Accelerating J2EE to .NET Migrations With the JLCA Companion: A Sample Application

Introduction

The Java Language Conversion Assistant (JLCA), developed by ArtinSoft and included within Microsoft's Visual Studio, is an automated code migration product that accelerates the conversion of existing Java/J2EE applications into Microsoft Visual C# under .NET. ArtinSoft's new JLCA Companion allows users to extend the functionality delivered by the JCLA by adding their own transformation rules, also known as custom mappings, to the JLCA translation dictionary. These rules can either add support for new code elements or redefine existing transformations already bundled within the JLCA.

This document will illustrate the advantages of using the JLCA Companion in the process of converting a real Java application. We will first use the standard JLCA to transform the application, showing with code snippets the outcome of the process. We will then show how the transformation results are significantly improved by using the JLCA Companion – thus reducing your developing time and increasing your productivity.

Overview

Among the major complications of any migration project are handling third party components (TPC) or J2EE packages that the JLCA does not automatically convert. This paper will illustrate how the JLCA Companion enables you to extend the functionality of the JLCA version 3.0 to address these issues. To illustrate how the JLCA Companion works, we are going to extend the JLCA to enable automated migration of the java.util.zip package.

ZIP is one of the most common and well-known file formats. ZIP files are data containers that store one or more files in a compressed form. This format is widely used over the Internet because it saves both disk space and bandwidth. A compression rate as high as 90% can be achieved by using ZIP files. ZIP algorithms for compression and decompression are freely available in the zlib library (http://www.gzip.org/zlib).

The Java API specification includes support for the ZIP format. The java.util.zip package provides all the necessary classes to compress and decompress data. Currently, the Microsoft .NET Framework does not include equivalent classes for manipulating ZIP files. However, there are third-party products available to handle this need: The #ziplib (SharpZipLib) is a port of the zlib ZIP library, written entirely in C# for the .NET platform. Its license allows developers to include this library in commercial, closed-source applications. This product can be found here: http://www.icsharpcode.net/OpenSource/SharpZipLib/default.asp. You will need to download this library if you want to run the sample code.

In the first section of this document we will describe ZipUtil, which is a simple custom Java application that manages ZIP files. In the following section, we will show in detail the migration of this application to .NET using the JLCA. We will then illustrate the actual migration improvements gained by using the JLCA Companion. The final section compares the transformation both with and without the JLCA Companion and summarizes the key benefits.

To download the code of this sample, get the compressed "zip" file with the code

The ZipUtil application

ZipUtil is a working Java application used as an example in this paper to contrast the migration process using the JLCA alone and with the JLCA Companion. It is a simple Java command-line application for managing ZIP files. You can list the contents of a ZIP file, unzip the entries, or compress the files within a directory into a new ZIP file.

ZipUtil Usage

1. List the contents of a zip file: java ZipUitl /list filename.zip
2. Unzip a file: java ZipUitl /unzip filename.zip
3. Zip a directory: java ZipUitl /zip directory

Note: You need JDK 1.3.x or later in order to run this application.

