UniSharping: конвертер C# в другие языки

# Введение

**UniSharping** – это конвертер кода **C#** на другие языки программирования: **Java**, **Python**, **PHP** и др. Основная «фишка» и отличительная особенность от других конвертеров состоит в том, что результирующий код получается работоспособным без какой-либо его ручной правки. Но приходится немного подправлять исходный код C#. То есть мы ориентируемся не на задачу разовой миграции, а на продолжение разработки на C# с получением в любой момент рабочего кода на нужном языке.

Эта амбициозная задача может быть в принципе решена, если исходный код будет удовлетворять некоторым ограничениям, касающимся конструкций языка, системных библиотек и технологий. Данное ограниченное подмножество условно здесь назвали **U# (Universal Sharp)**.

В целях кроссплатформенности Компания Microsoft уже сделала ограничение .NET Framework в плане библиотек и технологий: .NET Core. Это как бы первый шаг в нужном направлении, U# делает второй шаг к «кросспрограммируемости».

Ограничений U# в конструкциях языка оказалось немного – это атавизмы goto и case goto, а также yield, не моделируемый адекватно в автоматическом режиме. Не рекомендуется (хотя и можно) использовать struct, есть нюансы с наименованиями – всё это подробно описывается в отдельном документе. Парсер U# выдаёт ошибки и предупреждения, и для гарантии корректной генерации следует так подкорректировать исходный код C#, чтобы они в идеале совсем исчезли. Если всё-таки нужно сохранить исходный вариант, то можно использовать директивы препроцессора #if JAVA || PHP … #else … #endif. Данные ограничения действуют на уровне движка U# и не подлежат коррекции извне, как и список поддерживаемых языков.

А вот ограничения на уровне системных библиотек заданы не жёстко и конфигурируются извне через специальные текстовые файлы, определяющие, как переводить на соответствующий язык тот или иной класс и его члены. Если есть прямой аналог, то он и указывается, если ситуация сложнее, то пишется или фрагмент кода конечного языка, или вообще специальный (сервисный) класс, решающий нужную задачу. В совсем уж сложных случаях приходится «хардкодить» на уровне движка, но такие ситуации довольно редки (с десяток). Порядок настройки на системные классы и их члены описываются в отдельном документе.

Что касается технологий[[1]](#footnote-1), то здесь список ограничен на уровне движка консольным приложением и юнит-тестами (UnitTest)[[2]](#footnote-2). Ну и отдельные Lib-проекты, как частный случай, переводятся в соответствующие конструкции нужного языка.

Вообще для успешного перевода исходный проект C# (solution) должен иметь некоторую запускаемую часть, проверяющую работоспособность в рамках исходного C#. Хорошо, если это обширная система авто-тестов (стандартных UnitTest в разных реализациях или самописных), но по минимуму должно быть хотя бы консольное приложение, которое при запуске без какого-либо пользовательского вмешательства отрабатывает правильно. Необходимость этого очевидна – после генерации на конечный язык можно сразу проверить работоспособность. В идеале все тесты должны работать аналогично C#.

# Реализация конвертера

Конвертер UniSharping состоит из:

* UniSharping.exe - консольного приложения для пакетной обработки;
* UniSharping.Studio.exe – приложение с пользовательским GUI для настройки и конвертации;
* Текстовые файлы настроек для системных классов и их членов.

Конвертер разрабатывается на языке C# (естественно), но может свой код преобразовать в поддерживаемый язык (своё консольное приложение), что является одной из процедур проверки качества. Для запуска необходим .NET Framework 4 или выше, а на других платформах - Mono для консольного приложения.

Настроечные файлы должны находиться в той же директории, где расположены запускаемые модули, в следующих папках:

* C# - для текстовых файлов настроек, причём его система внутренних папок роли не играет, они просто для структурирования, а файлы извлекаются все с расширением txt;
* Java – сервисные классы Java, которые используются для моделирования некоторых системных классов C#, отсутствующих в Java. При генерации кода содержимое этой папки помещается в пакет unisharp в результирующую директорию. Особо следует выделить класс Utils – сборник статических методов «на разные случаи жизни»;
* Python, PHP … – аналогично для других языков;

Исполняемые модули, настроечные файлы и документация размещаются на [www.unisharp.ru](http://www.unisharp.ru) и <https://github.com/konstantin-smith/UniSharping>.

