Статья предназначена для исследователей, программистов и просто энтузиастов изучения разнообразных языков и стилей программирования. В ней рассмотрено слияние функционального, объектно-ориентированного, контекстно-ориентированного стилей программирования на примере языка C#. Результатом изысканий оказалась библиотека общего назначения, расширяющая возможности языка при минимальном объеме кода.

Долгие годы программный параллелизм стучался в наш мир из недр аппаратного обеспечения. Пример тому: архитектура x86 - в которой ассемблерные команды по большей части исполнялись последовательно; и пришедшая и укрепившаяся на рубеже минувшего тысячелетия, IA-64 – в которой параллелизм извлекается за счет упаковки компилятором нескольких команд, которые могут выполняться одновременно, в более длинную. Об этом, подробнее в интернет статье “Архитектура IA64”[#001]…

Для компании Intel, переход от x86 к IA-64 был кардинальным шагом. Другие производители не спешили отказываться от x86. Подобный “Флаттер” оставил продолжительный, и впечатляющий своими красками мазок кисти в истории программирования.

Обратим внимание на 2 языка программирования. Это С++ и Haskell. Они появились, в том числе и как следствие борьбы за стандарты 1970 – 2010 годы.

Haskell – стандарт-потомок языков программирования научно исследовательской среды, потому он и получился число функциональным языком программирования. Этот язык жил по принципу приближенному к тезису “Все есть функция”, это относилось даже к значениям переменных. Потому передаваемый параметр в функцию мог быть результатом вычисления другой функции, вычисления которой отложены до востребования (Ленивые вычисления). В подобных вычислениях порядок выполнения функций может быть предметом оптимизации компилятора…

За счет того что каждая задача решалась написанием функции и её подфункций, тестирование программы сводилось к тестированию функции с тестовыми значениями параметров по умолчанию. Потому, в функциональных языках программирования много поточность и параллелизм проще реализуются и отлаживаются. Все это было, так сказать, “Из коробки?”. В подтверждение этого факта упомяну следующее: функциональный язык Lisp (неофициальный предок Haskell), Lisp-машины, Lisp-ассемблер (он тоже функциональный), “…с аппаратной поддержкой лямбда исчисления — около 7000 штук в мире по состоянию на 1988 год” – цитата из статья “ Краткий очерк истории Lisp машин” [#002].

Использование 64 битной архитектуры: с 1970 год в супер компьютерах; c 1990 год в серверах; в 2003 год в персональных компьютерах; в 2010 в мобильных устройствах [#003]. Бурное использование параллелизма в высокоуровневых языках программирования, подогреваемое надвигающейся 64 битной архитектурой - запаздывало. По состояние на 2005-2010 год подавляющая часть программного обеспечения не использовала параллелизм и была однопоточной, к тому же 32 битной. Долгое время 32 битным был и пакет MOffice…

Промышленное, императивное/объектно-ориентированное программирование отчаянно нуждалось в параллелизме. Технологии, механизмы и способы реализации которого представляли широкий ассортимент возможностей юному программисту “Выстрелить себе с ногу” (устойчивый оборот в программистской среде на просторах интернета). И на этом фоне … функциональное программирование злободневно напоминает о том, что все новое, это хорошо забытое старое. Там параллелизм – “шел из коробки”, и при том без особых усилий и с незапамятных времен появления первых функциональных языков программирования…

2005-2010 – можно охарактеризовать жесточайшими кровопролитиями в интернет спорах на тему: Кто лучше? Функциональный или императивно/объектное ориентированный стиль программирования?  
Предлагаю разделить функциональный стиль программирования и функционально процедурный на два разных стиля, и последний пока не трогать…

Что-ж, это баталии давно минувших дней. С тех пор Haskell стал чем-то на подобии музейного экспоната функционального программирования, учебник по которому вручался отличникам чисто в образовательных целях. Язык красив, компактен и логичен, но тяжел в изучении. Освоить учебный материал по Haskell большинству студентов удается только можно, только с “третей попытки” полного прочтения книги, благо русскоязычная литература в достатке. Возможно, именно этот факт привел к тому что сейчас, по данным открытых источников, на Haskell пишут менее 0,01% населения программистов.  
Промышленная разработка на функциональных языках программирования встала на прочь, хотя именно на них в прошлом столетии были написаны самые крупные и крайне надежные системы…

По сравнению с функциональными языками программированиями освоение императивной/объектно-ориентированной парадигмы дается куда легче. Подозреваю, что в последнее время, на практике, ставится задача написать программный код не в каком-то определенном стиле, а так чтобы он просто работал при минимальных затратах на разработку.

