UniSharping: C# code converter

# Introduction

**UniSharping** is a **C#** code converter for other programming languages: **Java**, **Python**, **PHP**, etc. The distinguishing feature from other converters is that the resulting code is workable without any manual editing. But you have to fix the C# source code a bit. That is, we focus not on the task of one-time migration, but on the continuation of development in C# with the receipt at any moment of the working code in the desired language, that is to extend the integration capabilities of the project.

This ambitious task can be solved in principle if the source code satisfies some limitations related to language constructs, system libraries and technologies. This limited subset is called the **U# (Universal Sharp)**.

For the purposes of cross, Microsoft has already made a limitation of the .Net Framework in terms of libraries and technologies: .Net Core. This is as if the first step in the right direction, U# takes the second step to "cross-programming".

Limitations of U# in language constructs appeared a bit-it is atavisms goto and case goto, and also yield (for Java), not simulated adequately in automatic mode. It is not recommended (although it is possible) to use struct, there are nuances with names-all this is described in detail in a separate document. The U# parser generates errors and warnings, and to ensure correct generation, you should correct the C# source code so that they will ideally disappear altogether. If you still want to keep the original version, you can use preprocessor directives #if JAVA || PHP... #else... #endif. These restrictions apply at the level of the U# engine and are not subject to external correction, as is the list of supported languages.

But limitations at the level of system libraries are set not hard and configured from the outside through special text files, which define how to translate a class and its members into the corresponding language. If there is a direct analogue, it and it is specified, if the situation is more complicated, it is written or a fragment of the code of the target language, or in general a special (service) class that solves the necessary task. In very difficult cases it is necessary to "hardcode" at the level of the engine, but such situations are quite rare (with a dozen). The order of configuration on system classes and their members are described in a separate document.

As for technologies[[1]](#footnote-1), here the list is limited at the level of the engine console application and Unit-tests (UnitTest)[[2]](#footnote-2). Well, individual Lib projects, as a special case, are translated into the appropriate constructs of the desired language.

For a successful translation, the original C# (solution) project must have some startup part that verifies the health of the original C#. Well, if it is an extensive system of auto-tests (standard UnitTest in different implementations or his), but at least there should be a console application, which at startup without any user intervention works correctly. The need for this is obvious-after the generation of the target language you can immediately check the performance. Ideally, all tests should work in the same way as C#.

# Converter implementation

The UniSharping Converter consists of:

* UniSharping.exe - console application for batch processing;
* UniSharping.Studio.exe – application with user GUI for configuration and conversion;
* Text files settings for system classes and their members.

The converter is developed in C# (naturally), but can transform its code into a supported language (its console application), which is one of the quality assurance procedures. The .Net Framework 4 or higher is required to run, and the other platforms are Mono for the console application.

Settings files should be located in the same directory where the executable modules are located, in the following folders:

* C# - for text files settings, and its internal folder system role does not play, they are just for structuring, and the files are extracted all with a txt extension;
* Java – Java service classes that are used to model some C# system classes that are not in Java. When the code is generated, the contents of that folder are placed in the UNISHARP package in the resulting directory. Special mention should be made of the Utils class – Collection of static methods "for different cases of life";
* Python, PHP … – similar to other languages;

Executable modules, settings and documentation are placed on the [www.unisharping.ru](http://www.unisharping.ru) and <https://github.com/konstantin-smith/UniSharping>.

The converter was tested on a real linguistic text processing project ([www.pullenti.ru/DownloadPage.aspx](http://www.pullenti.ru/DownloadPage.aspx) , there is an example of the result of his work) and a few inner projects.

