Построение кода на лету в Microsoft .NET

Вашему вниманию предлагается статья, посвященная созданию кода «на лету» в Microsoft .Net. В ней рассматриваются вопросы построения кода непосредственно на MSIL (Microsoft Intermediate Language) и компиляции «на лету», а также язык моделей CodeDOM.

"Высшая ценность знания, абстрактного познания, заключается в том, что его можно передавать другим и, закрепив, сохранять: лишь благодаря этому оно делается столь неоценимо важным для практики."
Артур Шопенгауэр. Мир как воля и представление.

1 Введение или панегирик объектно-ориентированному программированию (ООП)

Совсем недавно я обсуждал с одним из моих коллег вопросы правильной разработки, документирования и сопровождения. Специальность моего коллеги не имеет прямого отношения к программированию, поэтому я не очень удивился, услышав что-то о необходимости рисования "блок-схем". Да, нас так раньше учили, и не только будущих программистов, а и электронщиков, и конструкторов, и технологов. Меня больше удивил мой непосредственный начальник, который, не будучи профессионалом в области ПО, тут же заметил, что рисование блок-схем это "вчерашний день". Я подтвердил это, уточнив, что, скорее всего даже не вчерашний, а позавчерашний.
Как много изменилось с тех пор, когда рисование блок-схем было необходимым атрибутом разработки. Изменилась сама парадигма программирования. Так, программы из процедурно-ориентированных стали объектно-ориентированными, да и языки программирования стали совсем другими. Результатом яркой борьбы за "правильный стандарт языка правильной разработки и документирования ПО" явился UML. Язык UML (Unified Modeling Language) предназначен для формализации (своего рода абстрагирования) того, что программисты стали помещать в программы, построенные на базе объектно-ориентированной парадигмы. То есть разработчики UML создали новую систему обозначений, основанную на диаграммах, позволяющую формулировать идеи, закладываемые в программу, и таким образом, обеспечивать процесс документирования и сопровождения ПО. Одной из проблем стало то, что UML предлагает использовать около десятка типов диаграмм вместо ушедшей в небытие блок-схемы. Надо заметить, что "отцы основатели" ([1]), не настаивают на создании "всех диаграмм" для "всех частей программной системы" и это облегчает участь разработчиков ПО. С другой стороны, этот подход стимулировал создание специальности архитектора ПО, одной из обязанностей которого является именно процесс создание диаграмм на языке UML, "позволяющий формулировать мысли и общаться с другими разработчиками" [2]. Усложнив с помощью UML процесс разработки/сопровождения, программисты занялись упрощением того же процесса. В результате появилась новая технология создания ПО - экстремальное программирование - XP (eXtreme Programming). Одна из идей экстремального программирования - это то, что "исходный код – это и есть документация на программу". В действительности, в этот период времени различные группы разработчиков разрабатывали ряд технологий, идеологически похожих на XP. И поскольку идеи были схожи, то вначале 2001 года идеологи этих направлений собрались вместе и создали Manifesto for Agile Software Development (Манифест гибкой разработки ПО). Я не стану перечислять всех участников, но то, что среди них были Мартин Фаулер (им написаны ряд книг об UML и рефакторинге [4], ставшие уже классическими), Кент Бек и Ворд Канингем (собственно говоря, они и разработали экстремальное программирование) говорит о том, это не была случайная встреча и намерения были самые серьезные. Созданный ими манифест, хоть и выражался в простых лозунгах, но в дальнейшем привел к созданию Agile Modeling, Agile Database Techniques и других технологий. Сегодняшние тенденции сохранили идеи гибкости, и даже системы, работающие с UML, позволяют генерировать код из диаграмм языка UML. Это направление перешагнуло через границы технологии создания ПО и используется непосредственно в программировании. Я хотел бы заметить, что Microsoft .NET создавалась примерно в тот же период времени, и на создание этой технологии влияли те же идеи, что и на формализацию объектно-ориентированного подхода, с одной стороны, и различных гибких, динамических технологий, с другой.
В этом контексте, при создании, построении и выполнении приложений в среде Microsoft .NET, наиболее интересными и важными являются следующие вопросы:

- Возможности динамической генерация кода на промежуточном языке MSIL во время выполнения программы;
- Поддержка языков высокого уровня (C#, Visual Basic, JScript) непосредственно в библиотеке классов Microsoft .NET;
- Поддержка модели единого базового объектно-ориентированного языка – CodeDOM;

- Возможности Microsoft .NET в контексте принципа – создал, построил, запустил.