Так что в споре “Кто лучше?” – не победил ни кто!

Подозреваю, как следствие этих споров Microsoft в 2010 выпустил F# .NET [#005].

Следующее поколение программистов старалось писать на стыке стилей программирования, в меру знаний своих.

Существующие стили программирования:

Императивное программирование; Структурное программирование; Процедурное программирование; Модульное программирование. Они были описаны в статье «Забытые» парадигмы программирования [#004]. Автор признался в том что не указал объектно-ориентированное программирование и функциональное программирование, по причине того что боялся гневных споров на тему “Кто лучше?” в комментариях к статье.

Кроме описанных выше, существуют ещё и: контекстно ориентированное программирование (наследник ООП, наделят объекты новым функционалом в зависимости от вызванных пространств имен); рефлексия и динамическое программирование (подтягивание скомпилированного кода в проект по сети в процессе выполнения программы), генетические типы (<T>), методы и динамическая моно-диспетчерная компиляция, динамическая мульти-диспетчерная компиляция. Список с каждым днем продолжает пополняться…

В 2017-2020 годах, сложилась такая ситуация, что учащиеся периодически используют приемы этих стилей особо не различая парадигмы из которых они пришли. Тем не менее, большинство из них пишет в императивно/объектно-ориентированной парадигме, слепая преданность к которой лишает красок жизнь программиста.

Как следствие на С# .NET проектов и рабочих мест во много раз больше чем на F# .NET.

Для того чтобы использовать функциональные принципы программирования, не обязательно писать на функциональном языке, то же справедливо и для объектное ориентированного языка... Как ярчайшее тому доказательство можно привести факт существования промежуточного языка IL платформы .NET . И С# и F# работают через него. В конце концов в дизассемблере при пошаговом исполнении один и тот же Microsoft Assembler (MASM). Вывод: нет той программы, которую можно реализовать только на С# или только на F#. Объектно-ориентированный или функциональный стиль программирования – это только вопрос оформления на уровне высокоуровневого программного кода, вопрос удобства программиста. Исторически сложилось, что функциональный программный код занимает меньше места чем объектно-ориентированный, то есть его можно быстрее набрать с клавиатуры, что мне в нем и понравилось.

Очень интересной попыткой использовать функциональную парадигму в С# является LinqTo: Objects, XML, SQL, DataSet, Entities, Parallel. Это язык интегрированных запросов.

LinqToObjects – диалект C# для обработки массивов, списков List и IEnumerable.

В нем прослеживаются до боль знакомые паттерны проектирования функционального языка Haskell: принцип возвращения с продолжением и отложенные вычисления… Паттернов функциональных языков программирования не так много не более 10, а в ООП их количество легко переваливает за пол сотни.  
У LinqTo есть как минимум одна беда: авторы пытались создать нечто красивое в функциональном стиле, но при этом страшно тяготели к объектно-ориентированной парадигме. Например, в чем это выражалось... Принцип возвращения с продолжением подразумевает цепочку вызовов функций, каждая из которых возвращает объект от которого она была вызвана. Такая цепочка обрывается только по необходимости программистом или при вызове функции, результат которой – возвращение вычисляемого значения. Есть ещё одно понимание возвращения с продолжением: это когда в функцию передается лямбда выражение с ветвлением (другая функция), и две другие функции из которых возвращается одна, в зависимости от результата ветвления. Пока что мы рассматриваем возвращение с продолжением в первом озвученном варианте…

Программный код на чистом C#, LinqToObjects, грубо обрывается функцией   
“void <List>.ForEach(a=>{});“, которая ничего не возвращает! Это не по функциональному!  
Согласно контексту со списком после “ForEach” можно сделать что-нибудь ещё.

Синтаксические подводные камни жестко ограничивают действительность и вынуждают создать локальную переменную, присвоить в неё значения и работать с ней дальше только в императивном ООП стиле. Изредка и не на долго перескакивая на “Макаронный” код.

В ООП это не понятый анти паттерн, который избегают как огня. В функциональном программировании это вылизанный до блеска принцип возвращения с продолжением, который то и дело проскакивает в DLL-ных программных кодах на всех программистских языках мира, Delphi, OpenGL/C++, D3/JavaScript, С#/ LinqTo, Go, Julia, Pyton...