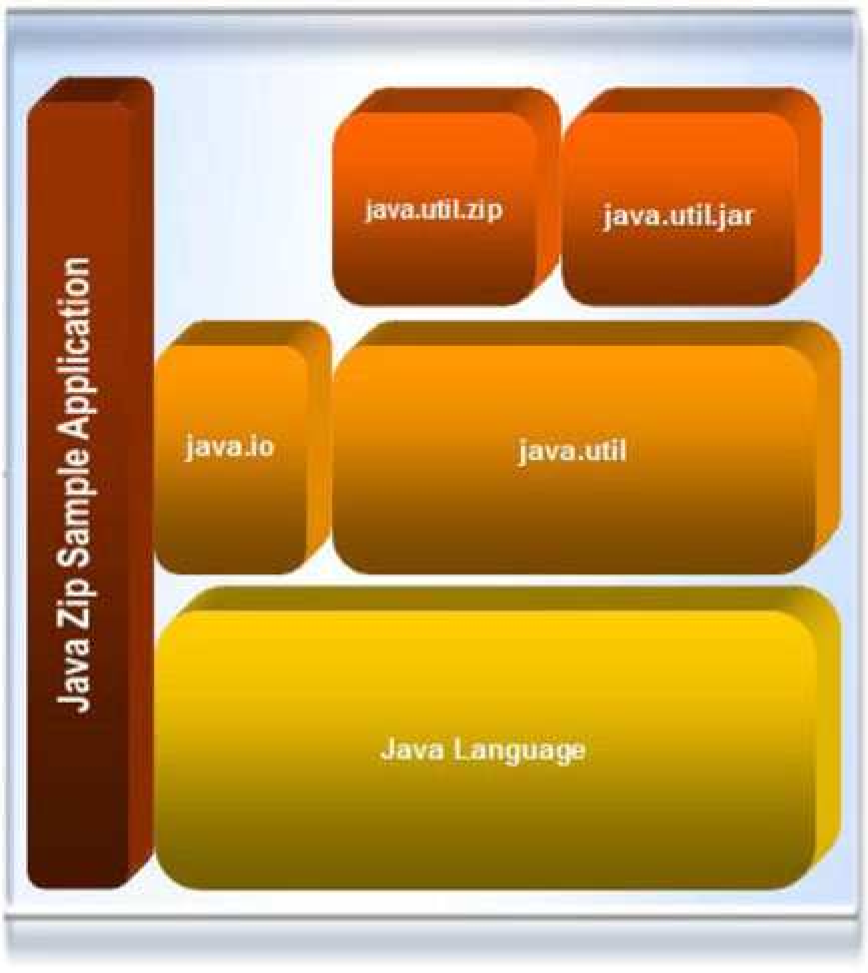
The Java source code for ZipUtil can be found in the folder "Zip Sample/Source Code" inside the "Zip Sample.zip" file.

The application contains two classes encapsulating the zip functionality:

- The ZipFileExtended class extends java.util.zip.ZipFile. The method toString() was overloaded so it returns the contents of the file. The method unZip() was added for performing the full file decompression. It creates the required directories and files.
- The ZipOutExtended class extends java.util.zip.ZipOutputStream. This class overloads the constructor; it receives a directory name and creates a ZIP file using the directory name. The zipDir() method was added to compress the directory represented by the current object.

Application architecture

Below is a diagram of the ZipUtil application architecture. The base packages used are java.io and java.util. In addition, the java.util sub packages java.util.zip and java.util.jar are core elements of the Zip Sample.



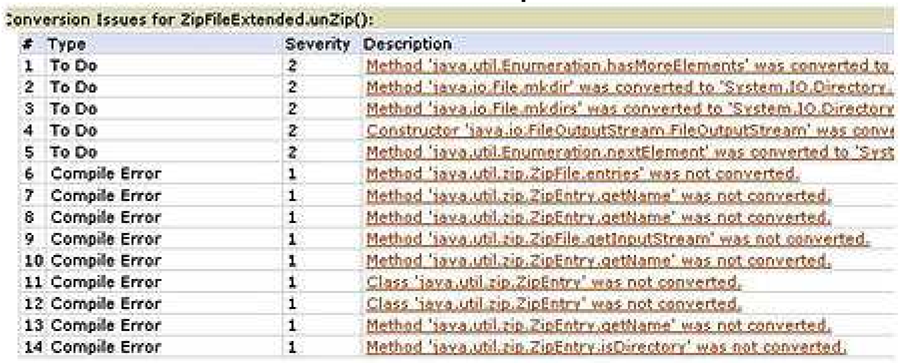
Conversion using the JLCA

We will now undertake conversion of the ZipUtil Java application to C# using the Java Language Conversion Assistant (JLCA) included within Microsoft Visual Studio.NET. By illustrating the transformation process using the key elements class, constructor, method and property, you will notice that there is a need for manual user intervention in order to make the code fully functional.

Migrating with the JLCA alone successfully converts all language elements, and most of the java.io elements used in the application. However, since there is no native support for ZIP in .NET, the JLCA is unable to perform code transformations for classes using the java.util.zip package. This means that you will need to work around such code to achieve functional equivalence.