Конвертер отлаживался на реальном проекте лингвистической обработки текстов (см. [www.pullenti.ru/DownloadPage.aspx](http://www.pullenti.ru/DownloadPage.aspx) , там же есть пример результата его работы) и нескольких внутренних проектах. В статье <https://habr.com/post/354942/> есть дополнительная информация.

# Технология преобразования

Источниками исходных кодов могут выступать:

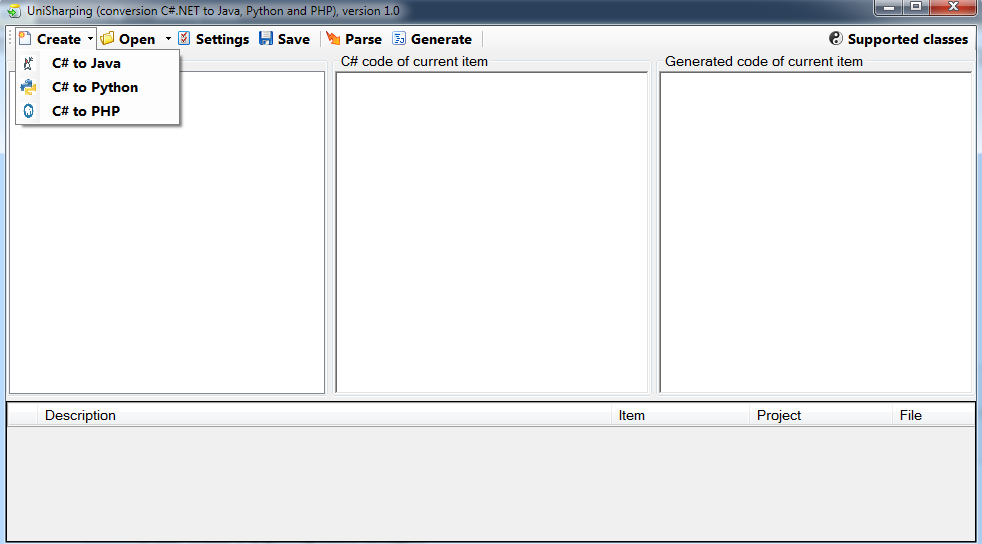
* Один или несколько cs-файлов;
* Один или несколько проектов csproj поддерживаемого типа, к которым в настоящий момент относятся (OutputType):
  + Library
  + Exe
* Солюшен (sln), из которого берутся все проекты поддерживаемых типов.

Понимается как «старый» формат проектов Framework, так и новый для .NET Core.

В проектах берутся не только файлы Build Action типа “Compile”, но и “Embedded Resource”, а также ресурсные файлы из Resources.resx, и они оформляются как аналоги ресурсов в конечной системе программирования.

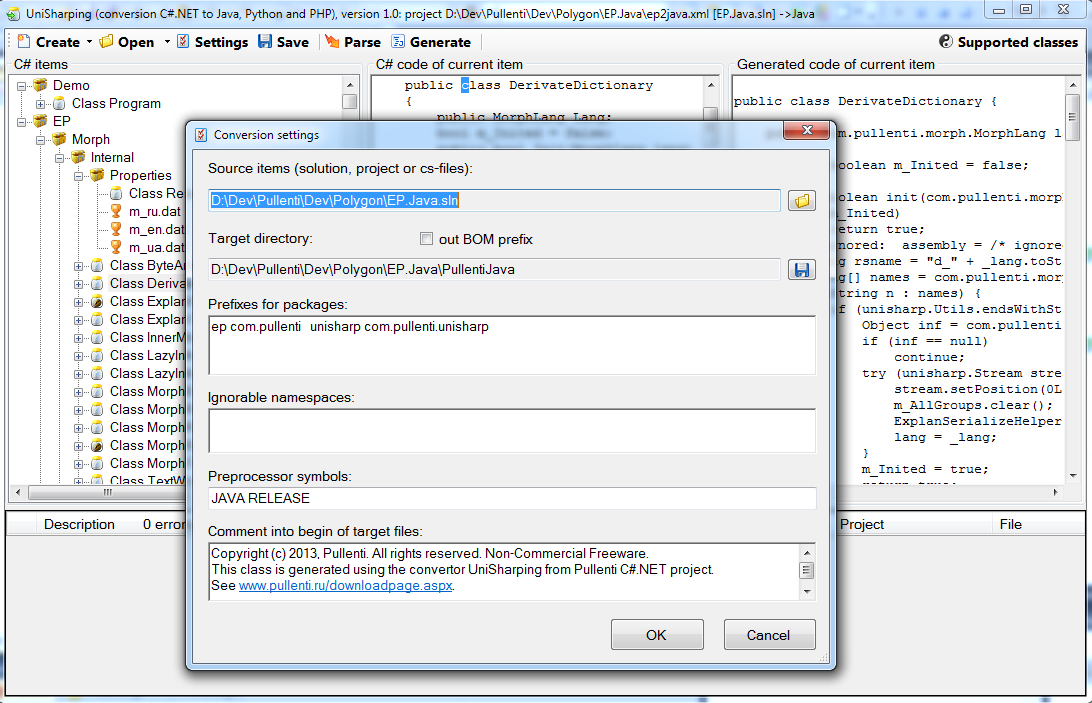
Необходимым (но не достаточным) условием успешной конвертации является отсутствие ошибок во всех конвертируемых исходниках, они должны компилироваться в Visual Studio.

