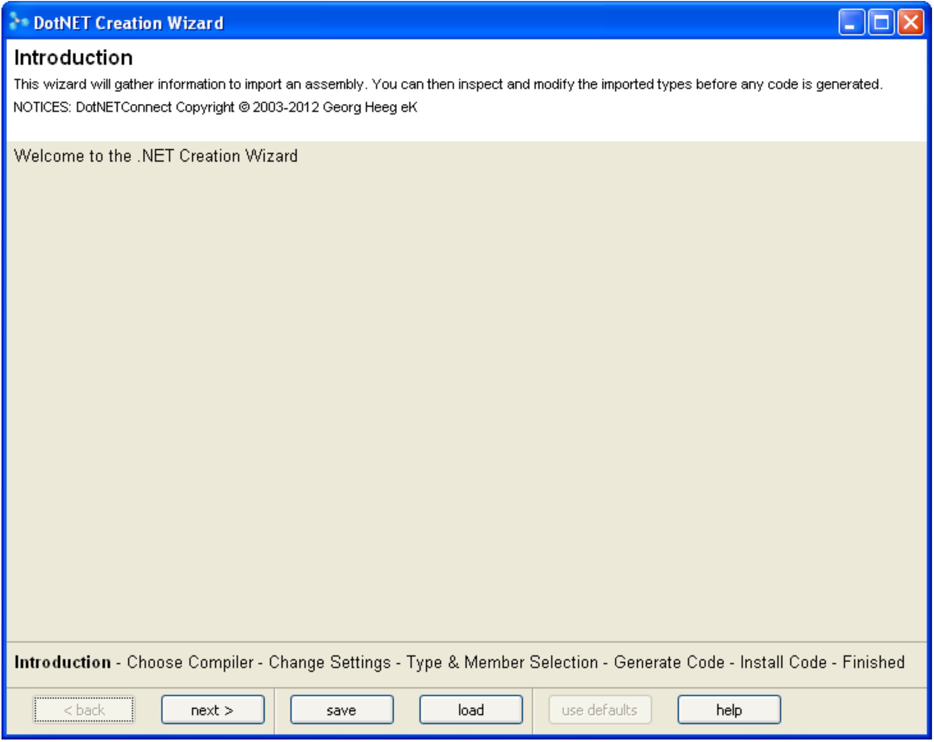
Creating Adaptors Using the Wizard

The DotNETConnect Creation Wizard leads you through the adaptor creation process. To start the wizard, select **Tools > DotNETConnect Code Generator**. Alternatively, evaluate:

```
DotNETCreationWizard open
```

or select an assembly in the Global Assembly Cache Browser and click the Launch button. Any of these actions launches the wizard.

The first screen simply describes the process and lists the steps the wizard will perform to generate the code for an assembly. Click **Next** to advance to the first step.

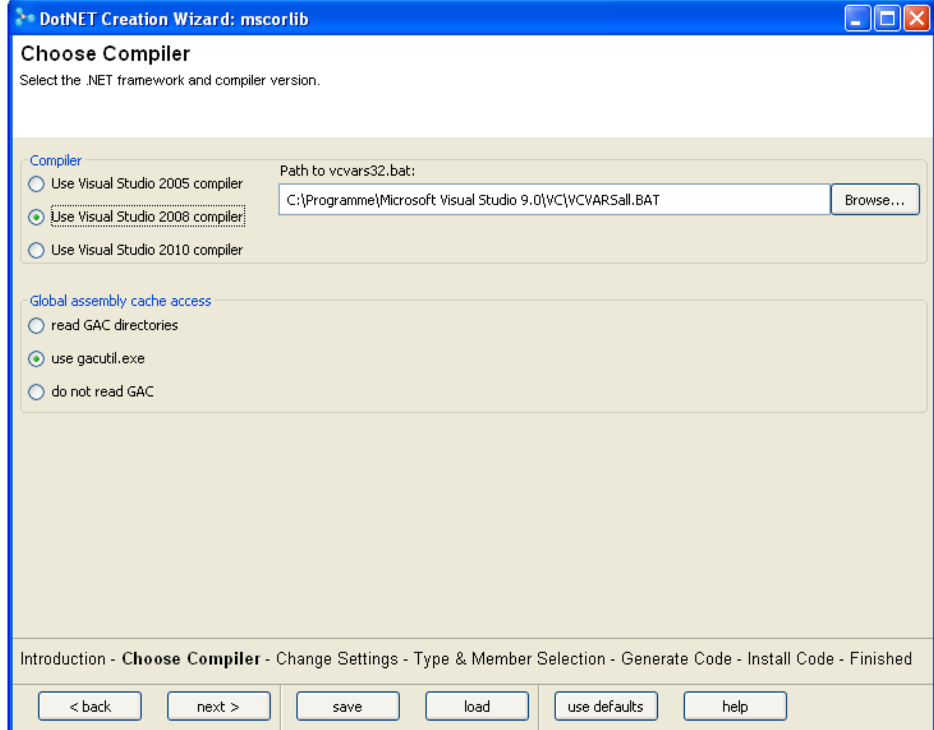


The wizard leads you through a five step procedure:

1. Choose your compiler version.
2. Identify the assembly to adapt, and set several parameters (see Change Settings)
3. Specify any additional types and members to exclude (see Review Kept/Deleted)
4. Generate the necessary C++ and Smalltalk code (see Generate C++ and Smalltalk Code)
5. Compile the generated C++ and load the Smalltalk code (see Install C++ and Smalltalk Code)

Once the wizard has successfully completed its work, you you can use the generated code. If there are errors that require code modification,you must compile the C++ code and load the Smalltalk code, as described under Compiling the Proxy Code and Installing the Smalltalk Stubs.

Choose the Compiler



Select the compiler you want to use. There are a few noteworthy aspects to the compiler choice:

- The version of the compiler determines which assemblies in the global assembly cache (GAC) are discovered by gacutil. Assemblies that are part of framework version 4.0 (or higher) are not discovered if you use VisualStudio 2005 or 2008.
- If you have several versions of one assembly installed, the compiler will choose the newest one it can handle, no matter which version you selected. For example. if you have versions 2.0, 3.5 and 4.0 of **mscorcfg** installed, VisualStudio 2005 and

2008 will compile against version 3.5, while VisualStudio 2010 will compile against version 4.0