# Conversion technology

Source code sources can be:

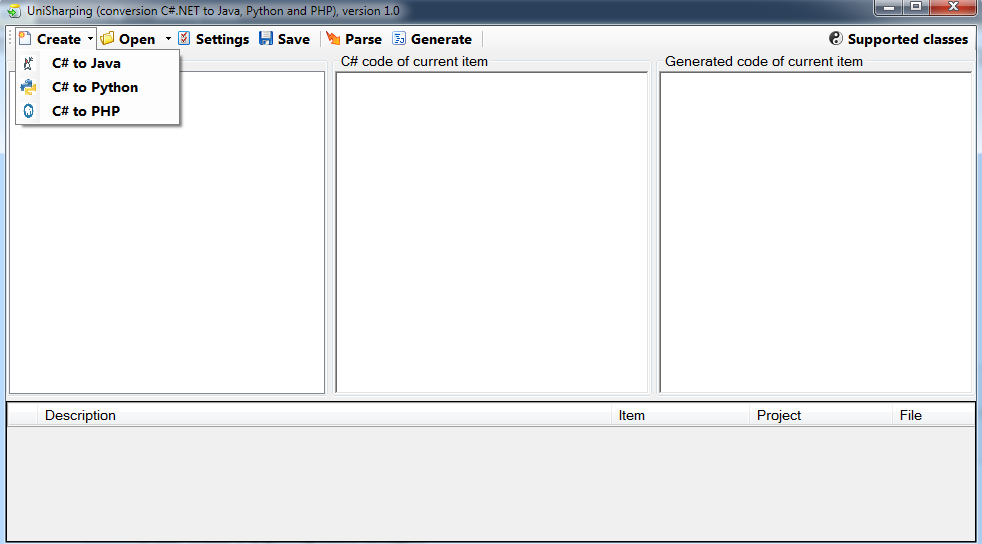
* One or more cs-files;
* One or more projects (csproj) of types (OutputType):
  + Library
  + Exe
* Solution (sln), from which all projects of supported types are taken.

It is understood as the "old" format of the Framework projects as well as the new one for .Net Core.

The projects take not only the “Build Action” files of type "Compile", but also the "Embedded resource", as well as the resource files from resources.resx, and they are drawn up as analogues in the finite programming system.

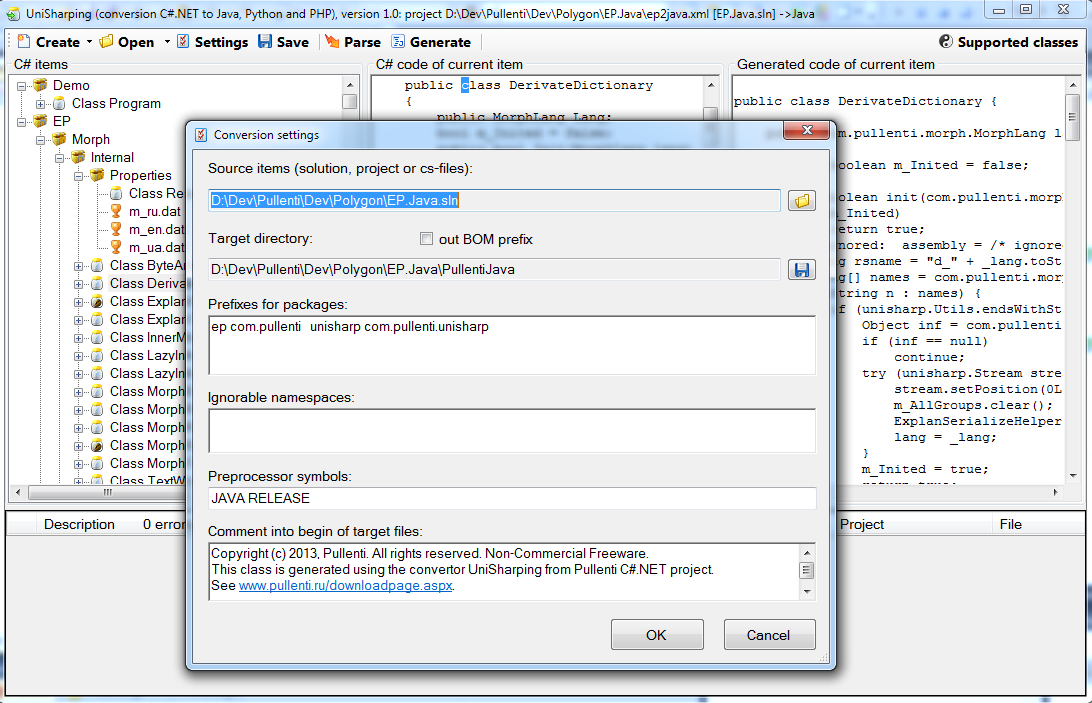
The necessary (but not sufficient) condition for successful conversion is the absence of errors in all convertible sources, they should be compiled in Visual Studio.

