UniSharping: Preparing the C# code

This document is under construction (translation), wait a few days …

Оглавление

[Введение 1](#_Toc524289362)

[Основные принципы конвертации 3](#_Toc524289363)

[Исправления исходного кода C# 4](#_Toc524289364)

[Недопустимые операторы 4](#_Toc524289365)

[Оператор yield 4](#_Toc524289366)

[Одинаковые сигнатуры методов 4](#_Toc524289367)

[Дублирования методов 5](#_Toc524289368)

[Имена классов 6](#_Toc524289369)

[Пересечение блоков оператора switch (Python) 6](#_Toc524289370)

[Exceptions в переопределяемых методах (Java) 6](#_Toc524289371)

[Пост-инкремент для элементов списков (Java) 7](#_Toc524289372)

[Присваивания внутри выражений (Python) 7](#_Toc524289373)

[Оператор ?? 7](#_Toc524289374)

[Структуры 8](#_Toc524289375)

[Кодировки текста 8](#_Toc524289376)

[Операция lock (Python) 8](#_Toc524289377)

[Flate/Deflate 8](#_Toc524289378)

[Круговая зависимость импортирования (Python) 9](#_Toc524289379)

[Порядок ключей Dictionary 9](#_Toc524289380)

[Сортировка списков (Python) 9](#_Toc524289381)

[Format 10](#_Toc524289382)

# Введение

Для понимания дальнейшего необходимо ознакомиться с содержимым UniSharping.Overview.docx. В этом документе описываются ограничения U#, позволяющие достигнуть «кросспрограммируемости», причины возникновения этих ограничений и способы их преодоления. Конечная цель – чтобы реальный проект C# автоматически переводился в исходные коды другого языка, которые работоспособны без какой-либо правки.

Здесь постараемся описать процесс из личного опыта, который проходит так:

1. В UniSharping.Studio запускаем парсинг;
2. Анализируем ошибки;
3. Исправляем код в Visual Studio (или дорабатываем настройки на ещё неподдержанные системные методы);
4. Возвращаемся к п.1, пока остаются ошибки.

Однако отсутствие ошибок ещё не гарантирует конечную работоспособность. Приходится делать доработки движка UniSharping. Тем не менее процесс оказывается сходящимся, и бонусом служит поддержка в рамках исходного продукта C# ещё одного языка.

Сразу перечислим, что в принципе не поддерживается и вряд ли получится в будущем (это диктуется особенностями конечных языков):

* Графические возможности и всё, связанное с GUI (System.Drawing, Windows.Forms и т.п.);
* Привязанные к Windows возможности (из namespace Microsoft);
* Операторы goto и yield;
* Interop-взаимодействие с DLL (DllImport);
* Unmanaged code (unsafe, pointers …);
* Атрибуты […] у членов классов (они в U# просто игнорируются);

Кстати, если проекты C# ещё не на .NET Core, то перевод их на Кору автоматически избавит от некоторых ограничений U#.

Понятно, что операторы goto и yield легко заменить, переписав код. В крайнем случае можно воспользоваться такой полезной возможностью, как директивы препроцессора, понимаемые UniSharping. В файле конвертации задаются “Conditional compilation symbols”, которые по умолчанию RELEASE, UNISHARPING и JAVA\PYTHON\PHP в зависимости от конечного языка. Так что я поступаю так, когда нужно сохранить исходный вариант для C#:

#if JAVA || PYTHON

 Вариант кода на C# без ограничений для перевода

#else

 Исходный вариант с ограничениями

#endif

Этот способ позволяет обойти некоторые принципиальные ограничения, но так победить GUI-шный проект не получится. А как получится?

А никак! Всё-таки конвертация в первую очередь рассчитана на невизуальные проекты типа SDK или на пакетные обработчики. Конечно, можно для использования функционала SDK на .NET в Java использовать Web-сервис и взаимодействие по какому-нибудь REST – так мы и поступали до появления UniSharping. Но использование нативного SDK удобнее…

Другой способ корректировки кода – это специализированные комментарии перед операторами, которые дают подсказку конвертеру в сложных ситуациях: //JAVA: … и //PYTHON: … Используются они крайне редко, но по существу, и эти ситуации описываются ниже.

Ещё есть возможность в файле конфигурации перечислять игнорируемые пространства имён. Например, у проекта могут оказаться такие части, которые не нужны на результирующем языке или просто не подлежат конвертации по ряду причин. В принципе, эту же задачу можно решить через директивы препроцессора, но для большого числа файлов это может оказаться весьма трудоёмко.