Программный код на чистом C#, LinqToObjects.

|  |
| --- |
| (new string[] { "40", "2012", "176", "5" })  .Select(s => Int32.Parse(s))  //Приведение всего массива к типу  .Cast<System.Object>().Cast<System.Int32>()  //Или тоже самое  .OfType<System.Object>().OfType<System.Int32>()  //Отличае только в том что OfType работает с унаследованием от коллекции а Каст нет  //Как пишут OfType обрезает коллекцию в тех случаях когда не может привести к типу  //А Cast генерит исключение  .Select(s => s + 1).ToArray()  .ToList().ForEach(a => Console.Write(a + " "))  ; |

Мною с 2014 по 2020 изучался/разрабатывался способ писать программные коды в С# .NET одновременно и в функциональном и ООП стиле программирования, полное их сращение. Была разработана библиотека “ArgeCode“, при подключении которой к проекту С# добавляется возможность писать программный код на стыке двух стилей, дозируя их в любых количествах. Название библиотеки выбрано не случайно. Программный код не претендует на золото, но и бронза его не назовешь. Argentum – элемент из таблицы Менделеева “**Ag**”(далее так и будем сокращать)- серебро. Обладает разумными дезинфицирующими свойствами. Но с непривычки даже им можно отравиться. Тем самым выношу библиотеку на суд времени и хочу обратить на неё ваше внимание.

Актуальная на момент статьи версия библиотеки “ArgeCode 0.0.2“. Версия стабильна, отлажена, уже используется как учебных целях, так и в разработке других проектов.

ArgeCode выложена в свободный доступ на GitHub и экосистему NuGet. То есть библиотеку уже можно скачать через менеджер NuGet пакетов Visual Studio.  
ArgeCode имеет жесткую внутреннюю организацию и вылизана на сколько это возможно. ArgeCode не будет меняться в ближайшее время. На сегодняшний день она в том состоянии, когда изменения программных кодов добавляются реже, чем раз в год по просьбе трудящихся и только после долгого мозгового штурма. Также, библиотека содержит примеры программных кодов. Папки пронумерованы в порядке рекомендуемого изучения исходных кодов. Для папок справедливо, что папка не может использовать программные коды из папки с более высоким номером. Это уберегает от излишнего количества перекрестных взаимосвязей и превращает обслуживание и изменение кода в перестройку верхних слоев кода.

В отдельной папке количество файлов и папок не превышает числа Миллера  
(7\_+-2\_или+-4\_) [#007], что улучшает психологическую воспринимаемость программного кода.

Изображение выглядит как текст

Автоматически созданное описание

Рис. 1. Структура “ ArgeCode “.

Начнем с 01\_System.

Изображение выглядит как текст, электроника

Автоматически созданное описание

Рис. 2. Структура папки “01\_System“.

Файл, с которого все началось… Set\_T.cs.  
В нем реализовано генетическое контекстно ориентированное расширение, которое добавляет к любому экземпляру объекта указанную функцию…

Не секрет, что в программировании, а в частности в C# есть проторенные дороги, а есть тропы, по которым никто не ходил, вообще. В С# достаточно часто можно найти некоторые вполне пригодные для одиночного использования конкретным образом приемы и механизмы, но при попытке их совместного или более обобщенного использования возникают неимоверно большие трудность, вплоть до того, что "В теории должно работать, а на практике не работает ... одни сплошные огорчения!". Поиск таких работающих сочетаний - весьма трудозатратное для этого языка дело, так как удача приходит примерно в каждом десятом случае. Остальные попытки упираются в жестокий синтаксический коммерческий тупик. На сей раз повезло, после многочисленных попыток инъекций зависимостей через менеджеры внедрения зависимостей, интерфейсы, конструкторы, и get/set методы выйти на это два приема, которые по отдельности встречаются в документации C# Microsoft: обобщения, Generics, <T>, генетические методы и типы [#008]; Методы расширения [#009] – но в месте нигде не фигурируют. Как будто о них никто не знает, или их старательно скрывают…? Судите сами:

|  |
| --- |
| //File Set\_T.cs  namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  ////////////////////////////////////////////////////  ////////////////////////////////////////////////////  public static T Set<T>(this T \_this, System.Action<T> x) { x(\_this); return \_this; }  ////////////////////////////////////////////////////  ////////////////////////////////////////////////////  private class LocalClassForTest  {  public System.Int32 p\_Int32 = 6;  public System.String p\_String = "Привет мир";  public override string ToString()=>"<"+this.p\_Int32+"><"+ p\_String + ">";  }  public static void Test()  {  (new LocalClassForTest())  .Set(a => System.Console.WriteLine(a.ToString()))  .Set(a => a.p\_Int32 =2)  .Set(a => System.Console.WriteLine(a.ToString()))  .Set(a =>{  a.p\_String = "Привет всем";  System.Console.WriteLine(a.ToString());  })  .ToString()  .Set(a => System.Console.WriteLine(a))  ;  (new LocalClassForTest())  .Set(a => a.p\_Int32 = 9)//Название переменной:а  ;  (new LocalClassForTest()).Set((LocalClassForTest MyObject) => MyObject.p\_Int32 = 2);  } } |