Все ссылки на исходные модули, а также ряд настроек хранятся в так называемом файле конвертации (ФК), имеющем формат XML (описывается ниже). Для его создания и дальнейшей работы предлагается воспользоваться студией UniSharping.Studio.



В предложенном окне можно будет установить нужные настройки, и в дальнейшем открывать этот файл для конвертации. В ФК жестко задаётся результирующий язык, его в дальнейшем нельзя будет поменять кроме как сознанием нового ФК.

В принципе, окно настроек зависит от результирующего языка, но многие пункты общие. Вот как выглядит окно настроек ФК для Java:



* Target directory – директория, куда будут генерироваться файлы при нажатии кнопки “Generate”;
* Out BOM prefix – кодировка результирующих файлов UTF-8, выводить ли начальный префикс EF BB BF;
* Префиксы для некоторых пакетов можно переименовывать, и здесь задаётся соответствия «namespace .NET» - «пакет Java». Например, на рисунке “ep” будет заменен на “com.pullenti”, и его внутренние namespace аналогично: “ep.inner” -> “com.pullenti.inner” и т.д.
* Список игнорируемых namespace, из которых классы будут проигнорированы при анализе;
* Символы препроцессора, которые учитываются конвертером, что позволяет пользователю подправлять исходный код C# для задачи конвертации, отключая, например, некоторые фрагменты кода;
* Комментарий, вставляемый в начало каждого генерируемого файла;

Вот как это представлено в XML-файле:

<?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>Здесь комментарий в начала файлов…</comment>

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

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

<condition>JAVA</condition>

<condition>RELEASE</condition>

</project>

Как видно, все дорожки хранятся относительно текущего ФК что позволяет перемещать его вместе с исходными данными. Кстати, элементов <source> может быть несколько.

Далее наступает этап корректировки исходных кодов C#, чтобы парсинг (по кнопке Parse) проходил без ошибок и, по возможности, без предупреждений. Данному этапу посвящён отдельный документ, где подробно рассматриваются связанные с этим вопросы.

Генерация результирующего кода производится по кнопке Generate. Если есть ошибки при парсинге, то в результирующем коде также будут ошибки в соответствующих местах. Но если всё сделано правильно, то полученный код будет выполняться аналогичным образом, как и на исходном C#.

Подразумевается, что проект на конечном языке уже есть – в начале создан, хотя и пустой. Генератор только записывает файлы классов и ресурсные файлы, причём сам создаёт нужную структуру папок для моделирования пакетов для Java и Python. В большинстве IDE есть возможность динамической загрузки в проект файлов из папок и подпапок, например, в Eclipse по нажатию F5. Кстати, генератор также удаляет ненужные файлы и папки, которые были сделаны в ходе предыдущей генерации, а в новой версии из-за переименований и корректировки исходного C# пропали.

Ещё отметим, что файлы генерируются в кодировке UTF-8. Однако в той же Eclipse по умолчанию может стоять текущая кодировка Windows, и для проекта необходимо в настройках (Resources\Text file encoding\Other) поставить UTF-8[[3]](#footnote-3).