# Основные принципы конвертации

Перечислим здесь основные принципы при получении результата:

* Самодостаточность результирующего кода: в результате получается исходный код, не требующий никаких дополнительных подключений модулей и библиотек, не входящих в стандартные пакеты и библиотеки (JDK, например)[[1]](#footnote-1);
* Следование рекомендациям по оформлению кода (отступы, наименования и т.п.): исходные имена классов и их членов корректируются соответственно (см. далее);
* Эквивалентность выполнения и работы с данными: если данные получены на одном языке, то они могут быть использованы и на другом (например, сериализовали некоторую структуру на модуле C# и десериализовали на Java);
* Корректность результирующего кода (как необходимое условие принципа эквивалентности);
* Комментарии /// переводятся в соответствующие /\*\* (Java, PHP) и docstring ‘’’ (Python) для автодокументирования;

Наши результирующие языки поддерживают ООП, и исходные класс переводится в соответствующий класс конечного языка.

Со сборками и пространствами имён сложнее ввиду отсутствия таковых понятий в Java\Python\PHP. Мы переводим иерархию пространства имён в иерархию пакетов (папок), а про сборки проектов «забываем» (как бы виртуально объединяем все проекты C# в один большой проект). Это решает задачу, но возникает проблема с ограничениями видимости private и internal у классов, которые действуют на уровне сборок в C#, а в конечных языках на уровне пакетов. В связи с этим у всех классов область видимости считается public.

Члены классов также переводятся в соответствующие члены, при этом если нет прямого аналога, то происходит моделирование. Например, property X { get; set; } отсутствуют в Java, но моделируются двумя функциями getX и setX, названия которых получаются из имени свойства плюс префиксы get и set.

Производится проверка всех имён на предмет конфликта с ключевыми словами и системными функциями конечного языка и автоматическая их корректировка.

Класс может дополняться новыми членами, отсутствующими в исходном классе. Например, для моделирования оператора new X() { Val = v, … } приходится вводить в класс новый метод, решающий задачу инициализации в своём теле.

Аргументы ref\out отсутствуют в Java\Python (в PHP есть), но легко моделируются обёрточными классами типа class ArgWrap { object Value; }.

Делегаты становятся интерфейсами с тем же именем и одной функцией с именем call.

События event моделируются списками экземпляров соответствующих реализаций делегатных интерфейсов. Это всё конвертер берёт на себя, добавляя в исходный класс внутренние подклассы и статические члены (примеры см. в реальном конвертере).

Ресурсные файлы имеют аналоги на результирующих языках, и конвертер понимает как встроенные ресурсы (Build Action = “Embedded Resource”), так и устаревший способ через resx.

Python не поддерживает операторы типа ++, --, += и пр., эта ситуация моделируется инициализацией перед основной операцией, но здесь возможны варианты, на которые парсер будет «ругаться».

# Исправления исходного кода C#

Итак, конвертер большую часть работы берёт на себя. Однако возможны ситуации, с которыми он справиться не может. Если это относится к той или иной системной функции, которая ещё не поддержана, то это может быть решено либо настройкой на эту функцию[[2]](#footnote-2), либо использованием поддержанной функции. Но если это касается языковых конструкций, то здесь в ряде случаев придётся подправить исходный код. Описанию таких ситуаций и отводится этот раздел. Причём ситуация может зависеть от конечного языка – для одного она может быть критичной, для другого нет.

Когда парсер сталкивается с такой ситуацией, то он генерирует ошибку или предупреждение с привязкой к месту в коде, в UniSharping.Studio реализована навигация по таким местам. Разработчику предлагается найти это место в MS Visual Studio и подправить код её средствами (например, переименование через рефакторинг), убедиться в работоспособности проекта и снова сделать парсинг в UniSharping.

## Недопустимые операторы

Редко, но встречаются goto и goto case, от них нужно просто избавиться, переписав код. Когда я переводил чужой старый код, то встретил с десяток таких операторов, с которыми расправился самым безжалостным образом.

## Оператор yield

Его поддержка обеспечена на уровне ядра .NET, но в других языках аналог отсутствует. Можно было бы попытаться автоматизировать процесс через создание локальных списков, заполнения их и возврате, но тут можем попасть на ситуацию бесконечного списка. Поэтому предлагается такие случаи исправлять вручную. Если из соображений эффективности нужно оставить исходный вариант в C#, то можно использовать директивы препроцессора, понимаемые UniSharping:

#if UNISHARPING // или JAVA или PYTHON …

Вариант для UniSharping с List<…> res … return res

#else

 Исходный вариант с yield return …

#endif

## Одинаковые сигнатуры методов

При формировании сигнатуры методов в C# учитываются генерик-параметры, а в Java нет, поэтому разные функции в C#:

void Func(List<int> …)

void Func(List<string> …)

но в Java они по сигнатуре одинаковы:

void func(ArrayList<Integer> …)

void func(ArrayList<String> …)

Такие ситуации оформляются ошибками, и предлагается переименовать одну из функций. Конечно, можно было бы автоматически корректировать имена, добавляя, скажем, нолик, но всё-таки решено оставить это на ручную корректировку, тем более, что в Visual Studio есть для этого удобная возможность.