The ZIP elements are reported as “compile errors”. For example, here is the conversion report produced by the JLCA for the class ZipFileExtended:

Conversion Report



:onversion Issues for ZipFileExtended.unZip():

#	Type	Severity	Description
1	To Do	2	Method 'java.util.Enumeration.hasMoreElements' was converted to
2	To Do	2	Method 'java.io.File.mkdir' was converted to 'System.IO.Directory.
3	To Do	2	Method 'java.io.File.mkdirs' was converted to 'System.IO.Directory
4	To Do	2	Constructor 'java.io.FileOutputStream.FileOutputStream' was conve
5	To Do	2	Method 'java.util.Enumeration.nextElement' was converted to 'Syst
6	Compile Error	1	Method 'java.util.zip.ZipFile.entries' was not converted.
7	Compile Error	1	Method 'java.util.zip.ZipEntry.getName' was not converted.
8	Compile Error	1	Method 'java.util.zip.ZipEntry.getName' was not converted.
9	Compile Error	1	Method 'java.util.zip.ZipFile.getInputStream' was not converted.
10	Compile Error	1	Method 'java.util.zip.ZipEntry.getName' was not converted.
11	Compile Error	1	Class 'java.util.zip.ZipEntry' was not converted.
12	Compile Error	1	Class 'java.util.zip.ZipEntry' was not converted.
13	Compile Error	1	Method 'java.util.zip.ZipEntry.getName' was not converted.
14	Compile Error	1	Method 'java.util.zip.ZipEntry.isDirectory' was not converted.

Conversion Report

Each of these issues would require manual changes before the migrated application will work – a process that can require considerable time and effort, and is error-prone. By adding the JLCA Companion, many of these errors can be automatically eliminated.

Class Migration

The following example shows how the JLCA converts the java.util.zip.ZipFile.

Java Code

```
public class ZipFileExtended extends java.util.zip.ZipFile{
}
```

JLCA Generated C# Code

```
//UPGRADE_ISSUE: Class 'java.util.zip.ZipFile'
//was not converted.

public class ZipFileExtended:java.util.zip.ZipFile{
}
```

Because the class is not supported by the JLCA, it is not migrated and a message is generated.

Constructor Transformation

This sample shows how the JLCA converts the ZipEntry constructor:

Java Code

```
new ZipEntry(path);
```

JLCA Generated C# Code

```
//UPGRADE_ISSUE:
//Constructor 'java.util.zip.ZipEntry.ZipEntry'
//was not converted.

new ZipEntry(path);
```

The constructor is not transformed by the JLCA; you would need to work around the code in order to obtain functional equivalence.

Method conversion

Here is code that illustrates how the JLCA processes the putNextEntry method:

Java Code

```
this.putNextEntry(new ZipEntry(path));
```

JLCA Generated C# Code

```
//UPGRADE_ISSUE: Method 'java.util.zip.ZipOutputStream.putNextEntry'
//was not converted.
//UPGRADE_ISSUE: Constructor 'java.util.zip.ZipEntry.ZipEntry'
//was not converted.

this.putNextEntry(new ZipEntry(path));
```

The method is not migrated as there is no support for ZIP in .NET. You would need to manually alter the code in order to make it work.

Conversion using the JLCA Companion The JLCA Companion can significantly improve the transformation results of the JLCA conversion process. As a consequence, you are able to reduce your development time and increase your productivity. We will now show the translation of the ZipUtil Java application using the JLCA Companion. By illustrating the transformation process using the key elements class, constructor, method and property, you will see that the resulting code is fully functional and does not require additional user manipulation. In a typical JLCA Companion project, which behaves like any other project inside the Visual Studio.NET IDE, you can specify classes, methods and field conversions. In addition, you can specify error messages and references to different components. This is accomplished by using Map Files and Definition Files, which are items inside this type of project.