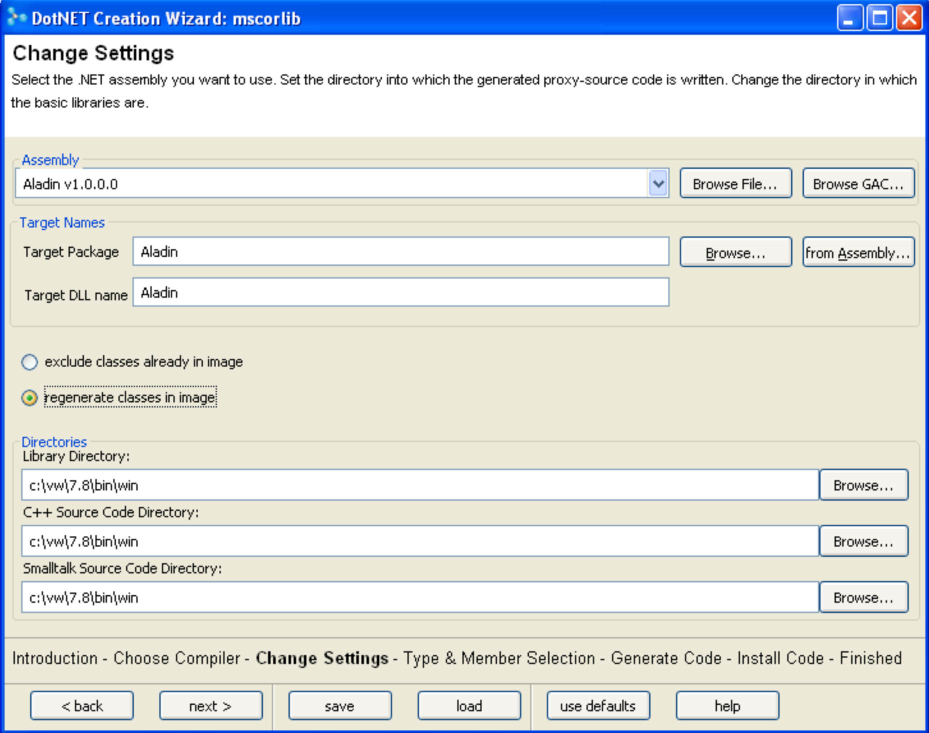
- By default, version 4.0 assemblies use a different runtime than earlier versions. Objects of the new runtime cannot access objects of the old one, and vice versa. This means that DotNETConnect, whose registry lives in the version 2 runtime, cannot access assemblies that run in the new runtime. To use assemblies that require version 4 of the framework, you need to set the useLegacyV2RuntimeActivationPolicy, by copying **DotNETConnect\sources-dll\vwnt.exe.config** into the VM directory (normally **bin\win**). See http://msdn.microsoft.com/en-us/library/bbx34a2h.aspx for more info on this setting.

The **vcvars32.bat** batch file sets up the environment for your VisualStudio C++ compiler. It enables VisualWorks to call the compiler, making manual compilation unnecessary. Depending on your VisualStudio installation, it may be called **vcvars32.bat**, **vcvarsAll.bat** or something similar.

Change Settings

In this step you select the assembly to adapt, and to specify additional settings for code generation.

For this example we have selected **`Aladin.dll`**, which is provided with the DotNETConnect (in **`DotNETConnect\Sample)`**. We will use this as the primary example for adapting an assembly in this chapter.



When you have specified the property values, click **next>** to set the values and proceed.

The properties and options are as follow.

Assembly

This property identifies the .NET assembly that you are adapting. You can adapt either a shared or an unshared assembly. Shared assemblies are usually installed in the global assembly cache (GAC). The drop-down list shows the assemblies in the GAC. You can also click **Browse GAC...** to open the GAC browser.

For unshared assemblies, click **Browse File**, navigate the file system and select the file. The file must be either an EXE or DLL that is the entry point to the .NET assembly.

Target Package

This property specifies the package that will contain the generated VisualWorks classes. We recommend defining one package for each assembly.

To define a new package, enter the package name. To select an existing package, click **Browse...** . To use the assembly name as the package name, click **from Assembly...** .

Target DLL name

This property specifies the name of the generated files. We recommend using the same name as the package.

Exclude classes already in image

If you have already generated code for some classes, these classes can be automatically excluded from the list. In the next step you can fine-tune the selection of classes and members.

Directories

Three directories must be specified, which identify the location of DLLs and the target directory for generated files. You may enter the diretory paths or click **Browse...** and select each directory. Specifically, the directories are:

Path to vcvars32.bat	This batch file sets up the environment for your VisualStudio C++ compiler. It enables VisualWorks to call the compiler, making manual compilation unnecessary. Depending on your VisualStudio installation, it may be called **`vcvars32.bat`**, **`vcvarsAll.bat`** or something similar.
Library Directory	The directory where we installed the DLLs for DotNETConnect. By default it is the directory where the virtual machine is installed
Source Code Directory	The directory where the code generator generates the code for the proxy. Default name is Sources-DLL.
Smalltalk Source Code Directory	The directory where the code generator generates the code for the stubs. Default name is Sources-VW.

Check Types

Because of differences between the languages, assemblies may contain types and members that cannot be adapted for use by VisualWorks. In this step DotNETConnect checks for all types and members in the assembly, to verify whether they can be adapted to VisualWorks.

The validation check tests whether:

- the base type of a type is known (if it is a DotNETConnect base type, an interface, or if the basetype is already represented, e.g., in the DotNETConnect base classes)
- the type of the return value of a member is known
- the parameters of a member are all known types
- the dimension of an array (as return type or as type of a parameter) is 1 (currently DotNETConnect does not support multidimensional arrays)
- a delegate has exactly one Invoke method
- a nested class is nested inside a private class

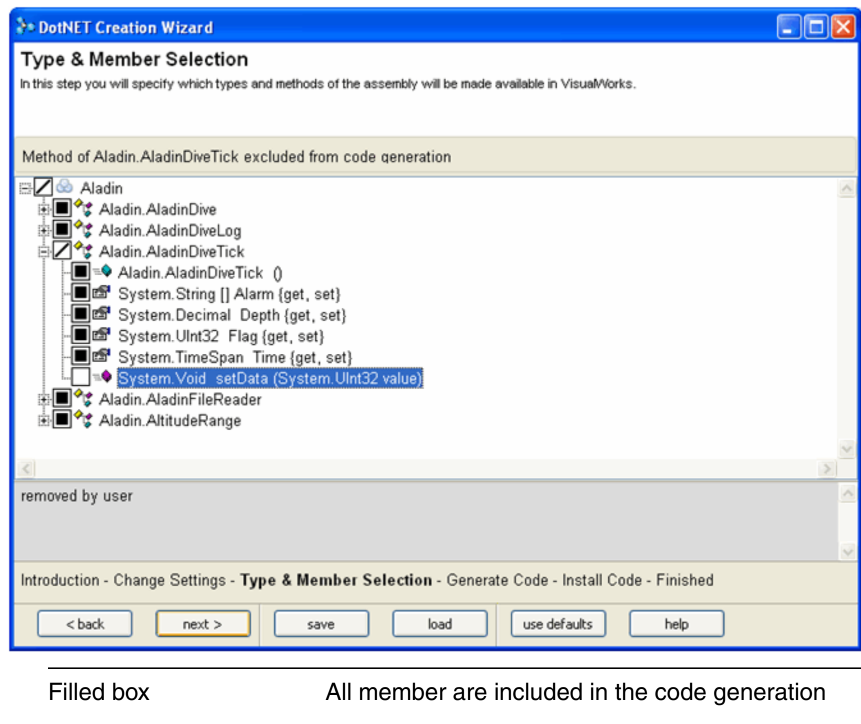
Any type or member that fails the test is marked to be ignored when generating the code. A notifier window reports how many items were removed, if any. You are also given the option of viewing a report of the the types and members that failed the tests and the reasons why they failed.