С Python ситуация «гораздо хуже». Здесь у методов отсутствуют типы параметров, так что сигнатура = имени, и в принципе возможна только одна функция с одним именем. Если решено поддерживать и Python, то придётся обеспечить полную уникальность на уровне класса. В том числе и для конструкторов, который в классе может быть только один.

Ещё возможна ситуация, когда в классе есть свойство property Name { get; set; } и есть отдельно метод GetName() или SetName(value). Поскольку для Java свойство оформляется через методы с префиксами get и set к имени свойства и для методов первый символ переводится в нижний регистр, то получится одинаковые 2 метода getName() или setName(…). Здесь также потребуется переименовать или имя свойства, или метода.

## Дублирования методов

При реализации некоторых системных интерфейсов могут возникнуть дублирования. Например, для IEnumerable<…> Например,

public class ChildClass1 : Class1, IEnumerable<Class1>

{

 public List<Class1> Classes1 = new List<Class1>();

 public IEnumerator<Class1> GetEnumerator()

 {

 return Classes1.GetEnumerator();

 }

#if !UNISHARPING

 IEnumerator IEnumerable.GetEnumerator()

 {

 return Classes1.GetEnumerator();

 }

#endif

То есть в C# должен быть реализован как генерик-вариант, так и старый вариант, которые по сигнатуре результирующих методов получаются одинаковыми. Решением является использование директив препроцессора, как в предыдущем примере.

Также часто встречается ситуация реализации IDisposable метода Dispose, и одновременно в классе присутствует метод void Close(), обычно просто вызывающий этот Dispose (например, разные Stream). Поскольку аналогом IDisposable является AutoСlosable с методом close(), а исходный метод Close() также получает эту сигнатуру после перевода первой буквы в нижний регистр, то нужно избавиться от одной из функций. Если Dispose() просто вызывает Close(), то нужно обрамить этот Dispose директивами #if !UNISHARPING - #endif. Кстати, в .NET Core заметили этот нюанс и вообще исключили Close() из всех Stream.

## Имена классов

Имя класса может конфликтовать с существующим системным классом конечного языка. Например, Integer и Character в Java. Такие ситуации оформляются ошибками и класс должен быть переименован вручную.

## Пересечение блоков оператора switch (Python)

Оператор switch почему-то отсутствует в Python, поэтому автоматически моделируется через if then else. И здесь есть один случай, который трудно промоделировать: отсутствие break в конце блока и переход на другой блок (но если в конце есть return или throw, то всё нормально).

switch(…)

case v1: БЛОК1 без break в конце

case v2: БЛОК2 …

то есть когда v1, то выполняется БЛОК1 + БЛОК2. Можно было бы при генерации эти блок БЛОК2 дублировать в case v1, но решено оставить эти случаи на ручную модификацию.

Кстати, случай пустого блока БЛОК1 вполне допустим, когда case v1: case v2: БЛОК2.

## Exceptions в переопределяемых методах (Java)

В Java необходимо при определении методов указывать thows-список потенциальных исключений, причём если внутри тела метода есть метод с некоторым исключением, то оно обязательно должно попасть в список определения.

Выяснилось, что для сложного проекта с циклическими зависимостями составление такого списка есть нетривиальная задача. Более того, иногда даже приходится «разрывать циклы», принудительно обрамляя тела некоторых методов через try \ catch, чтобы очистить список исключений. Большая часть этой работы проделывается UniSharping, но есть нюансы, подлежащие только ручной корректировке.

Есть правило Java: переопределяющий метод (override) не может расширять throws–список переопределяемого им метода (virtual или в interface). Если переопределяемый метод пользовательский, то тут система просто расширяет его trows-список, но если это системный метод, то тут нужно ручное вмешательство.

Например, пусть некоторый класс реализует IDisposable метод Dispose(), который генерирует IOException (например, закрывает файловый поток). Аналог в Java – это интерфейс Autoclosable с методом void close() throw Exception. Тут мы не можем расширить список, и нужно подправить переопределяющий метод, чтобы он или мог генерировать только Exception, или вообще ничего не генерировать [*вроде сейчас это делается автоматом*].