All references to the source modules, as well as a number of settings are stored in the so-called conversion file (CF), which is formatted as XML (described below). For its creation and further work it is offered to use UniSharping.Studio:



In the offered dialog window it is possible to set the necessary settings, and in the future to open this file for conversion. In CF the resulting language is strictly set, it can not be changed in the future except for the consciousness of the new CF.

Basically, the settings window depends on the resulting language, but many items are general. Here's what the CF for Java config window looks like:



* Target directory – directory where the files will be generated when the "Generate" button is pressed;
* Out BOM prefix – encoding of the resulting UTF-8 files, whether to display the initial prefix EF BB BF or not;
* Prefixes for some packages can be renamed, and here the match «namespace .NET» - «Java package». For example, in the picture "EP" will be replaced by "com.pullenti", and its internal namespace is similar to: "EP.Inner"-> "com.pullenti.inner" etc.
* List of ignored namespaces, from which classes will be ignored during analysis;
* Preprocessor symbols that are accounted for by the converter, allowing the user to correct the C# source code for the conversion task by turning off, for example, some code snippets;
* Comment inserted at the beginning of each generated file;

Here's how it's presented in the XML file:

<?xml version="1.0" encoding="UTF-8"?>

[<project version="**1.0**">](file:///D:\Dev\Pullenti\Dev\Polygon\EP.Java\ep2java.xml)

<source>../EP.Java.sln</source>

<target>../../Java/PullentiJava/src</target>

<lang>Java</lang>

<comment> Here is a comment at the beginning of files …</comment>

<packagepref target ="**com.pullenti**" net="**ep**"/>

<packagepref target="**com.pullenti.unisharp**" net="**unisharp**"/>

<condition>JAVA</condition>

<condition>RELEASE</condition>

</project>

As you can see, all the paths are stored relative to the current CF which allows you to move it along with the original data. By the way, the elements <source> can be several.

The next step is to correct the C# source code so that the parsing (Parse button) goes without errors and, if possible, without warnings. This stage is devoted to a separate document, where the related issues are discussed in detail.

Generation of the result code is made by the button generate. If there are errors in the parsing, the resulting code will also have errors in the appropriate locations. But if everything is done correctly, the resulting code will be executed in the same way as in the original C#.

It is assumed that the project is already in the target language – initially created, albeit empty. The generator only writes class files and resource files, and itself creates the necessary structure of folders for modeling packages for Java and Python. Most Ides have the ability to dynamically load files from folders and subfolders into a project, such as Eclipse by pressing F5. By the way, the generator also deletes unnecessary files and folders that were made during the previous generation, and in the new version because of renaming and correction of the original C# disappeared.

Also note that the files are generated in UTF-8 encoding. However, in the same Eclipse by default can stand the current encoding of Windows, and for the project is necessary in the settings (resources\text file encoding\other) to put UTF-8[[3]](#footnote-3).