На выходе мы получаем возможность от любого экземпляра класса создать его лямбда контекст без создания локальной переменной в ООП смысле. В любую цепочку возвращения с продолжением можем вклинить ещё один “.Set(a =>{})” Set-лямбда-контекст и что-то в нем сделать.

Прелесть этого неожиданно полученного за годы попыток эффекта в том что в любой момент времени можно переключиться между стилями Функ. и ООП при этом без малейших синтаксических потерь, потерь преимуществ обоих стилей, смены языка C# на F# и наоборот. Необходимость писать “.Set(a =>{})” – малая за это цена.

Необходимая теория:  
Лямбда выражения, делигаты,  
патерн функционального программирования "Возвращение с продолжением",  
патерн функционального программирования "Функции высших порядков".

За счет пространства имен “System”, функция доступна всегда.

\*Скептики могут усомниться в производительности такого программного кода.  
Использование лямбда выражений и стека вызова функций по времени исполнения много меньше, чем время отклика пользователя или объемы оптимизационных работ (рефакторинга) на современных проектах. Хронически делается выбор в сторону скорости разработки нежели внутреннего качества и простоты программного кода. Эксперименты показали (а их имеет смысл провести)(опыт многоголетнегого использования), что вызов лямбда контекста сравним (меньше чем) по времени с операцией преобразования числа в текст и обратно, или с десятью операциями сложения.

\*Рекомендании по повышению бастродействия:  
Не используйте глубокую степень вложенности лямбда контекстов. Более 6- 8 вложенных друг в друга лямбда выражений вызывают значительное замедление.  
Данный прием предназначен для использования на небольшой глубине вложенности в пределах текущей исполняемой функции. 1-3 уровень вложенности, если их более то это повод усомниться в качестве программного кода и начать его рефакторинг. Всегда можно написать проще.

\*Скептики могут усомниться в практической пользе этого приема.  
Факты говорят о следующем:  
\*\*Занимает меньше места  
При использовании этого приема не обязательно создавать локальные переменные. Лямбда контекст делает это за нас. Например  
(new LocalClassForTest())  
 .Set(a => a.p\_Int32 = 9)//Название переменной: а  
;  
Короткие названия переменных с компактной (неполной) формой их объявления как входных параметров лямбда выражения значительно сокращает объем программного кода.

\*\*Отвязывает программный код от названия переменных

Так ли уж ли важно как они называются, когда работа ведется в контексте?

При копировании программного кода в виде цепочки вызванных лямбда контекстов нет необходимости менять названия переменных.

Названия переменных существуют в виде одной буквы только в пределах лямбда контекста, который к тому же как правило не очень большой.

При необходимости никто не мешает Вам сделать следующее:  
(new LocalClassForTest()).Set((LocalClassForTest MyObject) => MyObject.p\_Int32 = 2);  
То есть в ущерб компактности применить объявление входных параметров лямбда выражения в полной форме.

Позволяет писать программные коды "в одну небольшую строчку", вместо половины страницы программного кода в классическом императивном стиле.

Такой программный код проще один раз увидеть, чем многократно листать несколько страниц программного кода.

\*Скептики могут усомниться в необходимости этого приема в C#. Да, можно всю жизнь смотреть под ноги и ни один раз не обратить свой взор к звездам. Зачем новые приемы и технологии, когда все и раньше работало. Технологический стек быстро меняется, повседневная ситуация напоминает разорение ткачей в эпоху изобретении ткацкого станка.   
У библиотеки уже более 300 скачиваний.  
На этот вопрос ответит только время.

Скептики могут упомянуть, что есть функциональный язык F#...

\*\*Во первых, не отрицаю пользу от существования F#, но проектов и вакансий C# куда больше.

\*\*Во вторых, процитирую (не дословно, но с совпадением смысла) разработчиков сообщества языка Julia.

"""Практика показала, что более живучим оказывается тот язык, который пользователю дает большую синтаксическую свободу, нежели загоняет их в узкие рамки!!!"""