А случай исключений в конструкторе не удаётся так просто решить, и всё потому, что в Java нельзя обрамлять вызов конструктора базового класса через try\catch.

Например, MyClass() : base(new FileStream(…)) { … } аргумент вызова конструктора базового класса сам генерирует исключение, которое не относится к исключениям базового класса. В Java так делать нельзя:

MyClass() {

try { super(new FileStream …); }

catch() {}

Нужно так переписать код, чтобы вызовы базового конструктора не содержали в передаваемых параметрах потенциальных исключений.

## Пост-инкремент для элементов списков (Java)

Аналогом List<…> в Java является ArrayList, но для получения значения используется метод get(int), а для записи – set(int, value). Когда в выражениях встречаются операторы типа … (li[i] += 10)…, то это моделируется как Utils.putArrayValue(li, i, li.get(i) + 10)[[3]](#footnote-3). Сложность возникает при использовании пост-инкремента или декремента внутри выражений или операторов.

Например, if(i > 0 && li[i]++ > 0) …

Если бы это был пре-инкремент, то if(i > 0 && Utils.putArrayValue(li, i, li.get(i) + 1)) … был бы корректен, но вот куда вставить изменение после выполнения оператора – это пока для UniSharping сложновато. Использовать системный set тоже не получится, так как в then-блоке должно быть ещё неизменённое значение. Поэтому такие ситуации оформляются ошибками и подлежат ручному исправлению.

## Присваивания внутри выражений (Python)

В Питоне нельзя использовать операторы типа =, +=, ++, /= и пр. внутри выражений и аргументами методов – они разрешены только как отдельно стоящие операции. Поэтому приходится моделировать путём размещения соответствующих операций перед основной операцией, содержащей эти операторы. И вот здесь возникают нюансы.

Например, if( (x += 10) > 100 && (y += 20) > 200) …

Это вовсе не эквивалентно: x += 10; y += 20; if(x > 100 && y > 200) …, поскольку при выполнении если после изменения x окажется меньше 100, то вторая операция выполняться не будет.

Поэтому UniSharping в данной ситуации сформирует ошибку для второй операции, а первая будет промоделирована указанным образом.

Пост-инкремент ++ и декремент - - по этой же причине неоднозначности запрещены внутри выражений и аргументов, но можно как отдельные операции.

## Оператор ??

Оператор x ?? y отсутствует в конечных языках и содержит подвох при своём моделировании. Если x != null, то до “y” вообще вычисления не дойдут и он может содержать в себе любые потенциальные ошибки, например, NullReference. Здесь UniSharping пытается эффективно промоделировать этот оператор в зависимости от x и y:

x != null ? x : y - если x является простым выражением

Utils.ifNotNull(x, y) – если x является сложным или y простым

Если в последнем случае обнаруживается, что “y” есть хотя бы один оператор типа a.b, то есть “a” может быть null и потенциально здесь NullException, то такая ситуация оформляется предупреждением и лучше всё-таки подправить код, даже если там по логике программы не может быть null.

## Структуры

Структуры (struct) также отсутствуют в конечных языках. UniSharping моделирует их классами, причём старается их явно инициализировать (через new) там, где они не инициализированы (так как создаются автоматически). Но возможны ситуации в алгоритме, когда будут скопированы ссылки, а не значения. На своём опыте я пришёл к тому, что структуры вообще не нужно использовать. В моём большом проекте было 4 штуки, и я думал, что они увеличивают производительность. Но при переводе возникли неточности функционирования именно из-за структур. Когда же я их переделал в классы, то неточности исчезли, а производительность осталось той же. Но если они используются «аккуратно» в местах своего копирования, то, может, их наличие и не повлияет на результат.

## Кодировки текста

Когда я поддерживал свои проекты и на .NET Core, то был удивлён, что всегда работающий Encoding.GetEncoding(1251) вернул пустой результат. Мораль понятна – всегда использовать UTF8. Для генерируемых мной данных я так и поступаю, но при анализе, например, формата rft там может встретиться и не такое. Пришлось полностью отказаться от GetEncoding и реализовать нужные кодировки самостоятельно.

UniSharping поддерживает разные кодировки а Java (то, что может java.nio.charset.Charset метод forName), но для Python сейчас можно только UTF-8. Поэтому если обнаружите предупреждение с недопустимой кодировкой, то лучше всё-таки по возможности исправить свой код.