Field translation

This code shows how the JLCA converts getName to a property:

Java Code

```
ZipEntry e = ((ZipEntry)entries.nextElement());
String fileName = e.getName();
```

JLCA Generated C# Code

```
//UPGRADE_ISSUE: Class 'java.util.zip.ZipEntry' was not converted.
//UPGRADE_TODO: Method 'java.util.Enumeration.nextElement'
//was converted to 'System.Collections.IEnumerator.Current'
// which has a different behavior.
```

```
ZipEntry e = ((ZipEntry) entries.Current);

//UPGRADE_ISSUE: Method 'java.util.zip.ZipEntry.getName' was not converted.

System.String fileName = e.getName();
```

As before, because there is no support for getName inside the JLCA, you would need to manually modify the code in order to make it work.

Conversion using the JLCA Companion

The JLCA Companion can significantly improve the transformation results of the JLCA conversion process. As a consequence, you are able to reduce your development time and increase your productivity.

We will now show the translation of the ZipUtil Java application using the JLCA Companion. By illustrating the transformation process using the key elements class, constructor, method and property, you will see that the resulting code is fully functional and does not require additional user manipulation. In a typical JLCA Companion project, which behaves like any other project inside the Visual Studio.NET IDE, you can specify classes, methods and field conversions.

In addition, you can specify error messages and references to different components. This is accomplished by using Map Files and Definition Files, which are items inside this type of project.

Map Files

Contain the equivalences or mappings that are to be applied in a conversion. The structure of a Map File has three main sections. The first is used to import your Definition Files (.def). The core of this file is the one enclosed by the package and endpackagekeywords. The last section is related to class definitions. These definitions are enclosed by the class and endclass keywords. You can specify as many class definitions as required. Class definitions contain items such as constructors, properties/fields and methods.

Definition Files

Contain elements that are to be referenced by a Map File. Inside these elements, you define messages and references to ActiveX components, COM objects and Assemblies.

Nearly 100% of the Java ZIP classes can be migrated to use the third-party #ziplib library. By using the JLCA Companion, you can specify a mapping that will automatically convert Java ZIP classes to the #ziplib library to achieve ZIP functionality. Next, we will learn how to use the JLCA Companion in order to extend the migration coverage of the JLCA.

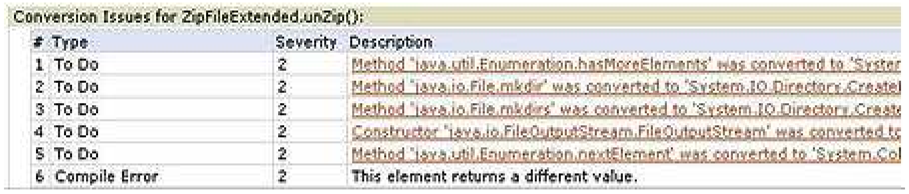
The ZipMaps Project

ZipMaps is a JLCA Companion project containing user-defined transformation rules, also called mappings, for the java.util.zip and java.util.jar packages. (You can find the source code for the ZipMaps project inside the “Zip Sample.zip” file, within the folder “Zip Sample\ZipMaps”.) These mappings are patterns that define how to convert one Java object to a C# object. Within a JLCA Companion project you will find Map Files and Definition Files. Since both the Java and C# versions implement the same library (zlib), we can map almost all the members using the JLCA Companion.

The files java.util.zip.emap and java.util.jar.emap contain all Java members and the corresponding #ziplib equivalences.

Looking at the new conversion report, you can see that the ZIP-related compile errors have now been resolved:

Conversion Report



Conversion Issues for ZipFileExtended.unZip():

#	Type	Severity	Description
1	To Do	2	Method 'java.util.Enumeration.hasMoreElements' was converted to 'Syster
2	To Do	2	Method 'java.io.File.mkdir' was converted to 'System.IO.Directory.Createl
3	To Do	2	Method 'java.io.File.mkdirs' was converted to 'System.IO.Directory.Create
4	To Do	2	Constructor 'java.io.FileOutputStream.FileOutputStream' was converted tc
5	To Do	2	Method 'java.util.Enumeration.nextElement' was converted to 'System.Col
6	Compile Error	2	This element returns a different value.