Рассматривая особенности динамического создания кода, невозможно не учитывать аспекты объектно-ориентированного программирования, которые накладывают особый отпечаток на все, что происходит в области программного обеспечения. Можно не сомневаться, что если в настоящее время кто-либо выступает против парадигмы объектно-ориентированного программирования, то в лучшем случае он заслужит подозрительные взгляды коллег, а в худшем – сомнительную славу нигилиста и диссидента. Я вспоминаю те времена, когда такие же несокрушимые позиции занимало структурное программирование на базе процедурных языков. Помнится, что много времени и сил занимали дискуссии о допустимости или не допустимости использования оператора goto в программах. Сейчас это кому-то может показаться смешным. Но я помню также, что именно тогда немало интересных работ по теории и практике программирования были написаны такими замечательными людьми как Дейкстра, Хоар, Йордан (я до сих пор восторгаюсь великолепной книгой Хоара о взаимодействии последовательных процессов [3]). И все же: время пришло, а процедурные языки куда-то ушли, оставляя свое место для языков, соответствующих парадигме объектно-ориентированного программирования. Надолго ли? А с другой стороны, ведь системы баз данных так до конца и не приняли объектный подход, и многочисленные попытки создания СУБД на основе объектов разбились о несокрушимую твердь SQL. Конечно, продукты, представляющие собой разработки в области объектно-ориентированных баз данных существуют, но им и по объему рынка и по распространенности не сравнится с Oracle, DB2, SQL Server. Какой там “объектный SQL”, что это такое, это должно быть полный абсурд. Впрочем, может и не удивительно, что SQL напоминает чем-то Кобол. Подобная мысль покажется кощунством. А с другой стороны, почему бы и нет. Разве один из ведущих разработчиков SQL был не Дейт, который работал в IBM, которая столько положила в Кобол? Хотя, скорее всего, это сходство внешнее и ассоциативное. На самом деле, именно крепость и незыблемость SQL (заметьте, этому языку более тридцати лет) наводит на мысль, что есть самые-самые высокие вершины, выше которых ничего уже нет. И почему бы объектно-ориентированному программированию также не являться одной из таких великих и непревзойденных вершин? Не факт. Отнюдь не уверен, что будущие системы управления знаниями не потребуют какого-нибудь принципиально нового подхода, как его ни называй – то ли декларативный, то ли когнитивный. Почему бы и нет. Все впереди. Поезд идет на восток. Вперед. Двери закрываются. Следующая станция…

Хотя нет, коллеги, давайте рванем стоп-кран. Мы все еще на станции Microsoft .NET и объектно-ориентированный подход нам здесь очень даже пригодится. Ведь Microsoft .NET просто не может существовать без объектно-ориентированного подхода. Убери объектно-ориентированный подход, и Microsoft .NET обрушится с грохотом и треском. Самое интересное, что это относится и к SQL. В частности в SQL Server 2005 с помощью программ, написанных в Visual Studio 2005, язык SQL обретает новые интересные возможности. Самая интересная из них, обещающая дальнейшее развитие, это возможность создание пользовательских типов данных (объектно-ориентированных) с помощью VS2005 и дальнейшее использование этих типов в операторах SQL. Интересно, что одновременно была создана новая технология LINQ, выполняющая противоположное действие – включение операторов SQL-типа в программы, написанные на языках VS2005. Создается впечатление, что Microsoft .NET (будь то библиотека классов, стандартная система типов, модель базового объектного языка, собственно конкретные языки, среда исполнения CLR и т.д.) буквально соткана из парадигмы объектного программирования. Стоит ли теперь удивляться, если той же парадигме соответствует даже MSIL – “язык двоичных файлов” Microsoft .NET?

2 Создание кода на промежуточном языке

“Следует также отличать прекрасные предметы от красоты
предметов, увиденных издали (которые из-за расстояния
часто не могут быть рассмотрены).”
Иммануил Кант. Критика способности суждения.

“Язык двоичных файлов” Microsoft .NET это отнюдь не “машинный” язык, это именно промежуточный язык и это следует из его названия - MSIL (Microsoft Intermediate Language). Он “больше”, чем “машинный” язык уже тем, что он объектно-ориентирован. Но поскольку это не “машинный” язык, то при выполнении он должен или интерпретироваться (так сделано в Java) или компилироваться (как и сделано в Microsoft .NET). Перед первым выполнением какого-либо метода специальный JIT-компилятор (just-in-time compiler) компилирует метод с промежуточного языка в машинный, и этот результат компиляции запоминается в памяти. Это позволяет не делать повторных компиляций при повторных вызовах метода. Структура MSIL хотя и напоминает машинный язык, но в то же время является объектно-ориентированной. Но, конечно же, близость промежуточного языка к машинному языку делает весьма затруднительным и неудобным

программирование на нем. Можно ли программировать на MSIL? Конечно, хоть это и неудобно, но более занимательным является вопрос о возможности динамического создания (и выполнения) кода “на лету” в своей выполняющейся программе на языке MSIL. Рассмотрим небольшую программу, написанная на C# - HelloFromMSIL (Листинг 1). Откомпилируем ее, а потом рассмотрим результат компиляции (метод Main) с помощью дизассемблера ILDASM (Рисунок 1). В данном конкретном случае кажется очевидным, что код на MSIL выглядит ничуть не сложнее, чем код на C#.