Сейчас больших успехов достигает разработка проекта, если в нем разумно сочетаются больше одного стиля программирования.

\*\*В третих, как практика показывает следующее: "Если желаете замусорить язык, тогда создайте ещё одно ключевое слово! И так много раз)".

При написании этой библиотеки использовалось минимально возможное количество ключевых слов и минимально возможный объем программных кодов, ну или к этому стремимся.

Опыт изучения нашей библиотеки и знание LinQ гармонично дополнят друг друга. В C#, как в функциональном языке почти не останется ощутимых синтаксических пробелов.

Ну раз есть “.Set(a =>{})”, то требуется и Get.  
“.Get(a => {return 0;})” – позволяет вернуть именно в том формате в котором Вы хотели, и ли осуществить вычисления в переменных из замыканий/контекста.

|  |
| --- |
| //File Get\_T.cs  namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  //////////////////////////////////////////////////////////  public static class Get\_T  {  public static T1 Get<T, T1>(this T \_this, System.Func<T, T1> x) { return x(\_this); }  public static void Test()  {  System.Console.WriteLine(  (6).Cast\_AsObj().Cast\_As<System.Int32>()  .Get(a => "System.Int32=<" + a.ToString()+">")  )  ;  }  }  } |

По мотивам LinQ. Cast\_As и Cast\_AsObj – две функции, которые позволяют преобразовать System.Object в выбранный Вами конкретный тип и любой тип в System.Object. Сокращает время набора программного кода в конце цепочки вызовов лямбда контекста. Нет нужды бегать в самое начало цепочки и делать объектные обертки, тем более что такой код потеряет в читаемости.

|  |
| --- |
| Ext\_Cast\_T.cs |
| namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Runtime.CompilerServices;  using System.Text;  using System.Threading.Tasks;  ///////////////////////////////////////////////////////////  public static class Ext\_Cast\_T  {  public static T Cast\_As<T>(this System.Object \_this) => (T)\_this;  public static System.Object Cast\_AsObj<T>(this T \_this) => (System.Object)\_this;  public static void Test()  {  "Привет мир".Cast\_As<System.Object>();  var Obj = ((System.Object)"Привет мир");  "Привет мир".Cast\_As<System.Object>().Cast\_AsObj();  System.String Str = ((System.String)(System.Object)"Привет мир");  }  }  } |

А сейчас мы заново изобретаем ветвление.

|  |
| --- |
| SetIf\_T.cs |
| namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  ///////////////////////////////////////////////////////  ///////////////////////////////////////////////////////  public static class SetIf\_T  {  public static T SetIf<T>(  this T \_this  , System.Nullable<System.Boolean> \_Bool = null  , System.Action<T> \_f1 = null  , System.Action<T> \_f0 = null  , System.Func<T,System.Boolean> \_fBool = null  )  {  System.Boolean \_Boolean = false;  if (\_Bool != null) \_Boolean = \_Bool.Value;  if (\_fBool!=null) \_Boolean = \_fBool(\_this);  if (\_Boolean)  { if (\_f1 != null) \_f1(\_this); }  else  { if (\_f0 != null) \_f0(\_this); }  return \_this;  }  public static void Test()  {  {  System.Boolean \_Boolean = true;  \_Boolean.SetIf(\_Boolean, a => System.Console.WriteLine("<true>"), a => System.Console.WriteLine("<false>"));  //Равносилен  \_Boolean.SetIf(\_Bool:\_Boolean, \_f1: a => System.Console.WriteLine("<true>"), \_f0: a => System.Console.WriteLine("<false>"));  //Равносилен  \_Boolean.SetIf(\_Bool: null, \_f1: a => System.Console.WriteLine("<true>"), \_f0: a => System.Console.WriteLine("<false>"), \_fBool: a => a);  }  true.SetIf(\_fBool: a => a, \_f1: a => System.Console.WriteLine("<true>"), \_f0: a => System.Console.WriteLine("<false>"));  (new string[] { "1", "2", "3" })  .Set(b => b.ToList().ForEach(a => System.Console.WriteLine(a)))  .Select(a => System.Int32.Parse(a))  .OfType<System.Double>()  .SetIf(\_fBool:a=>a.Count()==0,\_f0:a=>throw new System.Exception("Eror:20200806\_2150:неудалось преобразовать к System.Double[]"))  //Равностильно, только обработчик ошибо наш...  .Cast<System.Double>()  //...  ; LinQToObject.  for (int i = 0; i < 100; i++) ;  foreach (var qwe in (new string[] { "1", "2", "3" })) ;  }  }  } |