Conversion Report

You will find the converted application in the folder "Zip Sample\Migrated Code\Using Companion", and can open the project to see the readability of the converted code. If you compile and build the project, you will be able to run the ZipUtil .NET version. We now have all the functionality of the original Java application within a .NET executable. Almost no manual changes were needed, and usage of the application is identical to the Java version.

There are several important concepts you need to understand when creating member mappings (constructors, methods and fields). Transformation rules are created with a structure that uses patterns, which are not subject to a specific string value, but rather the form of the mold. Inside them you use Binding Names as placeholders. They are visual representations of elements to be transformed. This work scheme allows portions of the source expression to be manipulated individually. The following are placeholders used in this sample: a, b and p. In other words, the string values used for a, b and parameter identifiers p, p1...pn represent pattern variables.

Class conversion

A class map is defined using the class keyword, used to define the start of a class scope of member mappings (constructors, methods and fields), and endclass, which indicates the end of a class (end of a class scope of mappings). In addition to these keywords, -> is used to indicate correspondence. The following example shows how the JLCA converts the java.util.zip.ZipFile using the mapping created with the JLCA Companion:

Java Code

```
public class ZipFileExtended extends java.util.zip.ZipFile{
}
```

JLCA with JLCA Companion

Map Code

```
class ZipInputStream -> ICSharpCode.SharpZipLib.Zip.ZipInputStream
...
endclass
```

C# Code

```
public class ZipFileExtended:ICSharpCode.SharpZipLib.Zip.ZipFile
{
}
```

Using the JLCA Companion, the class java.util.zip.ZipFile will be automatically converted to ICSharpCode.SharpZipLib.Zip.ZipFile whenever a reference to it is found in your code. The class conversion is performed flawlessly, there is no need for manual intervention in the code; and it will compile and achieve functional equivalence.

Constructor transformation

You create a constructor map using the member keyword, which is used to define the start of a member scope; and endmember, which is used to indicate the end of the member mapping. In addition to these keywords, -> is used to indicate correspondence.

This following example shows how the JLCA processes the ZipEntry constructor using the mapping created with the JLCA Companion:

Java Code

```
new ZipEntry(path);
```

JLCA with JLCA Companion

Map Code

```
member ZipEntry
b(p:java.util.zip.ZipEntry) -> new ICSharpCode.SharpZipLib.Zip.ZipEntry(p);
b(p:java.lang.String) -> new ICSharpCode.SharpZipLib.Zip.ZipEntry(p);
endmember
```

The following are the matching values of the code with their pattern Binding Variables:

b matches ZipEntry: since it is the element we are mapping, there is no need to use it in the mapping body. p matches path: the match here is performed on the second overload since path is a java.lang.String type.

C# Code

```
new ICSharpCode.SharpZipLib.Zip.ZipEntry(path);
```

The constructor ZipEntry will be automatically converted to the constructor ICSharpCode.SharpZipLib.Zip.ZipEntry every time it is found in your code. In addition, the object path replaces the Binding Name p since it functions as a placeholder. The constructor transformation is performed perfectly; there is no need to work around the code as it compiles and achieves functional equivalence.

Method migration

A method map is created using the member keyword, which is used to delineate the start of a member scope, and endmember, which is used to define the end of the member mapping. In addition to these keywords, -> is used to indicate correspondence.

The following example illustrates how the JLCA processes the putNextEntry method using the mapping created with the JLCA Companion:

Java Code

```
this.putNextEntry(new ZipEntry(path));
```

JLCA with JLCA Companion

Map Code

```
member putNextEntry
a.b(p:java.util.zip.ZipEntry) ->
a.PutNextEntry(p);
endmember
```

C# Code

```
this.PutNextEntry(new ICSharpCode.SharpZipLib.Zip.ZipEntry(path));
```

The method putNextEntry will be automatically converted to the method PutNextEntry whenever a call to the Java method is found in your code. It is converted perfectly; the code will not generate compile errors.

Field translation

These types of map definitions consist of two parts: getvalue and setvalue. A field map is defined with the member keyword, used to define the start of a member scope, and endmember to indicate the end of the member mapping. In addition, -> is used to indicate correspondence.