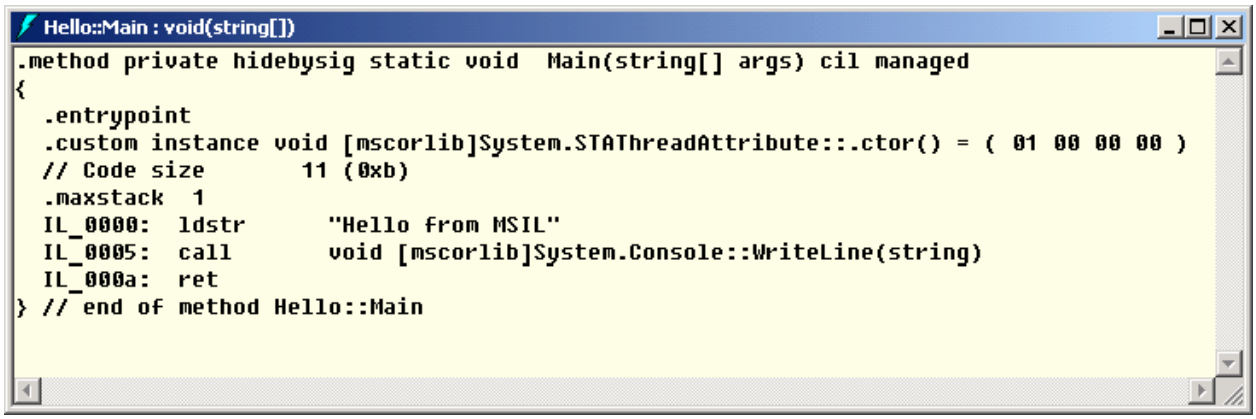

Рисунок 1. Код (MSIL), полученный дизассемблером ILDASM.

Рассмотрим, как можно создавать такого типа код (непосредственно на MSIL) “на лету”. Прежде всего, надо создать динамическую сборку (класс AssemblyBuilder из пространства System.Reflection.Emit предназначен как раз для этого). В сборке надо описать класс, а в этом классе надо предусмотреть конструкторы, методы, свойства и т.д. Пространство System.Reflection.Emit содержит группу классов, позволяющих создавать такие конструкции. Заметим, что код на MSIL может находиться только в теле конструкторов и методов, создаваемого класса. Для построения конструкторов используется класс ConstructorBuilder, а для построения методов - MethodBuilder. Каждый из этих классов имеет метод GetILGenerator. Этот метод возвращает экземпляр ILGenerator для соответствующего метода или конструктора. Получив экземпляр класса ILGenerator, можно создать код на промежуточном языке (Листинг 2). Сказанное выше относится к создаваемым конструкторам и методам. А вот для свойств (PropertyBuilder) и событий (EventBuilder) код на промежуточном языке непосредственно не создается. К свойствам и событиям необходимо присоединять методы, и в этих методах код на MSIL может быть сгенерирован обычным образом, то есть с помощью ILGenerator. Эти методы и будут выполняться при возникновении события или обращению к свойству. Замечу также, что для конструкторов по умолчанию код создается самой системой выполнения Microsoft .NET, и подключить к ним (к создаваемым конструкторам по умолчанию) ILGenerator невозможно. После создания классов, методов и других элементов программы надо указать, как должен использоваться полученный в результате код. В настоящее время предусмотрено три варианта - консольное приложение, DLL или Windows приложение. Метод SetEntryPoint класса AssemblyBuilder позволяет определить, какой из них будет использован. Вернемся, однако, к созданию кода. Как было указано выше, для создания кода надо использовать экземпляры класса ILGenerator. Класс ILGenerator содержит целый ряд методов, которые облегчают работу по созданию кода, в частности, метод Emit использует в качестве первого параметра код команды MSIL, которая и будет помещена в “двоичный файл”. Таким образом, последовательно выполняя метод Emit, можно наполнять тело метода командами языка MSIL. Как видно из рисунка 1, чтобы создаваемое консольное приложение могло сделать вывод строки, вызывается метод WriteLine с помощью команды call языка MSIL. Однако это можно упростить, если при создании кода использовать метод EmitWriteLine вместо метода Emit.
Этот пример крайне прост, но так ли просто создавать такой код в более сложных случаях? Вероятно, не просто, поэтому этот способ хорош, прежде всего, для создания компиляторов, глубоких системных средств, а не для “повседневного использования”.