If a member or type that you need is rejected because of an unknown base, return, or parameter type, you should first generate code for the assembly that defines the missing type. For example, to use the classes System.Windows.Forms.Button or System.Windows.Forms.Menu from the System.Windows.Forms assembly, you need to generate code for the System assembly first to get their base class System.ComponentModel.Component. The report will inform you which assemblies define the missing types.

Review Kept/Deleted

This step shows the initial selection of types and methods in the assembly which will be kept (imported) or deleted (notimported), based on the analysis of the previous step.The first level of the tree shows the namespaces, the second level shows the classes and interfaces,the third level shows the members of the class. Each entry

has two icons, the first shows if this entry is excluded, either partially or entirely; the second icon has the same meaning as in the Object Browser.



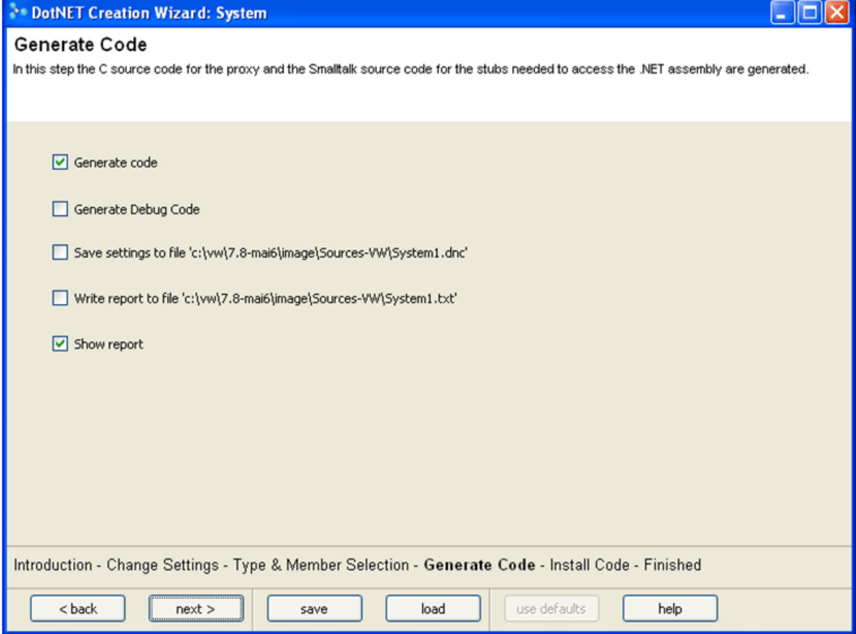
Filled box	All member are included in the code generation
Check mark	Some members were excluded by the automatic check
Diagonal line	Some members were excluded by the user
Empty box	All members are excluded

For the Aladin sample assembly, all types and members satisfied the checks, so all boxes are filled.If you see only empty boxes, that is because you have already loaded the Aladin example and the wizard detected that these classes do not need to be redone.You can select or deselect either types or their methods. For example, if you know that you do not need a class or some of its members, select that item and click on its check box. Deselecting a type deselects all its methods as well.For classes, a context menu allows you to automatically remove all subclasses and/or references as well.

Note: Keeping methods that the analysis has selected to remove requires special consideration. The DLL source code will probably fail to compile without additional effort. If you need to keep a method, review the analysis report for the reason the it was ignored. That will be helpful in correcting the requirements. The kinds of correction that may be required are beyond the scope of this document.

Generate C++ and Smalltalk Code

With the information provided by the preceding steps, everything is prepared for generating the supporting code. When you click **next>**, code generation proceeds without further user intervention.



This step generates:

- C++ code for the proxy and a make file to compile it. This code is stored in the source directory specified in the settings screen.
- Smalltalk code for the stubs and stores them into the Smalltalk source directory specified in the settings screen.

The wizard creates five source files and a make file for the proxy, and four files for the stub methods. The file names are based on the Target DLL name specified in the settings page (represented by * in the tables below). If you select the option Generate Debug Code, the target DLL will be compiled with debug information. The generated C++ file names and their contents, which are written to the Sources-DLL directory, are:

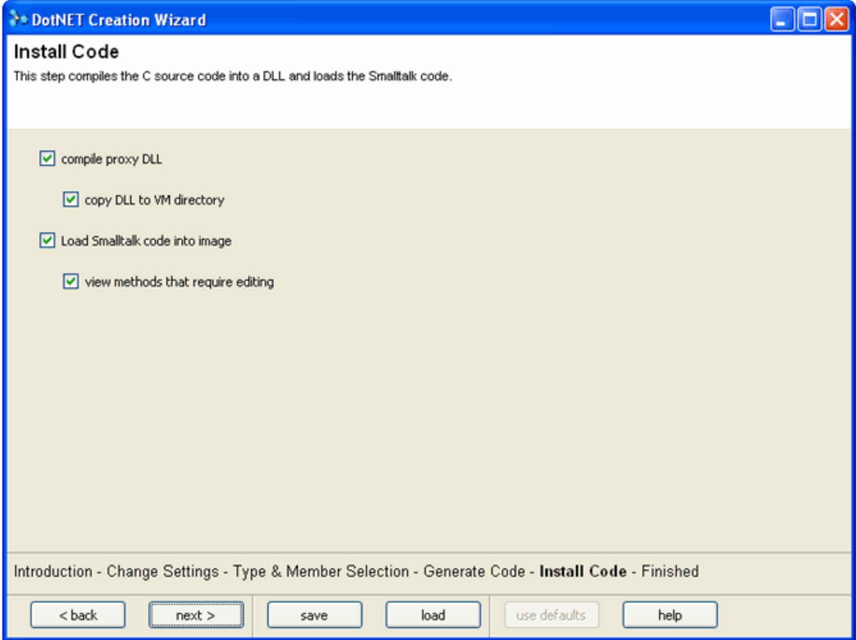
*Proxy_class.h	The C++ header file that defines the proxy class
*Proxy _class.cpp	The C++ source file that implements the proxy class
*Proxy _dll.h	The C header file that defines the DLL for the VisualWorks DLLCC
*Proxy _dll.cpp	The C header file that implements the DLL for the VisualWorks DLLCC
*Proxy_eventClass.h	The C header file that defines events
*Proxy.mak	The make file to compile the DLL

The generated Smalltalk filenames and their contents, which are written to the Sources-VW directory, are:

*Proxy_1_namespace.st	The file contains the namespace and package definition for the VisualWorks stub
*Proxy_2_ext_interface.st	The file contains the external interface to connect to the proxy DLL
*Proxy_3_ext_stub.st	The file contains the wrapper for the external interface
*Proxy_4_domain_stubs.st	The file contains the stubs for the imported classes of the assembly