The following code illustrates the JLCA converting the member getName and setName to the property Name using the mapping created with the JLCA Companion:

Java Code

```
ZipEntry e = ((ZipEntry)entries.nextElement());
String fileName = e.getName();
```

JLCA with JLCA Companion

Map Code

```
member getName
a.b() -> a.Name;
endmember
...
member setName
a.b() = c -> a.Name = c;
endmember
```

The following are the matching values of the code with their pattern Binding Variables:

a matches e: here we have access to the object instance whose method is being converted by the a placeholder. b matches getName: since it is the element we are mapping, there is no need to use it in the mapping body.

C# Code

```
//UPGRADE_TODO: Method 'java.util.Enumeration.nextElement' was converted to
'System.Collections.IEnumerator.Current' which has a different behavior.

ICSharpCode.SharpZipLib.Zip.ZipEntry e = ((ICSharpCode.SharpZipLib.Zip.ZipEntry)
entries.Current);
System.String fileName = e.Name;
```

getName will be automatically converted to the property Name whenever it is used in your Java code. The conversion leaves no errors so there is no need for any manual coding.

EWI Processing

An EWI (Error, Warning, and Issue) is a message that is printed in the target code. For example, if there is a different behavior in a method and there is a need to perform some manual changes you could print an EWI as a Warning to remind the user there is work to be done. The definition language allows the declaration of EWIs for usage in map definitions. The keyword ewi is used to create an EWI. This code shows how to process EWIs with the JLCA Companion:

Java Code

```
for (Enumeration entries = this.entries(); entries.hasMoreElements();) {
...
}
```

JLCA with JLCA Companion

Code inside definition file

```
ewi DIFF_RETURN_VAL : " CUSTOM_TODO: This element returns a different value.";
```

Map Code

```
using "declarations.def";
...
member entries
a.b() -> begin
printwarning(DIFF_RETURN_VAL);
a.GetEnumerator();
end
endmember ...
```

C# Code

```
//CUSTOM_TODO: This element returns a different value.
//UPGRADE_TODO: Method 'java.util.Enumeration.hasMoreElements' was converted to
'System.Collections.IEnumerator.MoveNext' which has a different behavior.

for (System.Collections.IEnumerator entries = this.GetEnumerator(); entries.MoveNext(); ){
...
}
```

The conversion process issues the EWI whenever the JLCA encounters the element where the EWI was defined. This means that every time the member entries are converted the EWI DIFF_RETURN_VAL will be printed.

References manipulation

It is possible to define references to external .NET assemblies to be added to the target language of the migrated project. Below is a case of an Assembly definition inside a definition file (in this sample is called declarations.def):

JLCA with JLCA Companion

Code inside definition file

```
reference SHARPZIPLIB : "ICSharpCode.SharpZipLib";
```