3 Компиляция на лету

“К языку, как и к другим вещам, значение которых казалось непостижимым, таким как дыхание, кровь, пол, молния, с того времени, как у людей появилась способность фиксировать

свои мысли, было всегда суеверное отношение."
Бертран Рассел. Человеческое познание.

Одна из интересных особенностей Microsoft .NET - это внутренняя поддержка языков программирования. Может показаться странным, но это правда. Ваша программа может создать файл, записать туда исходный код, например, на C#, откомпилировать и затем выполнить. Для поддержки компиляции существуют так называемые провайдеры кода. Для каждого языка имеется собственный провайдер, и в настоящее время Microsoft .NET имеет провайдеры кода для трех языков – C#, Visual Basic и JScript. Провайдеры кода выполняют две основные функции – они могут создать генератор и компилятор кода. Генераторы кода - это объекты, которые могут создавать исходный код "на лету". Они часто используются в Microsoft .NET, но как правило, незаметно на заднем плане. Например, при создании клиента для Web сервиса генератор создает класс, описывающий сервис на базе спецификации, написанной на языке WSDL. Функция компиляторов понятна – компилировать исходный код. Заметим, что в новых версиях Microsoft .NET количество поддерживаемых языков может измениться и, причем, не обязательно в большую сторону (об этом подробнее в конце статьи). С другой стороны, язык C# играет совершенно особую роль в Microsoft .NET. Я бы сказал, что C# невозможен без Microsoft .NET так же как и Microsoft .NET невозможна без C#. Microsoft .NET может обойтись без C++ и Visual Basic, но не обойдется без C#. Это особое отношение к C# выражается и в том, что в Microsoft .NET существует специальный класс Compiler (пространство имен Microsoft.CSharp) с методом Compile, но аналогичного класса нет ни у Visual Basic, ни у JScript.
Таким образом, существование в Microsoft .NET провайдеров кода позволяет без особых трудностей создать исходный код программы на лету и тут же откомпилировать. В тоже время, вполне возможно, что повозившись некоторое время с провайдерами кода Вы решите, что никакой необходимости в Visual Studio нет и Вы возьметесь за создание своей собственной оболочки, позволяющей Вам создавать ПО на C#. Ну что же, это вполне реально, но для этого Вам, возможно, придется изучить ряд дополнительных классов и интерфейсов, например, необходимых для создания дизайнера форм. С другой стороны, Вы можете разработать собственный язык и создать для него провайдер кода. Поскольку компилятор преобразует исходный код в MSIL код, то так как MSIL объектно-ориентирован, то было бы лучше, если исходный язык тоже будет объектно-ориентирован. В принципе он может быть и не объектно-ориентирован, но это значит получать недостатки от объектно-ориентированного программирования вместо преимуществ.

4 Дом для кода – CodeDOM

Итак, мы рассмотрели как можно на лету создавать код на промежуточном языке, затем – как это можно сделать на языке высокого уровня (а, создав, скомпилировать этот код на лету), теперь давайте поднимемся еще на один уровень вверх.
Прежде всего надо учесть, что парадигма объектно-ориентированного программирования вовсе не подразумевает ориентацию на какой-то конкретный язык программирования. Так в ООП "объекты являются главными элементами абстракции"([4]), а вовсе не какими-то языковыми конструкциями, якобы выражающими сущность парадигмы. Разумеется, что в зависимости от фазы жизненного цикла программного продукта сущность понятия "объект" может варьироваться (от единиц функциональности на этапе анализа задачи до единиц реализации на этапе программирования). Эти тонкости термина "объект" и выражают то, что речь идет о различных уровнях абстракции. Именно с этим и связана известная идея о том, что объектно-ориентированная разработка программ - это вовсе не умение программировать на C++, Object Pascal или SmallTalk, а, в первую очередь, это умение мыслить в терминах объектов. Поднимаясь на одну ступень абстракции (от программирования на MSIL, C# и т.д.), мы придем к выводу, что и постановку задачи, и ее решение можно выполнить почти до деталей, не привязываясь к конструкциям конкретного языка программирования. В принципе и программист, и постановщик задачи зачастую мыслят достаточно декларативно, что может выглядеть примерно так – "В моей задаче есть три объекта – велосипед, дорога и велосипедист. Их и надо реализовать в программе. Пусть имеется три типа объектов, соответствующие перечисленным выше. Объекты имеют свойства – например, велосипедист может быть мужчиной или женщиной, взрослым или ребенком и т.д. и кроме того, каждый объект может выполнять какие-то действия. Тот же велосипедист может крутить педали, показывать рукой поворот или падать с велосипеда. Как только мне удастся четко описать цели, структуры объектов, их действия и взаимные влияния, я практически решу задачу". Все это, конечно, достаточно упрощенно, но, надеюсь, наглядно демонстрирует насколько "мыслить в объектах" отличается от "кодировать в объектах".
Если представить себе какой-то "планетоход" где-то на краю солнечной системы, выполняющий в автономном режиме свою ответственную миссию, то кажется очевидным, что он может рассчитывать на успех, только если сможет сформировать собственную модель среды на основании внешних данных (температура, пейзаж, атмосфера и т.д.). Успех миссии (в виду отсутствия рядом человека) будет зависеть от правильности модели, от умения программы планетохода "мыслить" в объектах. Кто-то, возможно,