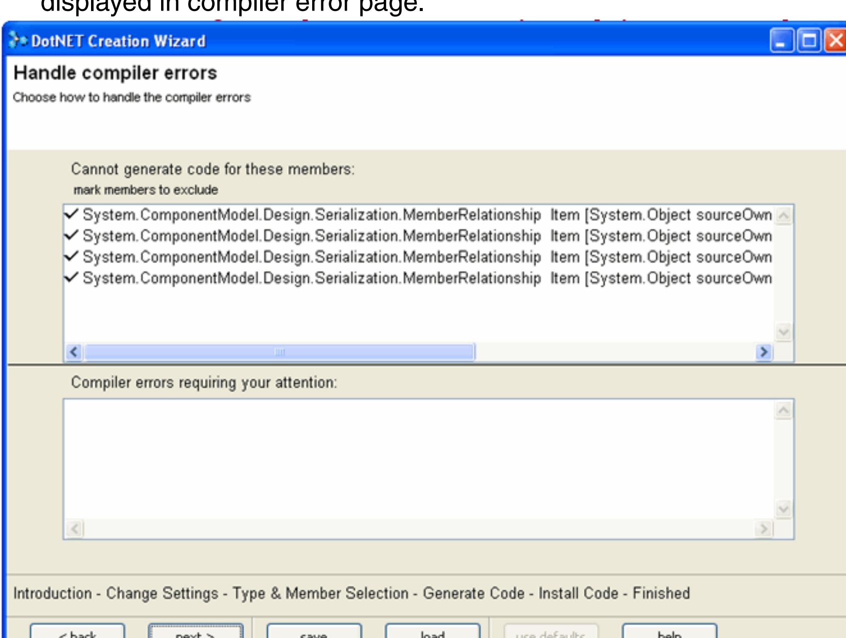
Install C++ and Smalltalk Code

This step compiles the DLL and installs the Smalltalk code.



When you click **next>**, the C++ files are compiled into a proxy DLL and the Smalltalk files are loaded into the image. If no errors occur, you are transferred to the final step, which lists the results of the code generation and installation.

If there are errors during the compilation of the DLL, they are displayed in compiler error page.



The upper list shows errors which can be attributed to the proxy code of one specific member. If you do not need this member, you can keep the check mark of the list entry and the member will be excluded. If all errors can be resolved by removing the member, the **next>** button brings you back to the Compile Code page for a new try without the marked member. If there are unchecked items in the upper list or items in the lower list, the **next>** button brings you to the final screen. It contains the complete error messages of the compiler. See Eliminating Compiling Errors below.

You can also always back up to the Type & Member Selection step using the **<back** button.After filing in the Smalltalk code, all methods for which DotNETConnect could not generate the complete code are listed in another Window. See Resolving Method Signatures.

Creating Adaptors Programmatically

The Code Generator wizard is a convenience, but at times it is desirable to take a programmatic approach. This section describes the process for doing this, again using the Aladin sample assembly.

Read the Assembly Members

First, read the types and members of an assembly to generate an assembly description. To do so, send an on: instance creation message to AssemblyDescription. The argument identifies the assembly, and the information string provided varies based on the location of the assembly.

For a shared assembly that is installed in the GAC, the identifier is a string including the name, version, culture, and public key. This string can be copied from the display field in the DotNETConnect Browser. For example, to adapt the System.Drawing assembly, browse the version to adapt, and enter the information into the message as follows:

```
asmDesc := AssemblyDescription on:
   'System.Drawing, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a'
```

(Note that there must not be a line break in the String literal.) For an unshared assembly, one that is not installed in the GAC, you must provide the complete path name as the identifier. For example:

```
asmDesc := AssemblyDescription on:
   'C:\vw7.3\preview\DotNETConnect\sample\ Aladin.dll
```

These commands create the AssemblyDescription instance and read in all the types defined in the assembly.

If you want to include only a selected number of types, send a on:types: message to AssemblyDescription:

```
asmDesc := AssemblyDescription on: aName types: aCollection.
```

aCollection should be a collection of strings with the type names including namespaces.

Checking the assembly

Often an assembly contains types and members that cannot be imported. To check that the types in the assembly can be included, evaluate:

```
asmDesc analyzeAssembly.
asmDesc check
```

Any types that are not known are added to the AssemblyDescription’s ignoreType list.

You can inspect the AssemblyDescription (asmDesc in the above examples) to see what is now held in the description.

Generate the Code

After reading the type signatures of the assembly we can now generate the source files for the assembly proxy. A make file is also created to aid in compiling the proxy.

Smalltalk code must be generated for two purposes: to handle the external interface to the proxy DLL, and to represent the classes and methods in the assembly. To generate the code evaluate this expression:

```
DotNETCodeGenerator
   createProxyForAssembly: asmDesc
   in: 'C:\development\projects\dotnet'
   packageName: 'AladinProxy'
```

The output directory (the in: argument) must exist and have two subdirectories named sources-vw and sources-dll; the generator does not create the directories.

The code generator creates the files:

- **`AladinProxy_1_namespace.st`**
- **`AladinProxy_2_ext_interface.st`**
- **`AladinProxy_3_ext_stubs.st`**
- **`AladinProxy_4_domain_stubs.st`**

File-in these files as described in Installing the Smalltalk Stubs.

Compiling the Proxy Code

Once the C++ code is generated, you need to compile it. DotNETConnect generates a make file that can be used in a command prompt to compile the DLL.

You must define environment variables for the compiler. There are three options:

- When using the Visual Studio .NET 2005/2008/2010 command prompt, the environment variables are set by the environment.
- When using a standard command prompt, but with Visual Studio .NET installed, execute the Visual Studio .NET 2005/2008/2010 environment batch file (**`vsvars32.bat`**) at the command prompt.

- With the .NET SDK installed but without Visual Studio .NET, execute the SDK environment batch file (**`SDK\v1.1\Bin\sdkvars.bat`**) at a standard command prompt.

In order to compile the just created source code this directory should contain the files **`DotNETProxyBase.h`** and **`DotNETProxyBase.cpp`**. For VisualStudio 2005, also include the files **`standard.manifest`** and **`Reshacker.exe`**.

The developer now can compile the C++ code. For example, to compile the Aladin example code, **`cd`** to the directory with the generated source code and execute:

```
nmake AladinProxy.mak
```

After the compile and link process, copy the created DLL (e.g., **`AladinProxy.dll`** in our example) and the assembly (e.g., **`Aladin.dll`**) to the directory that contains the VisualWorks virtual machine. If the assembly includes several files, they must all be copied.

Eliminating Compiling Errors

Even though we checked the assembly before generating the DLL source code, you may still get some compiler errors. To resolve these, you must identify the assembly member causing the problem, and remove it from inclusion.

For example, using either the wizard or the programmatic approach described above to prepare the System.Drawing (version 1.0.5000.0) assembly for import produces the following compiler errors:

```
SystemDrawingProxy_class.cpp
SystemDrawingProxy_class.cpp(11347) : error C2660:
    'System::Drawing::Imaging::ColorMatrix::get_Item' : function does not
    take 1 arguments
SystemDrawingProxy_class.cpp(11378) : error C2660:
    'System::Drawing::Imaging::ColorMatrix::set_Item' : function does not
    take 2 arguments
SystemDrawingProxy_class.cpp(30107) : error C2691: 'unsigned char
    __gc[]' : invalid type for __gc array element
SystemDrawingProxy_class.cpp(30107) : error C2691: 'unsigned char
    __gc[]' : invalid type for __gc array element
```

The basic solution is to identify the assembly member, generally a method or a class, that is giving rise to the error, and eliminating it from inclusion.