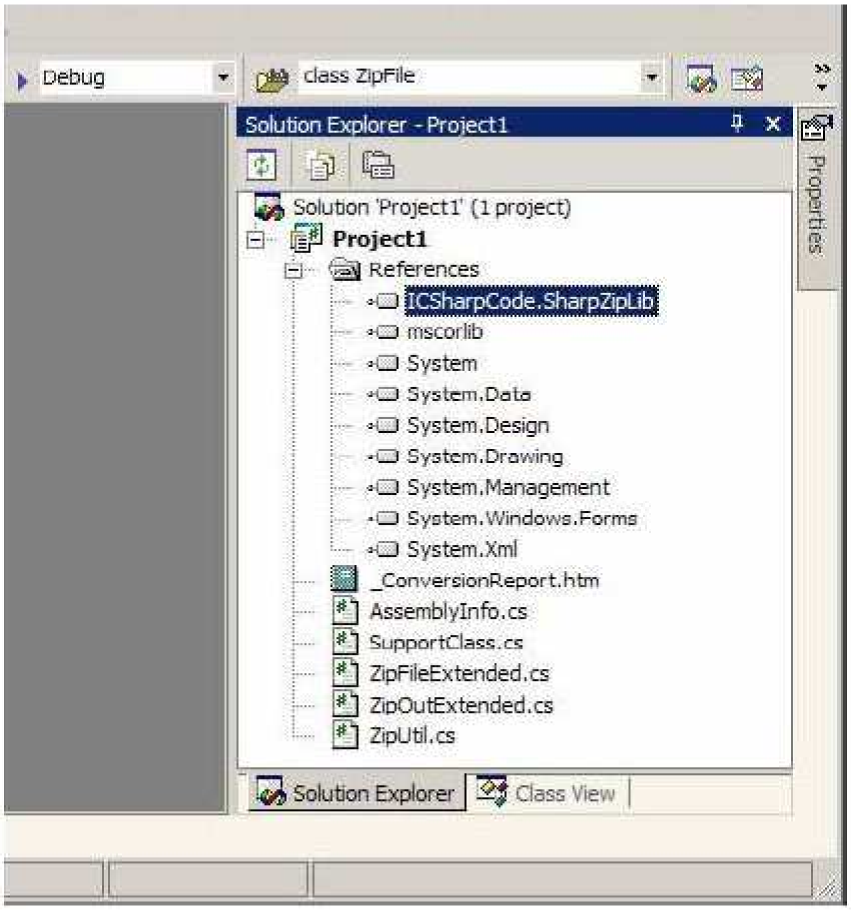
Map Code

```
using "declarations.def";

package java.util.jar
addref SHARPZIPLIB;
...
endpackage
```

When a reference is found in your mappings the JLCA Companion will add it to the converted project References Section, as shown below:

References Section

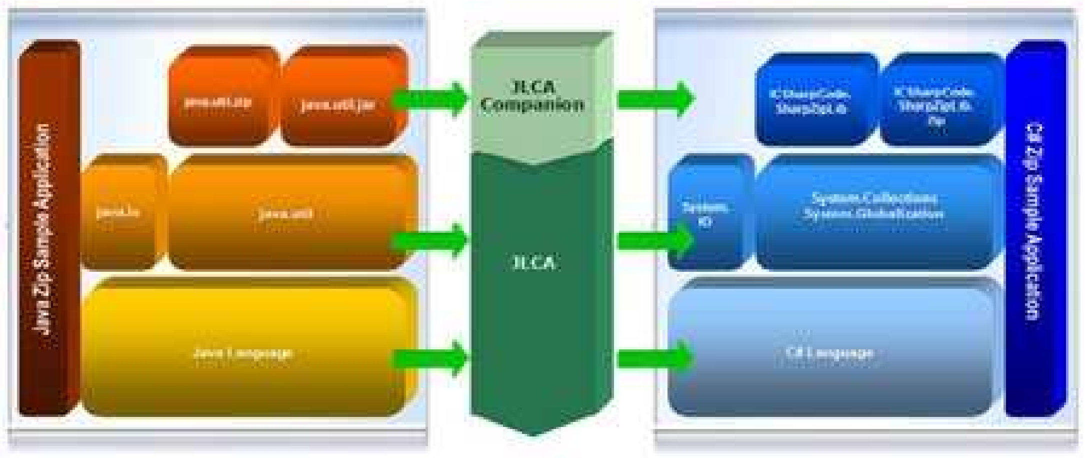


Solution 'Project1' (1 project)
Project1
References
ICSharpCode.SharpZipLib
mscorlib
System
System.Data
System.Design
System.Drawing
System.Management
System.Windows.Forms
System.Xml
_ConversionReport.htm
AssemblyInfo.cs
SupportClass.cs
ZipFileExtended.cs
ZipOutExtended.cs
ZipUtil.cs

Comparison of transformations

Here are the contrasting results of the two conversion processes: one using only the base JLCA, and the other using the JLCA with ArtinSoft's JLCA Companion.

The following diagram shows the migration course of action:



As shown, all elements inside the Java Language Box are translated to elements inside the C# Language Box using the JLCA. In addition, the items inside the java.io and java.util packages are also converted using also the base JLCA. However, the java.util sub packages java.util.jar and java.util.zip are transformed with the help of the JLCA Companion.