скажет – это не сейчас, это в будущем, а как же миссии к Сатурну, посадки и забор грунта с космических тел? А разве мы хотим постоянно рисковать жизнями шахтеров – должно же наступить время, когда в забой будут отправляться автоматы. И уже сейчас (а в будущем и подавно) существует целый класс программ, который должен формировать модели, объекты, описывать уровни взаимодействия объектов и, в конечном счете, из созданной модели на лету формировать программу, которая будет решать поставленную задачу. И вот для решения такого рода задач (и не только) и был создан CodeDOM, как составляющая часть Microsoft .NET Framework.

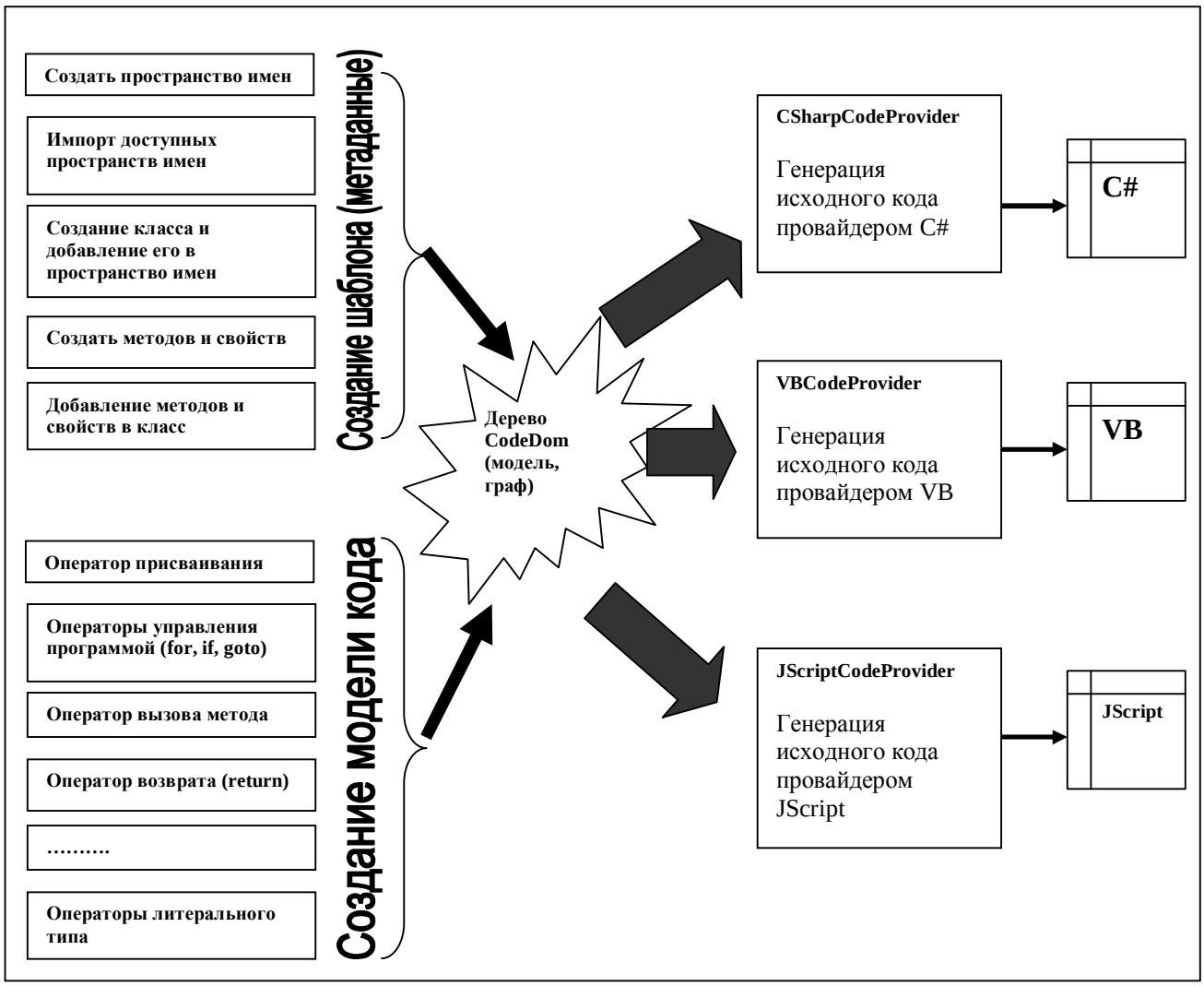

Рисунок 2. Технология динамического создания программы с помощью CodeDOM.

В практическом смысле для создания “программ” (в семантике CodeDOM используются термины: CodeDOM модель, CodeDOM граф), ориентированных на объекты (рисунок 2), используются ряд методов из пространства имен System.CodeDOM. Считается, что все языки программирования, которые могут быть разработаны для Microsoft .NET, должны обладать рядом свойств – например, технология работы с пространствами имен, поддержка объектно-ориентированного программирования, наличие в этих языках операторов “стандартного типа” – присваивание, управление программой, вызов методов и т.д. Вообще говоря, это означает, что язык CodeDOM моделей (язык, ориентированный на объекты) также должен обладать теми же свойствами. Это условие с одной стороны ограничивает возможности CodeDOM, а с другой – опирается на то, что мощь языков в Microsoft .NET прежде всего определяется мощностью самой библиотеки классов Microsoft .NET и в меньшей степени выразительной мощностью языков программирования.

Фактически, с помощью классов пространства имен System.CodeDOM можно осуществить описание системы объектов, определив их методы, свойства, особенности взаимодействия. С точки зрения объектно-ориентированного программирования это означает представление исходной задачи в некотором абстрактном виде, а если в это представление включены и уровни реализации для методов и свойств, то это также и решение задачи. В результате вызовов методов классов пространства имен System.CodeDOM и

создается CodeDOM модель (см. в центре рисунке 2), которая является объектным представлением программы не привязанным ни к какому объектно-ориентированному языку. Идея, однако заключается в том, что существуют специальные провайдеры языков, которые позволяют генерировать исходный код из CodeDOM модели на соответствующем языке программирования (а дальше можно компилировать на лету и запускать на выполнение).