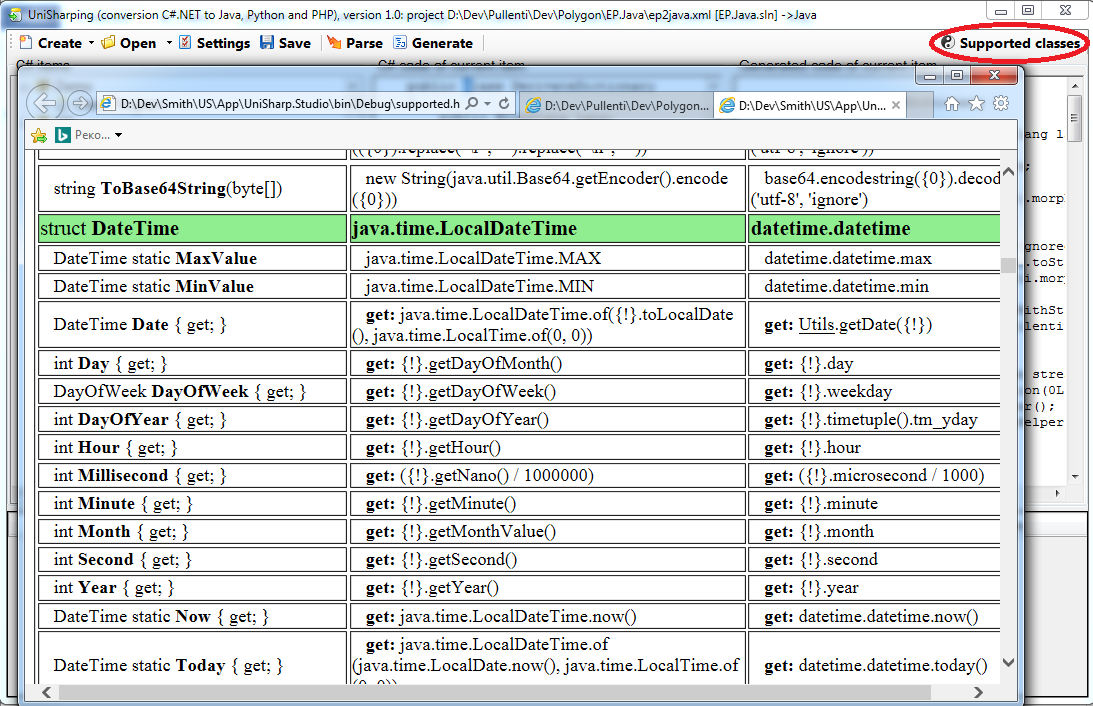
Thus, after the creation of CF (conversion file), correction of the original C# and creation of the final project the debugged conversion process can be used continuously, as the product evolves in the original C#, quickly generating versions in the target language. You can use a console application for this purpose, such as a embedding, for example, in a release generation process.

An XML conversion file is fed to the UniSharping console application. In the case of errors or warnings, the corresponding lines are written to the messages.csv file in the same directory as CF, but errors are better analyzed in Studio, where there is a system of navigation and error communication with places in the source code.

# Converter extensions

The correspondence between the C# system classes and methods and the target language is specified in the text files of the C# subdirectory where the converter starts. The addition of the corresponding files entails their support after restarting.

The list of current classes in the target languages can be obtained by clicking the "Supported Classes" button in the Studio:



The generated HTML with the current state will be displayed in the browser, in the left column of the C# element, further analogue Java, Python, etc. Here's how this information is presented in the file (reduced option):

struct System.DateTime

java java.time.LocalDateTime

python datetime.datetime

property get int Year

java {!}.getYear()

python {!}.year

property get DayOfWeek DayOfWeek

java {!}.getDayOfWeek()

python {!}.weekday

property static get DateTime Now

java {\*}.now()

python {\*}.now()

property static get DateTime Today

java java.time.LocalDateTime.of(java.time.LocalDate.now(), java.time.LocalTime.of(0, 0))

python {\*}.today()

.ctor(\*)

java java.time.LocalDateTime.of({0=1}, {1=1}, {2=1}, {3=0}, {4=0}, {5=0})

python {\*}({0=1}, {1=1}, {2=1}, {3=0}, {4=0}, {5=0})

method DateTime AddYears(int)

java =plusYears

python ({!} + datetime.timedelta(days={0}\*365))

The setup is described in detail in a separate document.

Attention! If you have found an error or have tuned into new methods or classes, please share this with us to insert into the next release.

1. This refers to end-application technologies such as Console, WinForm, WPF, ASP, and not intermediate types of WCF, XML, Entity, which are considered system libraries. [↑](#footnote-ref-1)
2. Perhaps in the future there will be something for the Web. [↑](#footnote-ref-2)
3. If you specify "out BOM prefix" in the CF config, you do not need to do this in Eclipse, but some other Java distribution systems are arguing without understanding the BOM. [↑](#footnote-ref-3)