А сейчас мы заново изобретаем ветвление. Не скрою, да это так. Мы шли от практической необходимости... Статистика использования такова что .Set/.SetIf/.Get/.GetIf как 25/5/1/1 в долях на вскидку. Хорошо что дело не дошло до изобретения цикла. Его изобрели до нас. Пример LinQToObject:  
(new string[] { "1", "2", "3" })  
 .Set(b => b.ToList().ForEach(a => System.Console.WriteLine(a)))  
;

Теперь “.GetIf” – он используется довольно редко, но без него порой бывает трудно обойтись. .SetIf/.GetIf используют прием пришедший из С++. Значения параметров по умолчанию. Они позволяют не создавать перегрузки функций и не плодить лишних сущьностей, а System.Nullable позволяет обернуть значимые типы и превратить их в ссылочные, чтобы они могли иметь значение null.

|  |
| --- |
| GetIf\_T.cs |
| namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  ////////////////////////////////////////////////////////////////  public static class GetIf\_T  {  public static T1 GetIf<T, T1>(  this T \_this  , System.Nullable<System.Boolean> \_Bool = null  , System.Func<T,T1> \_f1 = null  , System.Func<T, T1> \_f0 = null  , System.Func<T, System.Boolean> \_fBool = null  )  {  System.Boolean \_Boolean = false;  if (\_Bool != null) \_Boolean = \_Bool.Value;  if (\_fBool != null) \_Boolean = \_fBool(\_this);  if (\_Boolean)  {  if (\_f1 != null) return \_f1(\_this);  }  else  {  if (\_f0 != null) return \_f0(\_this);  }  System.Func<T, T1> \_fail = null;  return \_fail(\_this);  }  public static void Test()  {  {  System.Boolean \_Boolean = true;  System.Console.WriteLine(\_Boolean.GetIf(\_Boolean, a => "<true>", a => "<false>"));  //Равносилен  System.Console.WriteLine(\_Boolean.GetIf(\_Bool: \_Boolean, \_f1: a => "<true>", \_f0: a => "<false>"));  //Равносилен  System.Console.WriteLine(\_Boolean.GetIf(\_Bool: null, \_f1: a => "<true>", \_f0: a => "<false>",\_fBool:a=>a));  }  System.Console.WriteLine(true.GetIf(\_fBool:a=>a,\_f1: a => "<true>", \_f0: a => "<false>"));  //Актуальность:  //Данная функция - гибрид .Get(a =>...) и .SetIf<T>(\_Bool:...,\_f1:...,\_f0:...,\_fBool:...)  //и аргонично заполняет бреш в функциональных программных кодах.  }  }  } |

INewEable.cs – интерфейс –конструктор

|  |
| --- |
| INewEable.cs |
| namespace System  {  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  //////////////////////////////////////////////////////////////  public interface INewEable  {System.Object New();}  } |

Ext\_Cast\_T.cs – расширение, позволяет делать “обертки” объектов без перехода в начало строки.

|  |
| --- |
| Ext\_Cast\_T.cs |
| "Привет мир".Cast\_AsObj().Cast\_As<System.String>();  ((System.String)(System.Object)"Привет мир"); |

Ext\_List\_T.cs – расширение, операции над списками, принцип возвращения с продолжением.

01\_System/Diagnostics/TestLast.cs – инструменты для запуска последней редактируемой тестовой функции. Пример из функции тест запускает через рефлексию статическую тестовую функцию, отмеченную атрибутом System.Diagnostics.Att\_TestLast с самой поздней датой…

|  |
| --- |
| 01\_System/Diagnostics/TestLast.cs |
| public static void Test()  {  System.Console.WriteLine(System.Diagnostics.StackTracer.Get\_STS());  System.Diagnostics.TestLast.Do();  }  //…  [System.Diagnostics.Att\_TestLast(\_year: 2021, \_month: 11, \_day: 17, \_hour: 15, \_minute: 06, \_second: 0, \_millisecond: 0)]  public static void Test\_1()  {  System.Console.WriteLine(System.Diagnostics.StackTracer.Get\_STS());  } |

01\_System/Diagnostics/StackTracer.cs – инструменты для анализа стека вызова функций, в основном для программного определений названия функции из которой она было вызвано.

|  |
| --- |
| 01\_System/Diagnostics/StackTracer.cs |
| System.Console.WriteLine(System.Diagnostics.StackTracer.Get\_STS()); |

01\_System/Diagnostics/ Ext\_Stopwatch.cs – расширение методов класса System.Diagnostics.Stopwatch.