Самое время сказать о недостатках CodeDOM. И самый главный из них это то, что CodeDOM это ограничитель для разработчиков языков. Если кто-то создает новый язык и хочет, чтобы этот язык был равноправным языком (хоть в какой-то степени) по отношении к C#, VB .NET, то это значит, что во-первых, надо сделать реализацию той части CodeDOM, которая ответственна за генерацию кода на конкретном языке из CodeDOM модели, а во-вторых согласится со всеми текущими ограничениями CodeDOM. Суть ограничений я довольно абстрактно представил на рисунке 3. Сам CodeDOM представлен в виде серого круга (на самом деле это не CodeDOM, а возможности CodeDOM по представлению решения различных задач). Различные языки представлены на рисунке в виде кругов и эллипсов. Эти языки могут как-то отображаться в возможности CodeDOM(результат такого отображения – круги и эллипсы, нарисованные штрих-пунктиром). Если язык программирования совмести с CodeDOM, то он должен оказаться полностью внутри серого круга. Если язык содержит какие-то возможности, которые не представимы с помощью CodeDOM, то “отображение” языка не окажется полностью внутри серого круга, то есть не все возможности такого языка поддерживает CodeDOM.

Положительным моментом в CodeDOM является его поддержка многоязычности. Сама Microsoft, в частности, использовала CodeDOM непосредственно при разработке Visual Studio. Например, если пользователь Visual Studio создает клиента для Web сервиса, то он может это делать на различных языках. При этом Visual Studio на основании описания сервиса на языке WSDL создает специальный прокси класс для обращения к Web сервису (вообще говоря, для обеспечения совместимости технологии Web сервисов и объектно-ориентированного программирования). Так что, разработчики из Microsoft должны были описывать создание такого класса для разных сред (C#, VB .NET и т.д.) по-разному? Как раз – наоборот. В Visual Studio такого типа прокси классы вначале создаются в виде CodeDOM модели, а потом из этой модели генерируется код на том языке, на котором ведется разработка. Кроме Web сервисов, CodeDOM используется при генерации кода в ASP.NET, различных визардах (мастерах), дизайнерах и т.д. Ну и конечно для динамического создания кода на лету.

