Аннотация.

Итак. Здравствуйте.  
Вашему вниманию предлагается многоцелевая библиотека “ArgeCode.dll” написанная на языке С#.  
Понимаю, на сегодняшний день наработками и новыми библиотеками никого не удивишь. Причина тому кроется в необычайной пестроте инструментария программирования.

Каждый программист остается на едине с собой перед информационной лавиной из многочисленной рекламы как новомодных технологий крупных компаний, так и наработок, библиотек и фреймворков отдельных энтузиастов. Пишется всем миром, а изучать приходится одному. Трудно различить ремейки не самых удачных идей от действительно чего-то стоящего. Все сливается в один серый шум.

Причины за использование библиотеки “ArgeCode.dll”.  
Эта библиотека является той самой неповторимой и той самой недостающей синтаксической прослойкой/надстройкой, что … Позволяет писать программный код C# (без #F) как в функциональном стиле так и в императивном, без ущерба для синтаксиса и архитектуры приложения, играючи переключаясь между ними.

Мы добавили в C# новый синтаксис. А именно контекстно ориентированные генетические методы, в том числе и на основе рефлексии. Их можно вызывать от любого экземпляра любого объекта и строить новые миры очень компактного программного кода с высокой логической насыщенностью. При желании обычный объектное ориентированный императивный код в полтора и более раза.

Выводим понятие контекстно ориентированного программирования на новый уровень. Если раньше контекстно ориентированное программирование воспринималось как расширение существующих классов объектно-ориентированного программирования статическими функциями. То сейчас мы можем программировать цепочками последовательно запускаемых функций и лямбда выражений без использования локальных переменных, вообще.

Библиотека построена по принципу последовательных блоков, каждый из которых может использовать предыдущие, но не может быть в них использован. Исходные программные коды выложены на GitHub и NuGet. Они снабжены многочисленными комментариями. По этому, программный код с первой страницы и по последнюю читается как научно фантастический роман. Для читателя в плоть до последней страницы быстро но постепенно приподнимается занавес над ответом на вопрос для чего все это может быть использовано. И по окончании каждой главы (каждого блока) ответ на этот вопрос будет звучать сильно по разному. По этой причине нельзя судить или рассматривать каждый компонент или механизм по отдельности. Библиотека не есть сама цель, но часть инфраструктурного кода будущих проектов! Надеемся найти отклик в Вашей душе!

Этой библиотекой решаются и дополнительные задачи.  
“Глубокое рефлексивное копирование базовых типов, структур, объектов“. Эта задача ранее командой Microsoft не была решена. Ими ранее говорилось: вот Вам интерфейс IClonabel и крутитесь как хотите. По аналогии с ISerializable и многими другими интерфейсами…  
Их реализация под каждый отдельный случай и класс сильно увеличивает время разработки.  
Хотелось бы иметь под рукой базовые функции инфраструктурного кода, которые бы снимали с программиста задачи по написанию бессмысленного многостраничного кода под 2 или 3 тысячи строк, только чтобы повторить ранее достигнутый эффект, но в другом контексте.

“Рефлексивная интроспекция в процессе выполнения программного кода” в Веб программировании

возникает ситуация, когда в интерфейс, в обычный или псевдо-динамический объект прилетает класс неизвестного содержимого. Да это сделано для защиты программного обеспечения от несанкционированного доступа например к данным сервера. Но вопрос, что если Вы разработчик и требуется воспользоваться недокументированными, от слова вообще, возможностями этих классах на официальном основании. В целях отладки создан инструмент который позволяет выводить в консоль рефлексивные члены объектов со всеми их параметрами и свойствами. Рядом реализован младший брат этого инструмента, позволяющий с консоли вводить вообще любые объекты. Студентами он наивно используется, чтобы вводить не строки, а сразу дроби или структуры из нескольких параметров. И еще раз, это всего одна функция.

“Запуск последней тестовой статической функции” или запуск последнего теста. Но позволяет запустить функцию, которую Вы только что редактировали. Экономит время разработки на переходах между окнами и позволяет “Не отрывать рук от клавиатуры”. Рефлексия чистой воды.

“Запуск программного кода из текстовой переменной”. Во всех приличных языках программирования давно уже есть. А в C# реализован из под палки. Теперь любую строковую переменную последовательностью контекстно ориентированных функций можно нечаянно скомпилировать, добавить к уже работающему приложению, получить делегат и запустить. “Нечаянно” – это не когда не было так просто как теперь (благодаря нам), убедитесь в этом сами...