In this case, all of the errors are occuring while compiling the file **SystemDrawingProxy_class.cpp**. The line number shows precisely where the error occured. Using a text editor that uses line numbers, open the file and go to the line. For the first error, the context is:

```
// C++ proxy body for the property getter defined in
     System.Drawing.Imaging.ColorMatrix:// System.
Single  Item [System.Int32 row, System.Int32 column]
    {get, set}int SystemDrawingProxy::
    System_Drawing_Imaging_ColorMatrix_Get_Item(return_type*
     returnValue, wchar_t* objectID, __int32 row, __int32 column){
    int error_code = 0;
    try
    {       Object* rawObject = this->DeReferenceObject(objectID);
System::Drawing::Imaging::ColorMatrix* dotNetObject =

__try_cast<System::Drawing::Imaging::ColorMatrix*>(rawObject);
        System::Single theObject = dotNetObject->Item[row][column];
        this->WrapSingle(theObject, returnValue);
    }
```

The error line itself does not tell us what to exclude, but is in the definition that needs to be excluded. To identify the member to exclude, look at the second comment line and the method signature on the first uncommented line. This indicates that the member is System.Single Item [System.Int32 row, System.Int32 column] defined in class ColorMatrix.

Remove a Member Programmatically

To remove a single member, you can use the method #removeMember:fromAssembly: in class DotNETCodeGenerator. For example to remove this method:

```
// C++ proxy body for the method defined in System.DateTime:
//System.Int32  CompareTo (System.Object value)

intMscorlibProxy::System_DateTime_CompareTo02(return_type*
    returnValue, wchar_t* objectID, wchar_t* value)
{
    int error_code = 0;

    try
        {
            Object* arg3 = this->UnMarshallObject(value);
            Object* rawObject = this->DeReferenceObject(objectID);
            System::DateTime* dotNetObject =
```

```
            __try_cast<System::DateTime*>(rawObject);
         System::Int32 theObject = dotNetObject->CompareTo(arg3);
         this->WrapInt(theObject, returnValue);
      }
      use
      DotNETCodeGenerator removeMember:
   'System_DateTime_CompareTo02' fromAssembly: asmDesc.
```

To remove a property, e.g. the Item property in the class ColorMatrix, send

```
DotNETCodeGenerator removeMember:
   'System_Drawing_Imaging_ColorMatrix_$_Item' fromAssembly:
      asmDesc.
```

It is not possible to only remove the getter or setter of a property, you must always remove the complete property.

As described above, the System.Drawing assembly causes additional compilation errors. You can continue to find and resolve these one at a time by removing methods individually. In some cases, though, it is more efficient to remove a whole class and its methods. For the ImageCodecInfo class, for example, which produces a new compilation error after each individual deletion, it is easiest just to remove the whole thing. This can be done with these expressions:

```
asmDesc moveToIgnoreList:
   ( asmDesc types select: [ :x | x name = 'ImageCodecInfo']) first
```

This removes the class, all of its members and all other members that reference this class. To remove only the references to a class, e.g. if this class is defined in another assembly, use DotNETCodeGenerator class>>removeReferencesTo:inAssembly:. For example, to remove all members that have parameters of type System.DateTime, or return a System.DateTime, use

```
DotNETCodeGenerator removeReferencesTo:
   'System.DateTime' inAssembly: asmDesc.
```

Then regenerate the code and compile.

Once you can compile the DLL source code, use the same assembly description to regenerate the Smalltalk execution and class stubs as described above in Generate the Code.

Installing the Smalltalk Stubs

DotNETConnect also created four Smalltalk source code files, in the VisualWorks source directory (**`Sources-VW/`**). The source files must be filed in in the correct order, as indicated by the numbers in the file names:

1 *Proxy_1_namespace.st

2 *Proxy_2_ext_interface.st

3 *Proxy_3_ext_stub.st

4 *Proxy_4_domain_stubs.st

Resolving Method Signatures

.NET identifies methods and constructors based on their signatures, which include the return type, the names and the types of the parameters. Accordingly, there frequently are several implementations of methods or constructors with the same name, but differing in their signatures. Smalltalk, however, differentiates methods by their selectors only.

DotNETConnect creates stub methods for each method and constructor based on its name and the number of its parameters. The methods for the wrapper are automatically numbered to match a possible COM or remoting channel. Thus the methods in the wrapper are unique.

In the stubs there may be two or more methods with the same name and the same number of parameters. When possible, DotNETConnect creates methods that check the parametersl and dispatch to the correct stub methods, but in certain conditions DotNETConnect cannot find the correct tests and creates a skeleton for these methods that must be resolved by a developer after the code generation. These methods are marked by the message forManualEditing which is placed as only statement in the method body. The different representations for the method are places in the stub method as comment.

For example, the mscorlib class SortedList has three instance creation methods, differing in the type of their single argument:

- SortedList (System.Collections.IComparer comparer)

- SortedList (System.Collections.IDictionary d)
- SortedList (System.Int32 initialCapacity)

It is possible that a class implements both the IComparer and the IDictionary interface. In this case, the order of the tests influences which creation method would be used for instances of the class. Therefore, the code generator leaves that decision to you.

Since Smalltalk does not differentiate the methods for these three signatures, the code generator creates this skeleton New: method:

```
New: comparer
   "The corresponding .NET constructor exists in 3 variants:
   1. SortedList (System.Collections.IComparer comparer)
   2. SortedList (System.Collections.IDictionary d)
   3. SortedList (System.Int32 initialCapacity)"
   "SortedList (System.Collections.IComparer comparer)
   | returnData |
   returnData := MscorlibStub current
      System_Collections_SortedList_New02: comparer.
   ^returnData

   | returnData |
   returnData := MscorlibStub current
       System_Collections_SortedList_New03: d.
   ^returnData

   | returnData |
   returnData := MscorlibStub current
       System_Collections_SortedList_New04: initialCapacity.
   ^returnData "
   self forManualEditing
```

Edit this method to perform the correct actions for each of the parameter types:

```
New: c
   "This method supports three signatures. "

   ^c isInteger
      ifTrue: ["SortedList (System.Int32 c)"
      (c > 0 and: [c <= 16r7FFFFFFF]) ifFalse:
         [self invalidSizeError raise].
      MscorlibStub current System_Collections_SortedList_New04:
          initialCapacity. c]
      ifFalse: [c implementsInterface: 'System.Collections.IComparer'
         ifTrue: [MscorlibStub current
             System_Collections_SortedList_New02: c]
```

```
ifFalse: [c implementsInterface: 'System.Collections.IDictionary]
    ifTrue: [MscorlibStub current
        System_Collections_SortedList_New03: d]
    ifFalse: [self typeDispatchError]]].
```

Importing MSCorLib

For the special case of MSCorLib, the core .NET library, the following can be used:

```
mscorlibDesc:= DotNETCodeGenerator createTypes
```

DotNETCodeGenerator class method createTypes is a small utility method built to simplify generating the proxy and stubs for MSCorLib. This assembly contains many more types than can be used in VisualWorks. The class reads only a useful subset or the types in the assembly.

Caution: If you need to import MSCorLib to import more functions of the library, be aware that doing so will overwrite much of the work already provided in DotNETConnect.

You can modify the subset of types by editing the class methods, which list the types that are to be included. Browse the class accessing methods to see how this is done.