01\_System/VirtualConsole/Ext\_Get\_ReadLine.cs – расширение, позволяет читать с консоли конкретный тип данных (все примитивные типы), через заготовленную подпрограмму. Это снимает необходимость бесконечных конвертировании из текста в тип данных и наоборот при вводе с консоли.

01\_System/VirtualConsole/VirtualConsole.cs – виртуальная консоль, которая позволяет контекстно ориентированно расширить строку и перенаправить ввод вывод данных по лямбда выражению.   
Также позволяет писать в стиле: ” "Привет мир".WriteLine().WriteLine().ReadLine();”.

01\_System/VirtualConsole/GUIVC.cs – консольное меню общего назначения из одного класса, далее для консольных приложений используется как основной способ построения интерфейсов.

|  |
| --- |
| 01\_System/VirtualConsole/GUIVC.cs |
| new GUIVC()  .Set(\_Title: "Титульник Меню",\_IsRepeater:true  , \_GUIVC\_S: new List<GUIVC>() {  new GUIVC().Set(\_Title:"Выход",\_Act:a=>a.p\_Sender.p\_IsRepeater=false)  , new GUIVC().Set(\_Title:"Выбор 1")  , new GUIVC().Set(\_Title:"Выбор 2")  }  )  .Do()  ;  new GUIVC()  .Set(\_Title: "Титульник Меню"  , \_GUIVC\_S: new List<GUIVC>() {  new GUIVC().Set(\_Title:"ДА",\_Act:a=>a.p\_Title.WriteLine())  , new GUIVC().Set(\_Title:"НЕТ",\_Act:a=>a.p\_Title.WriteLine())  , new GUIVC().Set(\_Title:"Сомневаюсь")  }  )  .Do()  ; |

Итак, мы рассмотрели только одну папку, а допустимый объем статьи подходит к концу.

02\_System\_T – папка, в которой реализованы механизмы создания новых экземпляров объектов, создания глубоких копий объектов, рефлексивного чтения и вывода на экран любых иерархических объектов. Механизмы реализованы в максимально возможном объем виде…

03\_IMap – папка, в которой реализован механизм привязывания объекта к средству визуализации, или средству исполнения, и запуску последнего, реализованного средства визуализации. Механизм похож на контракт данных, но реализован на чистой рефлексии, компактнее и возможно проще.

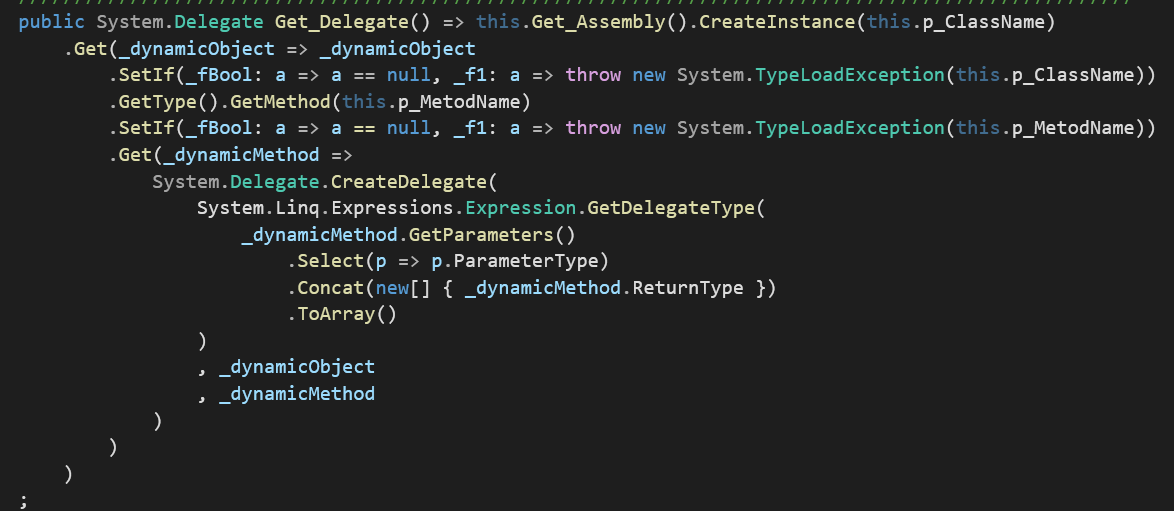
04\_CodeGearER - папка, в которой реализован механизм исполнения по горячему программных кодов, полученных в виде текстовой переменной.