“Мапинг объекта и пользовательских графических интерфейсов и запуск наиболее позднего из них” – примерно та же вещь что и контракт данных, но по другому реализована. Требуется унаследовать класс графического интерфейса от генетического (”Generic” перевод с англ. – обобщенный, в данном случае применено как устоявшийся литературный жаргон одной из субкультур разработчиков ПО) интерфейса с указанием редактируемого/отображаемого объекта и указать дату редактирования пользовательского интерфейса. Поверьте внешне выглядит крайне просто.

“Виртуальной консоль”. Функции чтения записи виртуальной консоли через лямбда выражения по умолчанию смотрят на обычную консоль C#, при этом они – это контекстно ориентированные расширения строки. То есть все тот же программный код можно перенаправить в лог файл с чтением входных данных из другого файла, или на удаленного пользователя. Звучит страшно, но реализовано так что запутаться там просто не в чем.

“CMDor” - C# позволяет запускать “\*.bat” под Windows и “\*.sln” под Lunix. Обычная реализация содержит ошибки. Делаем обертку, которая создает временны “\*.bat” исполняет его в перенаправлением потоков ввода/вывода, если нужно, и подчищает его за собой.

Забавное следствие!!! Теперь в качестве пользовательского интерфейса можно использовать обычный блокнот: создаем файл, в комментариях задаем вопрос пользователю, открываем его в блокноте, дожидаемся когда пользователь введет одну или несколько настроек через блокнот в файл, а затем сохранит и закроет блокнот, программа считывает параметры, подтирает за собой файл и продолжает работать дальше. Как видите, можно обойтись и без написания графического интерфейса.

Краткая предыстория вопроса. Часть первая.

В период с 2000 по 2015 годы на просторах российского интернета разразилась баталия на тему “Кто лучше?”, “Императивное программирование или Функциональное программирование?”. Пред баталией расклад сил в стане “императившиков” был примерно следующий: быстрое обучение и вхождение в программирование всех от мало до велика за счет теорий объектно-ориентированной парадигмы и языков C++, C#, при высокой стоимости и медленной разработки крупных проектов за счет построения архитектуры, поддержки программного кода больших объемов и малой информативности. Также многочисленные паттерны проектирования ООП при их неоднозначной интерпретации и слепой веры новобранцев в их непоколебимость.

В стане функциональных программистов исторически сложилось так что... Подобное программирование использовалось в академической среде и требует дополнительного или даже специализированного образования. Количество паттернов программирования не превосходит одного десятка.

Типичные функциональные языки программирования LISP, Haskell, R, Mathcad, Matlab, Julia – общемировой процент коммерческих проектов с трудом переваливает в разные годы порог в 1-4% процента. Для сравнения, один C# - порог в 4 процента, Java порог примерно в 40%.

Приведём основопологающие определения.

Императивная парадигма программирования – программный код состоит из последовательно исполняемых инструкций, изменяющих состояние программы. Дальнейшее развитие императивного программирования можно классифицировать по принципу изменяемого контекста исполнения программных кодов. Структурное программирование - изменяемый контекст это отдельная структура в оперативной памяти. Процедурное программирование – локальные переменные и структуры входных данных, реже глобальные переменные. Объектно-ориентированное программирование – изменяемые объекты в оперативной памяти, обладающие модификаторами доступа, процедурами и функциями изменяющие состояние объекта

<https://habr.com/ru/post/223253/>

Функциональная парадигма программирования, в первоисточниках подразумевает: что “все есть функция”; любые данные являются неизменяемой структурой, передаваемой как входная или выходная переменная функции; у функции есть только одна входная и выходная переменная, это кортеж, то есть неизменяемая структура из других типов данных; понятие состояния объекта не используется вообще.

<https://bestprogrammer.ru/programmirovanie-i-razrabotka/yazyki-funktsionalnogo-programmirovaniya-polnoe-rukovodstvo>

Теперь, разберёмся с синтаксисом и функциями библиотеки

Начать рекомендуется с папки “01\_System”.  
Файл “Set\_T.cs”, добавляет контекстно ориентированную генетическую функция с сигнатурой “T Set<T>(this T \_this, System.Action<T> x)”. Пространство имен, для простоты, “System”, для того чтобы при вызове не приходилось тратить лишнее время на обращение к пространству имен через “using”. Функция является, генетической или обобщенной (), контекстно ориентированной, создана так что её можно вызвать от любого экземпляра любого типа данных кроме “dynamic”, да и эта проблема решается обертыванием, например, в “System.Object”, “(System.Object)dynamic\_Value”.