A similar approach can be used to restrict the types of other assemblies, and might be useful if you repeatedly import the same assembly.

3

Using an Assembly in VisualWorks

By installing DotNETConnect, you already have access to standard .NET classes included in the MSCorLib assembly. Adapting additional assemblies gives you access to the classes and methods defined in them.

All .NET proxy classes are created as subclasses of `DotNETObject`. Browse its subclasses to see what classes are installed.

This chapter describes how to use the facilities provided by .NET assemblies within VisualWorks.

Using a .NET Assembly Proxy

Facilities provided by .NET base classes are generally simple to use. One example, using `Random`, was given in Developing with DotNETConnect. In this section we give additional examples and discuss other usage issues.

Supported Types

.NET and its supported languages define types and members that cannot be integrated into VisualWorks, because they are not supported in an untyped system. Therefore, DotNETConnect supports only those member types that can be represented in VisualWorks.

.NET supports four object types (classes, value types, interfaces and enumerations).

Base Types

Some types, which we call "base types," are transported by their value between VisualWorks and .NET. They are represented in VisualWorks and .NET by different classes and are converted when used as a parameter or return value. These types cover numeric values (int, float, single, decimal …), Boolean values, characters and strings. Because .NET uses UTF-16 as strings encoding we represent them as two byte strings.

The following list shows all .NET base types and their counterparts in VisualWorks

.NET	VisualWorks
Boolean	Boolean
Byte	Integer
SByte	Integer
Int16	Integer
UInt16	Integer
Int32	Integer
UInt32	Integer
Int64	Integer
UInt64	Integer
Single	Float
Double	Double
Decimal	FixedPoint
Char	Character
String	TwoByteString

Enumerations

Subclasses of System.Enum have a predefined set of instances that correspond to integers. For example, the class MidpointRounding has two instances: MidpointRounding ToEven (0) and MidpointRounding AwayFromZero (1).

Instances of enumerations are often used to specify options, for example:.

```
searcher := CultureInfo CurrentCulture CompareInfo.
searcher IndexOf: 'Some text' with: 'some'
   with: CompareOptions IgnoreCase.
   "searches case insensitive"
```

Some enumerations allow combining options, for example:

```
(FileSystemRights ReadAttributes |FileSystemRights ReadData |
   FileSystemRights ReadExtendedAttributes |FileSystemRights
   ReadPermissions)
== FileSystemRights Read
```

Reference Types

All other objects remain in .NET, and a reference, which is a unique object id, is transported between VisualWorks and .NET. Both the proxy in .NET and the stubs in VisualWorks use a registry to connect a reference to its real object (in .NET) or stub (in VisualWorks). This ensures that you always get the same object in VisualWorks for the same .NET object.

For each object id, the stub object is chosen according to the following rules:

1. If the stub object already exists, it is found in the registry and returned.
2. If a proxy class exists in VW for the class of the .NET object, an instance of this class is created.
3. Otherwise, an instance of the type specified in the signature is created. This is either a superclass of the real type, or an interface type.

For example, according to the signature, the method getType: in Assembly returns a System.Type, but it really returns a RuntimeType, which is a private subclass of Type. If there is a proxy class for RuntimeType, then the result of such a method send will be a RuntimeType object, otherwise a Type object.

Types in .NET have static and non-static members (methods, constructors, properties and events). Static members are executed for the type itself and non-static members are executed for instances of a class. This distinction is similar to class-instance method distinction in Smalltalk. Therefore, DotNETConnect implements static member as class methods and non static member as instance methods.

Creating instances

First, we create a DateTime, which is the .NET equivalent of a VisualWorks Timestamp. Browsing the DateTime proxy class in VisualWorks, on the **Class method** page, there are several method categories in addition to the familiar instance creation category. Methods in the additional categories can often be used to create an instance as well as those in the instance creation category. For example, the Now method in the properties category creates a DateTime instance with the current date and time:

```
DateTime Now.
```

The category reflects the kind of member the method represents in .NET. Now, in .NET, is a property, but it returns a DateTime instance. Because it is a static property, it is rendered in DotNETConnect as a class method returning a new instance.

The standard .NET constructor methods consist of the class name followed by a number of arguments, such as:

```
DateTime (System.Int32 year, System.Int32 month, System.Int32 day,
    System.Int32 hour, System.Int32 minute, System.Int32 second,
    System.Int32 millisecond, System.Globalization.Calendar calendar)
```

These are represented as Smalltalk instance creation methods named New, New:, or New:with:..., with the number of keywords determined by the number of arguments in the original constructor. For the constructor above, for example, the corresponding instance creation method selector is:

```
New: year with: month with: day with: hour with: minute with: second
   with: millisecond with: calendar
```

The keyword argument variable names are the same as the .NET constructor argument variable names. Operators are implemented as static methods in .NET (the C# compiler translates operators into these methods based on receiver and argument type), for example:

```
TimeSpan
   op_Addition: (TimeSpan fromHours: 2)
      with: (TimeSpan fromMinutes: 30).
```

For the argument types, you need to refer to the constructor signature, which is included in the instance creation method comment for your convenience. Int16 or Int32 argument types can be provided by Smalltalk Integer values; the appropriate Integer subclass (SmallInteger or LargeInteger) is selected by DotNETConnect. NET allows several implementations of methods or constructors with the same name, but differing in their signatures. When possible,

DotNETConnect generates methods that select the appropriate implementation at runtime. If more than one implementation matches the parameter types, the most specific one is used, e.g:

```
Convert ToByte: '123' with: NumberFormatInfo New.
   "uses ToByte (System.String, System.IFormatProvider)"
Convert ToByte: true with: NumberFormatInfo New.
   "uses ToByte (System.Object, System.IFormatProvider)"
```

Special argument types need to be created appropriately. In this case, an appropriate Calendar instance must be created and provided as the last argument. So, to create a DateTime instance using the method above, you would evaluate an expression such as:

```
DateTime New: 2004 with: 7 with: 28 with: 9 with: 14 with: 39 with: 0
   with: GregorianCalendar New
```

The value returned by such constructors can then be used for successive messages sent to the .NET assembly. For example, using the Now property as an instance creator:

```
| now gregorianYear thaiYear |
now := DateTime Now.
gregorianYear := GregorianCalendar New GetYear: now.
thaiYear := ThaiBuddhistCalendar New GetYear: now.
```

The results are held in VisualWorks in instance variables, and so can be used directly by other VisualWorks expressions. For example, to display the results in a text window, you can format them as follows:

```
params := Array with: gregorianYear with: thaiYear
   with: (thaiYear - gregorianYear).
^('Current year from gregorian calendar: <1p> Current year from Thai
   calendar: <2p> Difference between Thai and Gregorian calendar: <3p>'
   expandMacrosWithArguments: params)
```

Generic Types

To construct an instance of a generic type, you need to specify its generic placeholder types, using proxy classes or instances of System.Type. For example:

```
myList := List_1 NewT: TimeSpan.
   "returns a List that stores TimeSpans"
Dictionary_2 NewTKey: String withTValue: ArrayList
   "returns a Dictionary where keys are Strings and values are ArrayLists"
List_1 NewT: myList GetType.
   "returns a List of Lists of TimeSpans"
```

Generic constructors can have parameters and type specifications, for example:

```
List_1 New: 100 withT: BitArray
   "returns a List of BitArrays with 100 slots"
```

Converting between .NET and Smalltalk types

In some cases it is useful to convert between VisualWorks and .NET types.