|  |
| --- |
| CodeGearER |
| System.Diagnostics.StackTracer.Get\_STS().WriteLine();  System.String \_NameSpace = "Space\_"  + System.DateTime.Now.ToString("yyyyMMdd\_ssmmhh\_ffffff");  System.String \_NameClass = "Class\_"  + System.DateTime.Now.ToString("yyyyMMdd\_ssmmhh\_ffffff");  System.String \_NameMetod = "Metod\_"  + System.DateTime.Now.ToString("yyyyMMdd\_ssmmhh\_ffffff");  (@"  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  ////////////////////////////////////////////////  using System.Globalization;  namespace " + \_NameSpace + @"  {  public static class " + \_NameClass + @"  {  public static void "+\_NameMetod+ @"()  {  "+"\""+"Привет Котишка.Как твои дела ? Хорошо ли ты кушаешь ? " + "\"" + @".WriteLine();  }  }  }  ")  .Get\_Assembly()  .GetType(\_NameSpace + "." + \_NameClass)  .GetMethod(\_NameMetod)  .CreateDelegate(typeof(System.Action))  .Cast\_As<System.Action>()  ()  ; |

12\_CMDoor\_V0 - папка, в которой реализован механизм обращения к консоли CMD через временные не повторяющиеся файлы.

|  |
| --- |
| 12\_CMDoor\_V0 |
| ">>>:System.CMD.CMDoor\_V0.Test\_2():>>>".WriteLine();  System.CMD.CMDoor\_V0.Get\_Resalt(\_CMDInPutSkript:  @"@echo off  echo HelloWorld>>Text.txt  echo Привет\_мир>>Text.txt  echo Закройте без сохранения и программа будет работать дальше>>Text.txt  notepad Text.txt  DEL Text.txt  "  ).WriteLine();  "<<<:System.CMD.CMDoor\_V0.Test\_2():<<<".WriteLine(); |

Как ни странно, основная цель с которой создавался CMDoor – это реализация через CMD/DOS/\*.BAT для открытия временного текстового файла для ввода параметров дальнейшей работы программы. При закрытии окна блокнота, программы C#, продолжает свое исполнение.

Компактно, дешево, сердито, и без технологических наворотов: “Блокнот как пользовательский интерфейс“.  
CMDoor реализован в соответствующем:

PS.  
Авторские технологические идеи дотируются 17-18 годом.  
Программная реализация дотируется 19 и началом 20 года.  
Выставляемые программные коды размещены в виде GitHub репозитория со свободным доступом. “ https://github.com/IvanSibirevV2/ArgeCode”

Выставляемые программные коды в дальнейшем будут оформлены в виде nuget package для Visual Studio. “https://www.nuget.org/packages/ArgeCode/”

Документация планируется только русскоязычная.  
Лицензия: свободно распространение, не для продажи, в учебных целях.  
(Не пытайтесь продавать и видоизменять этот программный код он будет в свободном доступе! Все для Вас.) Мы открыты к идейным предложениям. Будем их реализовывать…

Литература.  
#001) Архитектура IA64/ Николай Дорофеев (16 февраля 1998) Электронный ресурс.  
Url (24.09.2020): <https://www.ixbt.com/cpu/ia64.html>

# #002) Краткий очерк истории Lisp машин/Автор: пользователь “holmuk“ (14 августа 2013)Url (24.09.2020): <https://habr.com/ru/post/190082/>

# #003) 64\_бита / Url (24.09.2020): <https://ru.wikipedia.org/wiki/64_бита>

# #004) «Забытые» парадигмы программирования/ Автор: пользователь “domu“/ Url (24.09.2020):https://habr.com/ru/post/223253/

# #005) F Sharp-ВикипедиЯ/ Url (24.09.2020):https://ru.wikipedia.org/wiki/F\_Sharp

# #006) LINQ - язык интегрированных запросов / Url (24.09.2020): https://professorweb.ru/my/LINQ/base/level1/info\_linq.php

# #007) Магическое число семь плюс-минус два / Url (24.09.2020): <https://ru.wikipedia.org/wiki/Магическое_число_семь_плюс-минус_два>

# #008) Generics (C# Programming Guide) / Url (24.09.2020): https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/

# #009) Методы расширения (Руководство по программированию в C#) / Url (24.09.2020): <https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>

# #010)