Рассказ о поездке на конференцию .NEXT

Денис Гладкий
denis.gladkiy@yahoo.com, d.gladkiy@ftc.ru

Управление предоплаченных карт, ГК ЦФТ

2 января 2016 г.

1 Введение

Отчитываюсь о посещении конференции по .NET-технологиям .NEXT (`http://dotnext.ru/`). Действие проходило 4 апреля 2014 года в городе Санкт-Петербурге. Список докладов ещё до начала вызывал у меня нездоровое возбуждение. В итоге, так и вышло — .NEXT оказалась лучшей конференцией, которую я посетил за последние три года. Пообщался с матёрыми .NET-гуру, впитал новые знания по программированию, получил заряд мотивации. Далее постараюсь передать то, что узнал на различных докладах.

2 Дмитрий Нестерук. Евангелистическое тро-ло-ло за эффективность программирования.

Тут повествование будет идти, в основном, от первого лица, но будут иметься в виду слова докладчика с наслоением моей интерпретации. Как и все выступления евангелистов, из речи господина было сложно выцепить какую-то полезную информацию. Вроде, и говорит всё правильно, но ничего нового. Кое-что всё-таки удалось осознать. Докладчик представил набор параметров, характеризующих «эффективное программирование»:

1. качество кода:
    - корректность;
    - производительность;
    - расширяемость (насколько легко сопровождать).
2. скорость написания кода.

Обычно, 1 и 2 коррелируют: качественный код пишется не быстро, а наспех запрограммированное в последствие приходится оптимизировать, исправлять дефекты и т.п. Так же, несмотря на все продвижения в области переработки кода (англ. refactoring), всё равно бывают клинические случаи (а.к.а «switch» на

тысячу строк кода), в которых никакая переработка не поможет — код нужно выкидывать и писать заново. Задача среды разработки (далее IDE), а для её разработчиков — цель, уменьшать вышеупомянутую корреляцию. Рассмотрим инструменты, доступные программисту в комплекте Visual Studio + ReSharper, которые помогают улучшать тот или иной аспект эффективности программирования.

2.1 Корректность:

- компилятор — формальная проверка типов;
- инспекции — эвристические проверки кода;
- подсветка ошибок и ляпов в реальном времени — минимизация времени между внесением дефекта и его обнаружением;
- «quick fix» — быстрая правка дефекта по месту, без откладывания в «долгий ящик»;
- «context actions» — переработка кода по месту, без откладывания в «долгий ящик».

2.2 Производительность:

- встроенные в IDE или поставляемые в виде плагинов средства профилирования;
- подсветка подозрительных конструкций в коде.

Кстати, утверждается, что ReSharper 8 умеет подсвечивать неочевидные места создания объектов (например цикл foreach), boxing, unboxing.

Ещё забавную фразу докладчик сказал: «автомагически» :) Надо взять на вооружение.

2.3 Поддержка кода:

- парсинг комментариев в формате XML-doc (указание недостающих или лишних аргументов функции и пр.);
- поддержка соглашений об именовании.

Автор так же высказал мысль, которую я уже слышал от местного (не только .NET) гуру Никиты Каменского, что UML, как средство разработки и документации провалился (это ещё не мысль :), при этом функции, которые, по идее, должны были на себя брать среды UML-проектирования, появились и очень естественно зажили в IDE. Например, кодогенерация: написали набор полей, нажали Alt + Ins — сгенерирован набор properties. При этом список полей класса доступен в соответствующей вкладке, а анализатор кода способен показать любые зависимости между классами.

2.4 Расширяемость (она же гибкость):

Какое-то ололо-тро-ло-ло в духе «следи за собой, будь осторожен» и «повышения бдительности личного состава» :) Чего-то полезного извлечь не смог.

2.5 Скорость написания:

- live templates — генерация куска кода по набору ключевых символов;
- кодогенерация через Alt + Ins — конструкторы, специальные методы (ToString, Equals+HashCode) и т.д.;
- правильное представление предметной области.