Some classes, such as DateTime, for which conversions may be common, can provide explicit conversion methods. DotNETConnect includes methods to convert DateTime instances to VisualWorks Date, Time, and TimeStamp instances, and to create a DateTime instance from instances of these classes. Browse the conversion method category, both on the instance and class tabs in the browser, to see these methods.

As illustrated by these conversion methods, the general approach is to create a new instance of the desired class, either in VisualWorks or .NET, using property values of the existing object to specify the new object. For example, to create a new Time (VisualWorks) object from an existing DateTime (.NET) object, send an instance creation message to Time with parameters derived from a DateTime instance:

```
dt := DateTime Now.
Time fromSeconds: ((dt Hour * 3600) + (dt Minute * 60) + dt Second)
```

Conversly, to create a new DateTime object from an existing Time object, send the appropriate instance creatin message to DateTime with the properties of the Time as arguments:

```
time := Time now.
DateTime New: 1 with: 1 with: 1
   with: time hours with: time minutes with: time seconds
```

As shown here, values that are not provided by the original object may need to be supplied, even with incorrect values. In this case, the DateTime object is only useful for holding the time, since the year is inaccurate.

Argument types

All .Net integer objects (System.Int64, System.Int32, System.Int16, System.UInt64, System.UInt32, System.UInt16) are automatically converted to VW Integer objects (SmallInteger, LargePositiveInteger, LargeNegativeInteger). Similarily, other numeric values are converted: Decimal to FixedPoint, Single to Float, and Double to Double.

Apart from the return values, .NET can return values in parameters defined as reference parameters in the method signature as well. Smalltalk does not support the concept of reference parameters, but you can achieve the same effect with value holders. For example, if you adapt the WordCount example assembly that is provided with the .NET SDK, CountStats:with:with:with:with:with: is supposed to write the various counts into numLines, numWords, numChars and numBytes. So you need to use something with semantics like a value holder for the parameters. For this purpose DotNETConnect has the class DotNETRefByObject. Its main difference to the class ValueHolder is that it does not inform dependents of a value change.

The example creates two objects (a WordCounter object and an Encoding object) in .NET and its stub objects in VisualWorks. We execute the function

```
System.Boolean CountStats (System.String pathname,
   System.Text.Encoding fileEncoding,
   out System.Int64 numLines&,
   out System.Int64 numWords&,
   out System.Int64 numChars&,
   out System.Int64 numBytes&)
```

through the wrapper method CountStats:with:with:with:with:with:. The function writes the results into the last four parameters. Therefore we have to create them as reference objects. Later we can get the values by sending the message value to a reference object.

The file **palm.log** contains 309 lines with 2206 words and 19295 bytes. The return value simply indicates that the parsing was successful.

DotNET also supports "inout" parameter. For those you can also use DotNETRefByObject, but you have to set them to a valid value, which will be passed into the function. For example:

```
num := DotNETRefByObjects new.
num value: 42.
Counter Add: num.
```

Using .NET Events and Delegates in VisualWorks

.NET objects can publish events. For example, the class Clock in the file **sample\EventClock.cs** publishes an event called SecondChangeHandler. If we have a VisualWorks object w1 that is interested in SecondChangeHandler events from an object c1 of the .NET class Clock, we can send this message to c1:

```
c1 upon: c1 OnSecondChange
   send: #receiveEvent:arguments:
   to: w1.
```

c1 OnSecondChange returns a symbol that identifies the event. It consists of the class name and the event name. #receiveEvent:arguments: is the selector of a method with two parameters that w1 must implement. Whenever c1 signals an OnSecondChange event, w1's method receiveEvent:arguments: is performed. A method that handles events always has two arguments, the first is the source of the event (e.g., in the case of c1), the second is always a subclass of EventArgs and provides additional data. To unsubscribe from event message send

```
c1 remove: w1 fromEventListFor: c1 OnSecondChnage.
```

See the class Watcher in the EventExamples parcel for a simple example that uses the **EventClock.dll**.

The class PanelTest in the EventExamples parcel opens a Windows Form with three TextBoxes and a Button and then registers for the ButtonClick event. The example uses several classes from the System.Windows.Forms assembly; the proxy for this assembly would be too big to include here.

Delegates

Delegates are objects that encapsulate a piece of code with a certain signature. To create a delegate, pass a block with the code to the constructor. The block must match the number and types of the arguments and evaluate to an object of the correct type. For example:

```
t := TimerCallback New: [:a | Transcript show: a].t Invoke: 'Hello World'.
   "prints 'Hello World' to the Transcript"
```

4

Debugging

Overview

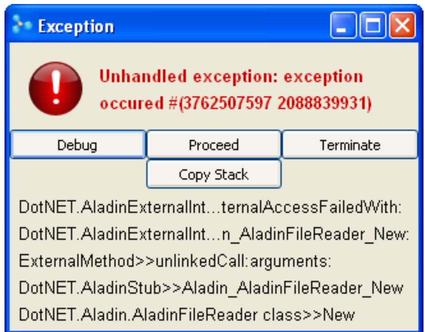
The DotNETConnect exception handler prevents methods in the stub that fail in execution from blocking VisualWorks. These errors are reported as instances of the VisualWorks class DotNETError, a subclass of Error. They can be caught and handled like normal VisualWorks errors. For debugging, they provide the string representation of the .NET exception, which includes an error message and a stack dump. Unfortunately, other useful information, such as the values of parameters and local variables, is not available.

This means, if the exception object does not provide enough information to fix the problem, you will need to use the VisualStudio debugger. The stack helps you to identify the method that failed, insert a break point in that proxy method, and step through it using the VisualStudio debugger.

This chapter describes a process for debugging proxy methods.

Debugging Errors While Loading the Proxy

If you encounter an exception like this:

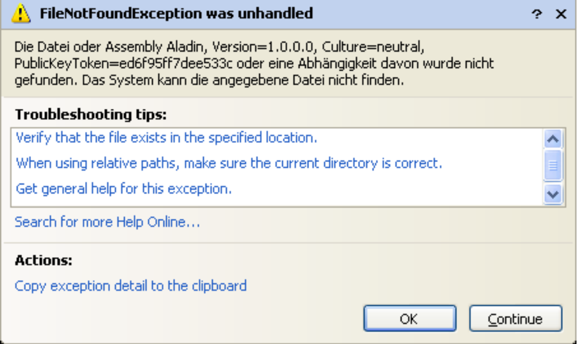


you can tell VisualWorks, not to handle these exception by executing

```
ObjectMemory registerObject: false
    withEngineFor: 'catchExternalCallErrors'.
```

Before retrying to load the assembly, be sure to have saved all your changes, as this may not possible anymore from the VisualStudio debugger.

Then, if you try to execute the code again, the exception will fall through to the VisualStudio debugger, which will provide a more informative error message:



Create a Debug DLL

To generate a Debug DLL, check **Generate Debug Code** in the Generate Code Step of the Wizard.