Class conversion result

The following diagram illustrates the behavior of the JLCA with and without the JLCA Companion when converting Classes:

Java Code

public class ZipFileExtended extends java.util.zip.ZipFile{

}

JLCA C# Code

//UPGRADE_ISSUE: Class 'java.util.zip.ZipFile' was not converted.

public class ZipFileExtended :

java.util.zip.ZipFile{

}

The C# code generated by the base JLCA contains compile errors and requires manual user intervention.

JLCA with JLCA Companion C# Code

```
public class ZipFileExtended : ICSharpCode.SharpZipLib.Zip.ZipFile{
}
```

The C# code generated by the JLCA with the JLCA Companion is flawless and does not require manual intervention.

Constructor transformation outcome

The following extract illustrates the performance of the conversion tool with and without the JLCA Companion when migrating constructors:

Java Code

```
new ZipEntry(path);
```

JLCA C# Code

```
//UPGRADE_ISSUE: Constructor 'java.util.zip.ZipEntry.ZipEntry' was not converted.
new ZipEntry(path);
```

The JLCA C# Code contains build errors and requires manual changes.

JLCA with JLCA Companion C# Code

```
new ICSharpCode.SharpZipLib.Zip.ZipEntry(path);
```

The JLCA with JLCA Companion C# Code is error-free and functionally equivalent.

Method migration comparison

Next we will compare the actions of the JLCA with and without the JLCA Companion when translating methods:

Java Code

```
this.putNextEntry(new ZipEntry(path));
```

JLCA C# Code

```
//UPGRADE_ISSUE: Method 'java.util.zip.ZipOutputStream.putNextEntry' was not converted.
//UPGRADE_ISSUE: Constructor 'java.util.zip.ZipEntry.ZipEntry' was not converted.
this.putNextEntry(new ZipEntry(path));
```

The JLCA C# Code contains errors and requires a manual workaround.

JLCA with JLCA Companion C# Code

```
this.PutNextEntry(new ICSharpCode.SharpZipLib.Zip.ZipEntry(path));
```

The JLCA with JLCA Companion C# Code is functionally equivalent to the source code.

Field translation end result

The following example demonstrates the process of the JLCA conversion tool with and without the JLCA Companion when converting C# properties.

Java Code

```
ZipEntry e = ((ZipEntry)entries.nextElement());
String fileName = e.getName();
```

JLCA C# Code

```
//UPGRADE_ISSUE: Class 'java.util.zip.ZipEntry' was not converted.
//UPGRADE_TODO: Method 'java.util.Enumeration.nextElement' was converted to
'System.Collections.IEnumerator.Current' which has a different behavior.

ZipEntry e = ((ZipEntry) entries.Current);

//UPGRADE_ISSUE: Method 'java.util.zip.ZipEntry.getName' was not converted.
System.String fileName = e.getName();
```

The JLCA C# Code was transformed with major correspondent issues.

JLCA with JLCA Companion C# Code

```
//UPGRADE_TODO: Method 'java.util.Enumeration.nextElement' was converted to
'System.Collections.IEnumerator.Current' which has a different behavior.

ICSharpCode.SharpZipLib.Zip.ZipEntry e = ((ICSharpCode.SharpZipLib.Zip.ZipEntry)
entries.Current);
System.String fileName = e.Name;
```

The code generated by JLCA with the JLCA Companion achieves functionality.

Summary

The JLCA Companion provides a powerful mechanism for extending the Java Language Conversion Assistant coverage, as shown in the previous section. It also enables the generation of cleaner code and reduces the number of messages displayed within the code. Finally, the JLCA Companion allows you to use third-party libraries, or even your own implementations for migrating Java packages with no equivalence in the .NET platform. This capability will help you speed the process of moving real world Java applications to .NET.

The JLCA Companion projects you create can be used in later migration processes, saving time and providing clean and clear target code.

Furthermore, using the JLCA Companion will increase your productivity and reduce the time required to convert an application as most of the work is done automatically.