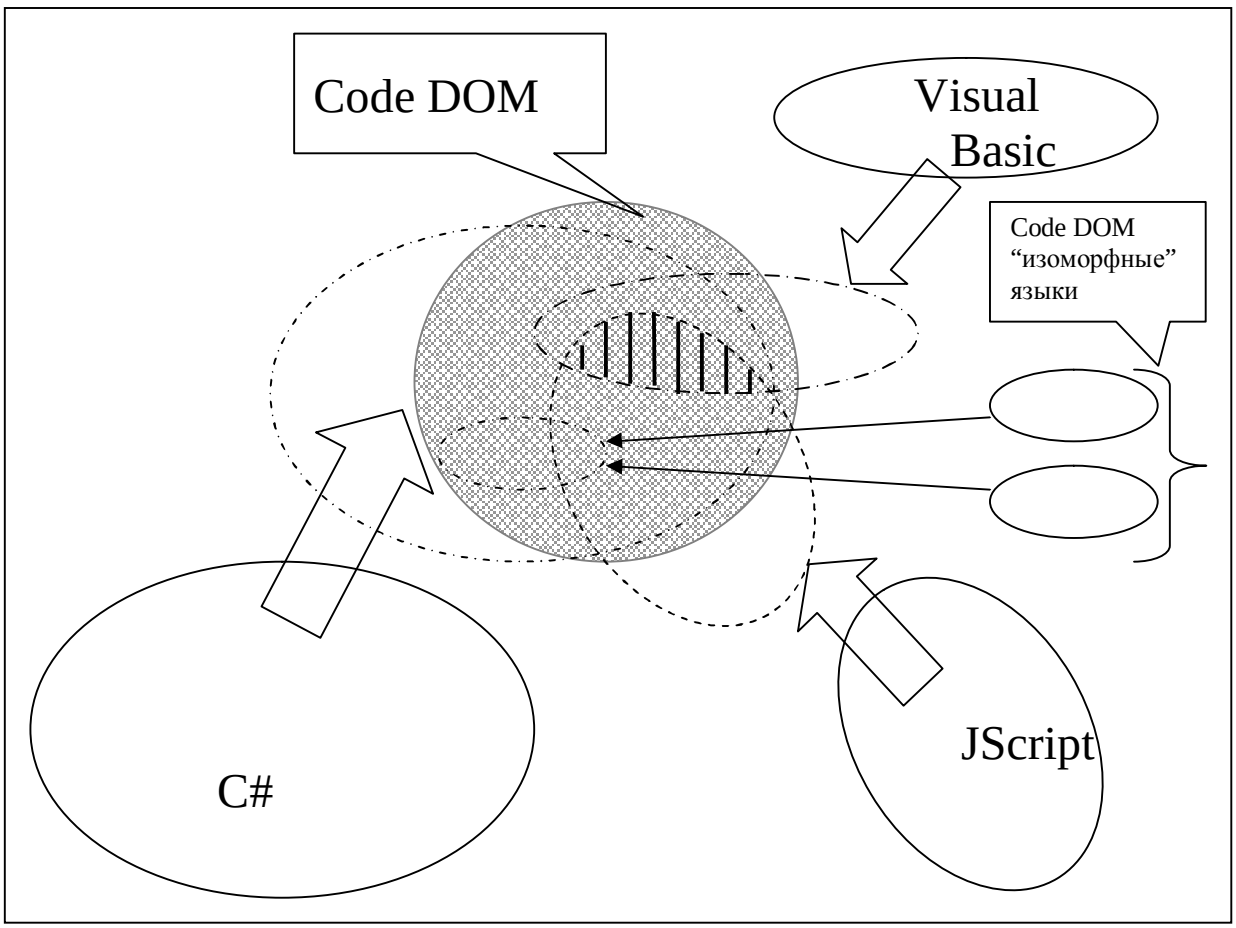

Рисунок 3. “Реальное” отображение языков программирования в CodeDOM представления языков.

5 Создал, построил, запустил

Из всего вышесказанного следует, что для программирования в стиле “Microsoft .NET” быстрое динамическое создание кода является чем-то естественным изначально заложенным в технологию разработки программ. Это не высовывается на первый план, но это и не прячется и при желании всегда может быть использовано в обычных программах. Созданный на лету код может быть откомпилирован и тут же вызван как часть текущего приложения или как независимое приложение. Уверен, что эти возможности не случайны и многое в ближайшем будущем будет связано с ними.