Функция принимает внутрь себя лямбда-выражение, которое применяется к объекту, от которого она вызвана. Под “контекстом лямбда-выражения” обычно понимается “замыкание”, то есть доступность для лямбда-выражения, всех переменных на момент его создания, так и после, уже за приделами области видимости переменных, во время последующего существования лямбда-выражения. В наших текстах это определение имеет силу.

Вводим новое определение “лямбда-констест” – (замыкание в узком смысле) - это лямбда-выражение, созданное и запущенное в контексте экземпляра объекта, запущенное как функция объекта с очень коротким временем существования, направлена преимущественно на изменение внутреннего состояния объекта. Позволяет строить сколь угодно длинные цепочки лямбда-выражений. Каждый лямбда-контекст возвращает свой контекст вызова, экземпляр объекта от которого лямбда контекст был вызван (удобная вариация на тему паттерна возвращения с продолжением).

Пример:

|  |
| --- |
| (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))  ; |

Как видно из примера лямбда контекст или Set функция позволяет через замыкание в широком смысле обращаться к глобальным переменным или функциям, например для отладки, позволяет через замыкание в узком смысле обращаться к контексту вызова, например, чтобы изменять его. Деление на замыкание в широком и узком смысле – условно. Как следствие мы можем писать код как в функциональном стиле, так и в императивном объектно-ориентированном стиле программирования, переключаясь между ними без архитектурных проблем, со сменой языка программирования или глобальным изменением структуры кода. Любые отклонения от каноничных определений функционального и императивного программирования допущены (по просьбам трудящихся) в угоду ускорения и повышения самого комфорта разработки.

Файл “Get\_T.cs” – отвечает на вопрос, а что если захотим возвратить результат изменив его. Это вариация на тему мутаторов и селекторов, только для экземпляра объекта.

|  |
| --- |
| T1 Get<T, T1>(this T \_this, System.Func<T, T1> x) |
| System.Console.WriteLine(  (6).Cast\_AsObj().Cast\_As<System.Int32>()  .Get(a => "System.Int32=<" + a.ToString()+">") )  ; |

Файл “SetIf\_T.cs” – позволяет использовать ветвление.  
Да, есть обычное ветвление, есть сокращенное ветвление ((a==0)?(true):(false)). В новых открывающихся реалиях все эти синтаксические конструкции не являются удобными. По этому вот ещё один вариант:

|  |
| --- |
| 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;  } |

Пример использования, если флаг ветвления заранее известен:

|  |
| --- |
| System.Boolean \_Boolean = true;  \_Boolean.SetIf(\_Boolean  , a => System.Console.WriteLine("<true>")  , a => System.Console.WriteLine("<false>")  );  \_Boolean.SetIf(\_Bool:\_Boolean  , \_f1: a => System.Console.WriteLine("<true>")  , \_f0: a => System.Console.WriteLine("<false>")  ); |

Пример использования, если флаг ветвления требуется вычислить:

|  |
| --- |
| (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[]"))  .Cast<System.Double>()  ; |

Файл “GetIf\_T.cs” – по аналогии с “SetIf\_T.cs”.

Пример использования, если флаг ветвления заранее известен:

|  |
| --- |
| System.Boolean \_Boolean = true;  System.Console.WriteLine(\_Boolean.GetIf(\_Boolean, a => "<true>", a => "<false>"));  System.Console.WriteLine(\_Boolean.GetIf(\_Bool: \_Boolean, \_f1: a => "<true>", \_f0: a => "<false>"));  System.Console.WriteLine(\_Boolean.GetIf(\_Bool: null, \_f1: a => "<true>", \_f0: a => "<false>",\_fBool:a=>a)); |

Пример использования, если флаг ветвления требуется вычислить:

|  |
| --- |
| System.Console.WriteLine(true.GetIf(\_fBool:a=>a,\_f1: a => "<true>", \_f0: a => "<false>")); |

Как видите “Set,Get,SetIf,GetIf” – все это лямбда контекст. Позволяет писать крайне компактный функциональный код с использованием контекстно ориентированных принципов. Это позволяет потратить на программный код в полтора раза меньше строк, чем при классическом императивном программировании. Это означает что код большинства функций будет менее страницы или даже не более нескольких строк. Это увеличивает объем логики, которой можно прочитать с одной страницы монитора. Все к лучшему.