## Операция lock (Python)

В C# в качестве lock(…) можно использовать объект любого типа. В Python для этого предназначен специальный тип threading.Lock(), поэтому в C# этот объект должен быть типа object и нигде больше не использоваться кроме как в lock-операциях. Например, если вы локируете экземпляр некоторого класса, то создайте в нём поле object m\_ForLock = new object() и используйте его.

## Flate/Deflate

Если используются потоки DeflateStream, то Java-аналоги java.util.zip.Infrater/Deflater работают по- другому. Если сжать алгоритмом C#, то не получится его разархивировать на Java и наоборот. Лучше использовать GZipStream, аналоги который идентично работают как в Java, так и на Python.

## Круговая зависимость импортирования (Python)

В Python при задании import должна соблюдаться иерархия загрузки и отсутствие циклических зависимостей, что трудно избежать в реальных проектах C#. UniSharping пытается эту зависимость «упорядочить», вставляя в начала файла те import, которые удовлетворяют иерархичности, а остальные import добавляют в те методы, где используются соответствующие классы. Таким образом большинство сложных ситуаций удаётся автоматически обработать, но редко попадаются случаи, когда автоматика не справляется.

В этом случае перед оператором, в котором используется некоторый класс и импорт которого не должен попасть в начало файла, а попасть именно в текущий метод, нужно вставить комментарий: //PYTHON local import ИМЯ\_КЛАССА

В моём проекте такая ситуация встретилась, и вот её решение:

 public void LoadNode(MorphTreeNode tn)

 {

 lock (Engine.m\_Lock)

 {

 Data.Seek(Begin);

 //PYTHON local import MorphSerializeHelper

 MorphSerializeHelper.DeserializeMorphTreeNodeLazy(Data, tn, Engine);

 }

 }

Вот результирующий код:

def **load\_node**(*self*, tn : *'MorphTreeNode'*) -> None:

 with *self*.engine.\_m\_lock:

 from pullenti.morph.internal.MorphSerializeHelper import MorphSerializeHelper

 *self*.data.seek(*self*.begin)

 MorphSerializeHelper.\_deserialize\_morph\_tree\_node\_lazy(*self*.data, tn, *self*.engine)

Надеюсь, вам это не потребуется...

## Порядок ключей Dictionary

Теоретически при использовании Dictionary нельзя полагаться на порядок элементов (ключей и пар «ключ-значение»), однако на практике в C# этот порядок такой, в каком элементы добавляются в словарь. Но в конечных языках это не так – порядок не соблюдается. Если алгоритм учитывает порядок поступления элементов в Dictionary, то следует избавиться от этой зависимости, переписав алгоритм, чтобы обеспечить тождественность результата в конечных языках. Отметим, что эту ситуацию UniSharping никак не обнаруживает, это выясняется уже на этапе отладки на конечном языке.

## Сортировка списков (Python)

В Python 3 для сортировок списков (list) экземпляров классов оставили из версии 2 только один способ – сортировка по ключу. Когда список из элементов простой, то сортировка идёт функцией sort(), для сложного случая универсального решения пока не найдено, и в каждом конкретном таком случае предлагается добавлять комментарий с оператором sort, который будет использоваться при генерации.

Например, class Class1 { public int IntVal; } реализует IComparable<Class1> для сортировки по IntVal, тогда в месте непосредственной сортировки List<Class1> li нужно добавить комментарий:

//PYTHON: sort(key=attrgetter('int\_val'))

li.Sort();

где указать оператор, причём он должен быть уже в том виде, чтобы сразу быть вставленным в результирующий код, чтобы в результате получилось:

li.sort(key=operator.attrgetter(*'int\_val'*))

Данный способ нужно использовать как для Sort() без параметров в предположении, что элементы списка реализуют IComparable, так и в случае подачи параметром Sort(cmp) объекта, реализующего IComparer.

Согласен, что решение корявое, и в будущем должно появиться что-либо более удачное.

## Format

Форматированный вывод строки по шаблону используется в нескольких классах: string, Console, StringBuilder. Сейчас анализ шаблона происходит в «хардкоде» для генерации адекватной последовательности операторов и операций, поэтом основное требование – первый параметр форматирования (шаблон) должен быть константной строкой (лексемой).

В будущем планируется написать разбор шаблона на конечных языках, чтобы снять это ограничение.

1. За исключением UnitTest для Java и PHP – нужно подключить к проекту Junit5 или … [↑](#footnote-ref-1)
2. Описывается в отдельном документе. ВНИМАНИЕ! При такой настройке просьба ей обязательно поделиться. [↑](#footnote-ref-2)
3. К сожалению, set возвращает старое значение, поэтому не может здесь быть использован, приходится через нашу сервисную функцию. [↑](#footnote-ref-3)