6 Заключение

“Здравый смысл – это толща предрассудков,
успевших отложиться в нашем сознании к
восемнадцати годам”.
Альберт Эйнштейн.

Как-то в Интернете я прочитал, что Европейский суд наложил штрафы на Microsoft по причине того, что Media Player был включен в состав операционной системы. По мнению суда, этот продукт не должен входить в состав ОС Windows XP. Честно говоря, мне хочется верить, что эта статья была некорректным переводом с иностранного языка. В противном случае, не совсем понятно как Европейский суд мог принять такое решение. Если речь идет о конкуренции и монополии, то решение представляется вполне возможным, но неужели суд мог принять это решение на основании каких-то программистских соображений. В конце концов, может возникнуть аналогичный вопрос о возможности включения в операционную систему средств по поддержке алгоритмических языков (о чем собственно и рассказывает эта статья). Ведь эти средства – это часть Microsoft .NET, а Microsoft .NET – это неотъемлемая часть новой операционной системы фирмы Microsoft. Вопрос с конкуренцией и монополией понятен и это действительно вопрос судебных расследований, но хочется надеяться, что высокий суд будет меньше ссылаться на программистские тонкости (как например, что может, а что не может входить в состав ОС).
А разве мы работали бы сейчас на персональных компьютерах, если бы когда-то какой-то суд запретил бы первые компьютеры Apple на основании того, что в их операционную систему входят возможности расширенной поддержки мультимедиа по сравнению с большими ЭВМ?
(Исходный код к статье написан на Visual Studio 2005 Beta и может быть загружен с сайта журнала).

7 Литература

1. UML в кратком изложении. Приложение стандартного языка объектного моделирования. М.Фаулер, К. Скотт. Издательство “Мир”, 1999.
2. Применение UML и шаблонов проектирования. Крэг Ларман. Издательский дом “Вильямс”, 2002.
3. Взаимодействующие последовательные процессы. Ч.Хоар. Издательство “Мир”, 1989.
4. Принципы объектно-ориентированной разработки программ. Антон Эпиенс. Издательский дом “Вильямс”, 2002.

Афанасьев Владимир
avv@systema.kiev.ua

Листинг 1.

```
using System;

namespace HelloFromMSIL
{

	class Hello
	{

		[STAThread]
		static void Main(string[] args)
		{
			System.Console.WriteLine("Hello from MSIL");
		}
	}
}
```

Листинг 2.

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Security.Policy;


namespace MSILCodeCreator
{
	class CodeCreator
	{
		[STAThread]
		static void Main(string[] args)
		{
			if (args.Length == 0 || Int32.Parse(args[0]) > 3)
			{
				Console.WriteLine("Используйте вызов с аргументом:");
				Console.WriteLine("1 - построить и запустить.");
				Console.WriteLine("2 - построить и сохранить.");
				Console.WriteLine("3 - построить, запустить и
сохранить.");
				return;
			}

			AssemblyName assemblyName = new AssemblyName();
			assemblyName.Name="MSILCode";

			//создание динамической сборки
			AssemblyBuilder
AssemBuild=AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,

	System.Reflection.Emit.AssemblyBuilderAccess.RunAndSave);

			//модуль для сборки
			ModuleBuilder
module=AssemBuild.DefineDynamicModule("ILSample","MSILCode.exe");

			TypeBuilder NewTB=module.DefineType("Hello");

			Type[] param={typeof(System.String[])};

			//создать метод Main
```

```
                  MethodBuilder meth=NewTB.DefineMethod("Main",
MethodAttributes.Public |
                        MethodAttributes.Static, null, param);

                  //тело метода
                  ILGenerator ILGeng=meth.GetILGenerator();
                  ILGeng.EmitWriteLine("Hello from MSIL");
                  ILGeng.Emit(OpCodes.Ret);

                  Type HelloClass;
                  if(Int32.Parse(args[0]) == 1 || Int32.Parse(args[0]) == 3)
                  {
                        HelloClass=NewTB.CreateType(); //только для запуска
метода
                        HelloClass.GetMethod("Main").Invoke(null,new
object[]{new string[]{}});
                  }


                  if(Int32.Parse(args[0]) == 2 || Int32.Parse(args[0]) == 3)
                  {

      AssemBuild.SetEntryPoint(meth,PEFileKinds.ConsoleApplication);
                        AssemBuild.Save("MSILCode.exe"); //сохранить
приложение
                  }

            }
      }
}
```