Connect VisualWorks and Visual Studio

To debug DotNETConnect application, Visual Studio must be running, and the VisualWorks image must be running and connected to the Visual Studio debugger. This procedure establishes these conditions.

1. Start Visual Studio.
2. Launch VisualWorks running an image with the DotNETConnect application accessing the assembly through the proxy.
3. In Visual Studio, open the Debug Process selector (**Debug > Processes**).
4. In the **Available Processes** list, find and select the VisualWorks process, and click **Attach...** .

The VisualWorks process is then added to the Visual Studio debugger, and you can begin stepping through the process.

Debugging a Proxy Method

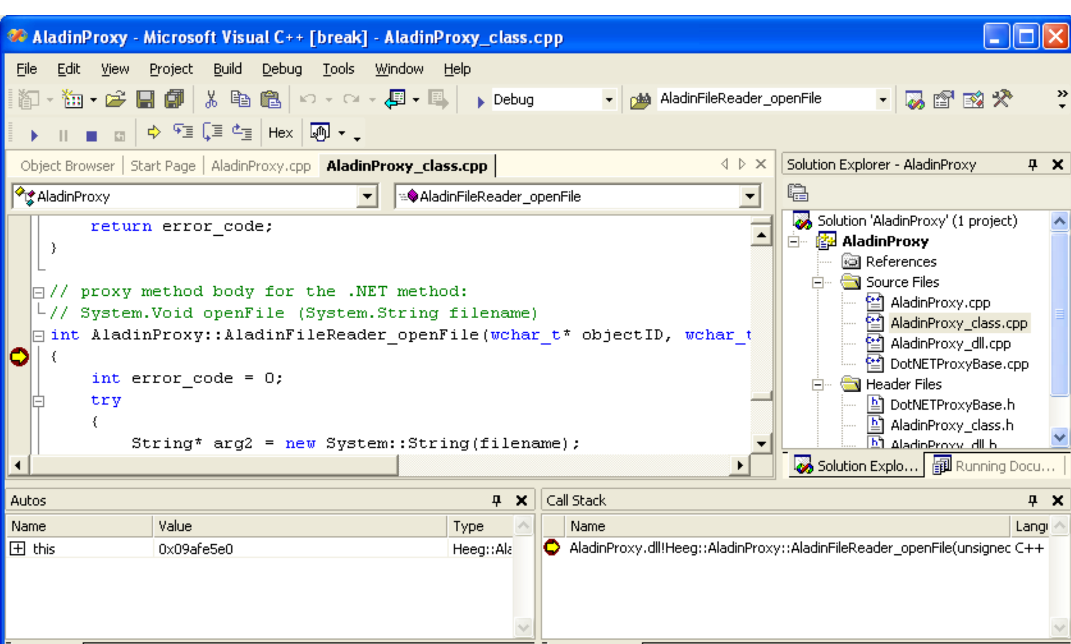
We illustrate debugging with a simple example that raises an exception. To raise an exception in using the Aladin assembly, evaluate:

```
reader := AladinFileReader New.
reader openFile: 'readme.txt'
```

When executing these statements the last statement raises an exception in VisualWorks. In the debugger, we identify the proxy method that raised the exception as AladinFileReader_openFile: objectID with: filename in AladinFileReader. This method is Smalltalk syntax, but the initial part of the selector, “AladinFileReader_openFile,” tells us what to look for in the source file.

In Visual Studio, view the source file (**AladinProxy_class.cpp**) and search for the method AladinFileReader_openFile. You can place a break point in the source file in Visual Studio (**Insert Breakpoint**), and then restart the method in the VisualWorks debugger.

The execution of the proxy method stops at the break point.



You can step through the method and check what caused the exception. This process is also useful if a method does not return an expected value.

5

Deploying a DotNET Application

Deploying an application containing DotNETConnect components is the same as for any other VisualWorks application, except that in is necessary to ensure that the application has access to the .NET common language runtime and the DotNETConnect DLLs.

When creating the runtime image, load the two DotNETConnect base components—DotNETConnectRuntime and DotNETBaseClasses—and the components created using DotNETConnect development. Then the developer can create a runtime image as usual.

The DLLs that are needed by the DotNETConnect base components (**`DotNETBase.dll`** and **`MscorlibProxy.dll`**) and the created proxy DLLs must be copied to the directory that contains the VisualWorks virtual machine. Also, the DLLs and/or executables that were integrated must either be installed in the .NET global assembly cache or copied to the directory that contains the VisualWorks virtual machine.

There are currently two options to start an image containing DotNETConnect components:

- Create a batch file that calls the .NET environment batch file and then start the image.
- Create environment settings for Windows using the data inside the .NET environment batch file. Then the image can be started as usual.

Index

Symbols
 2-1
A
Aladin DiveReader 1-4, 1-5
Aladin.DLL 1-6
AladinProxy.DLL 1-6
application delivery 5-1
argument type 3-4
assembly 1-2, 2-5
AssemblyDescription class 2-13
C
C++ code 2-9
check types 2-7
CLR, see "common language runtime"
code 2-9
Code Generator 1-7
code generator 2-9
common language runtime 1-1
common type system 1-1
compiler errors 2-15
compiling proxy code 2-14
configuration file 1-2
constructor method 3-4
create instances 3-4
CTS, see "common type system"
D
deployment 5-1
DLL 1-2, 1-3
DotNET.System namespace 1-12
DotNETBase.DLL 1-6
DotNETBaseClasses 1-5
DotNETConnect Base Classes 1-4
DotNETConnect Creation Wizard 2-1
DotNETConnect Development 1-4
DotNETConnect Runtime 1-4
DotNETConnectDevelopment 1-5
DotNETConnectInstaller 1-4
DotNETConnectRuntime 1-5
DotNETConnectTest.DLL 1-6
DotNETConnectTestProxy.DLL 1-6
DotNETProxyBase.cpp 2-15
DotNETProxyBase.h 2-15
DotNETReflection.DLL 1-6
E
environment variables 2-14
events 3-8
Example 1-4
EXE 1-2
F
files generated 2-10
G
garbage collection 1-2
generating proxy code 2-13, 2-14
global assembly cache 1-2
Global Assembly Cache Browser 1-8
I
icons 1-8
instance creation 3-4
instance creation method 3-4
Int16 3-4
Int32 3-4
L
Library Directory 2-6
lifecycle management 1-2
linker error 2-15
M
manifest 1-2
members 1-2
method signature 2-18
mixed DLL 1-3
MSCorLib 2-20
mscorlib 1-9
MscorlibProxy.DLL 1-6
N
name space 1-12
P
package 2-6
prerequisites 1-3
proxy classes 1-2
R
Random class 1-9

reflection API 1-2

S

sdkvars.bat 2-15
signature 1-2
signatures 2-18
Smalltalk code 2-9
Smalltalk source code 2-18
Smalltalk Source Code Directory 2-6
Source Code Directory 2-6
stub method 1-2

T

target directory 2-6
target package 2-6
tools 1-7
type conversion 3-6
types 1-2

V

validation check 2-7
verifying types 2-13
Visual Studio .NET 1-1
vsvars32.bat 2-14

W

WalkThru 1-10
web services 1-1
wizard 2-1