Если есть желание, то можно избавиться вообще от локальных переменных, используя только контекст, то есть только входные и выходные значения лямбда выражений. В таком случае все программирование ведутся только по контексту вызова. Используя это мы делаем ещё больший шаг в сторону мира функционального программирования.  
Этот новый механизм не в коем случае не являет заменой всего что было ранее. Если писать все только через лямбда контекст, то Вы получите анти паттерн “Золотой молоток”, то есть решение всех проблем одним способом, это будет по крайней мере странно. Все должно идти от практической пользы. При написании программного кода с высокой логической насыщенностью практическая польза очевидна. Это сокрушение объемов кода. Это сокращение времени разработки. Правда это потребует от программиста дополнительного опыта и сноровки.

Продолжаем

Дальше не читать…

Магическое число 7. иЛИ ЧИСЛО ДжорджА МиллерА =)

-Психология, за раз человек способен запомнить 7+-2 объекта.

При программировании больших моноблоков это проблема.

Когда один уровень иерархии объектов разрастается больше числа Джорджа Миллера код перестаёт быть томным (понятным).

////////////////////////////////////////////////////////////////////

Обычно иерархия проекта следующая

Решение/Проект/Папки+классы

////////////////////////////////////////////////////////////////////

Проблема.

Написание проеста на скорую руку путем вкрапления небольших изменений программного кода без контроля архитектуры

и рефакторинга приводит к лавинообразному разрастанию прогрмаммного кода.

Проведу аналогию.

Это как большой кристалл соли вырашеный на ниточке в соленом растворе в стакане.

Невозможно получить гомогенную структуру с небольшим колличеством чистых и четких граней геометрии кристалла.

Все время будет получаться бешеное нагромождение очень маленьких кристаликов. Тронешь один и все посыпется или пойдет трешенами.

////////////////////////////////////////////////////////////////////

Предлагаю использовать следующее соглашение по написанию этого проекта.

\*Каждое следующее решение будет состоять из вызова и использования предудущих проектов.

\*Каждый проект будет обладать названием [Год][Месяц][День]\_[Час][Минута](Создания и начала проекта IDE)\_[Название проекта(Не слишком длинное)]

\*Каждый проект использует линейную комбинацию предыдущих проектов

\*Каждый проект содержит в корне ReadMe.txt

\*Каждый проект (В целях предотвращения разростания сущиностей более числа Миллера) содержит папки

с названием Unit000 (Модуль порядковый номер создания внутри проекта)

В пределах проекта допустимо альтернативное название модулей, главное чтобы названия были единообразными.

Например, \*\*(2 разряда 16 ричного числа)\_[Название папки]

\*Каждый Unit\*\*\* использует линейную комбинацию предыдущих проектов, предыдущих модулей внутри проекта

И при необходимости содержит контекстно ориентированные расширения для разных контекстов (пространств имен)

\*Каждый Unit\*\*\* содержит в своем корне ReadMe.txt

\*У каджого класса и у некоторых функций могут быть свои методы Test(), демонстрирующие пример работы соответствующей сущиности

\*Методы Test() выводят промежуточные и конечные результаты в консоль не напрямую а только черезнадстройку консоли,

которую можно перенаправить в файл (для моульных тестов, но в них особо нет необходимости) или в браузер.

\*При необходисости каждый проект можно разбить на линейную комбинацию его модулей оформленцю в виде проектов (Будем называть это восходящая трансляция)

и наоборот (Будем называть это нисходящая трансляция).

\*При необходимости каждый модуль можно разбить на линейную комбинацию его подмодулей

(оформленную в виде модулей)(Будем называть это восходящая трансляция)

и наоборот(Будем называть это нисходящая трансляция)

\*

Так можно избежать проклятья Миллера 7+-2 и нагромождения всего этого дела

////////////////////////////////////////////////////////////////////

Как рефакторить рефакторим сначала первый модуль, затем переписываем все остальные его использования.

При рефакторинге допустима трансляция в проекты с более поздней датой создания.

При рефакторинге допустима восходящая трансляция с последующей нисходящей трансляцией

////////////////////////////////////////////////////////////////////