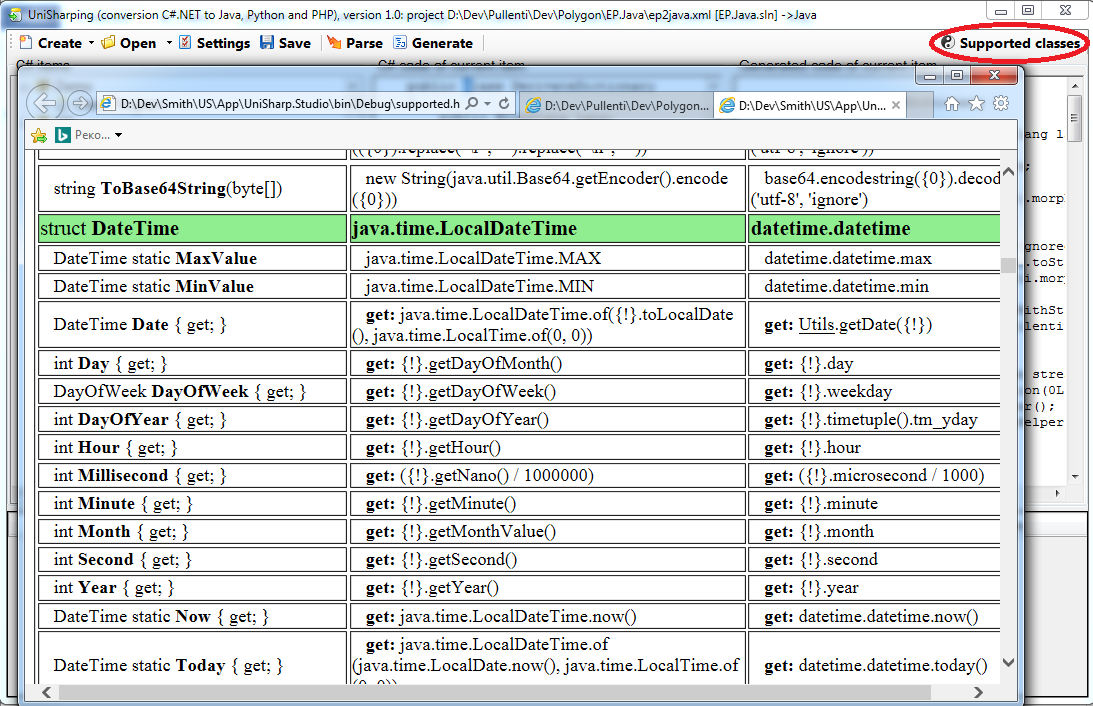
Таким образом, после создания ФК (файла конвертации), корректировки исходных C# и создания конечного проекта отлаженным процессом конвертации можно пользоваться постоянно, по мере развития продукта на исходном C# быстро генерируя версии на конечном языке. Можно использовать для этих целей не Studio, а консольное приложение, встраивая его, например, в процедуру формирования релиза.

Консольному приложению UniSharping на вход подаётся XML-файл конвертации. В случае ошибок или предупреждений соответствующие строки записываются в файл messages.csv, находящийся в той же директории, что и ФК, однако ошибки лучше анализировать именно в Studio, где есть система навигации и связи ошибок с местами в исходном коде.

# Расширения конвертера

Соответствие системных классов и методов C# и конечного языка задаётся в текстовых файлах поддиректории C#, откуда запускается конвертер. Добавление туда соответствующих файлов влечёт их поддержку после перезапуска.

Список текущих классов на конечных языках можно получить по кнопке “Supported classes” в Studio:



Сгенерированный HTML с текущим состоянием отобразится в браузере, в левой колонке элемент C#, далее аналог Java, Python и др. Вот как эта информация представляется в файле (сокращённый вариант):

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))

Настройка подробно описывается в отдельном документе.

ВНИМАНИЕ! Если Вы нашли ошибку или настроились на новые методы или классы, поделитесь, пожалуйста, этим с нами для вставки в очередной релиз.

1. Имеются в виду технологии конечных приложений типа Console, WinForm, WPF, ASP, а не промежуточные типа WCF, XML, Entity, которые считаем системными библиотеками. [↑](#footnote-ref-1)
2. Возможно в будущем здесь появится что-либо для Web. [↑](#footnote-ref-2)
3. Если в настройках ФК указать “out BOM prefix”, то этого в Eclipse делать не нужно, но при этом ругаются какие-то другие системы дистрибуции Java, не понимающие BOM. [↑](#footnote-ref-3)