Последний пункт требует пояснения на примерах. Рассмотрим документ Word со сложными формулами (3-х этажный интеграл в пару строк). Нужно перенести формулу в код. Если просто переписывать: *a*) долго; *b*) можно понаделать ошибок; *c*) работа машинистки, а не программиста. Решение: написание транслятора из формулы в формате Word в код на C#. Под форматом Word понимается не именно файл doc или docx (хотя и он может), а более простая схема данных (есть Office COM API, можно экспортировать формулу в общеизвестный формат и пр).

Следующий пример: нужно написать код по выдаче некоего XML. Есть API, к примеру, LINQ2XML. Удобно написать именно XML, а потом как-то его превратить в код API. Решение: плагин к ReSharper (вот тут я не понял, он уже есть или утверждалось, что его создание просто), который умеет генерировать код по данным из предметной области. Например, пишем:

```
<html><head></head><body></body></html>
```

Далее нажимаем ⟺ и строка автоматом транслируется в что-нибудь вроде:

```
var body = new XElement("body");
var head = new XElement("head");
var node = new XElement("html", head, body);
```

Или же написание дат. Пишем код:

```
var date = 01.01.2014;
```

Нажимаем ⟺ и получаем:

```
var date = new DateTime(2014, 1, 1);
```

Итого, что можно вынести из доклада:

- современные IDE сильно далеко ушли от «блокнотов» и требуют изучения;
- применение IDE снижает корреляцию между скоростью и качеством кода;
- важную роль в эффективности играет правильное представление предметной области;
- современные IDE позволяют поддержать это представление через систему плагинов (люди, не бойтесь делать плагнины к IDE).

3 Станислав Сидристый. Unmanaged .NET.

В докладе было показано, как на почти чистом C# создать объект ссылочного типа на стеке без вызова его конструктора, как подменять записи в таблице виртуальных функций класса, как обеспечить работу объектов из разных процессов/app domain без механизмов reflection и сериализации.

Конкретный код привести не смогу — переписать его с экрана не представлялось возможным. Скажу лишь, что все означенное выше действительно возможно. Ограничения: full trust режим работы кода, кое-какие действия автору пришлось писать не на C#, а на IL, некоторые решения непереносимы между платформами различной битности.

Продемонстрированные трюки были именно трюками и далеки от боевого применения. Смысл их в том, что человек, реализовавший их начинает очень глубоко понимать платформу, принципы работы сборщика мусора (далее GC), без которых, как показал другой доклад (оптимизация .NET-приложений, секция 5), невозможно понять причины некоторых проблем в производительности кода. Так же автор обозначил некоторые направления всё-таки реального использования трюков:

- ручное управление памятью через пулы объектов (когда нужно создавать огромное количество объектов);
- оптимизация взаимодействия между кодом в различных app domain или процессами (не нужно сериализовывать/десериализщовывать: создаём через WinAPI разделяемую память (shared memory), далее создаём в ней .NET-объекты, и прописываем им указатели в таблицах виртуальных функций на методы реальных объектов и соответствующего app domain или процесса);
- оптимизация Hibernate-подобных [1] библиотек: вместо механизма reflection и генерации byte-кода можно получить прямой доступ к таблицам виртуальных функций.

Необходимым для выполнения трюков было понимание устройства памяти CLR. Она делится на области:

- stack — локальные переменные функций;
- small objects heap (SOH);
- large objects heap (LOH) — для объектов >= 85000 байт [2];
- code heap — для размещения результатов работы JIT;
- high frequency heap — различные внутренние структуры данных, к примеру, таблицы виртуальных функций.

[1] `http://hibernate.org/`

[2] Подтверждение в официальной документации найти не вышло, но есть некий блог на MSDN `http://blogs.msdn.com/b/tess/archive/2008/04/17/how-does-the-gc-work-and-what-are-the-sizes-of-the-different-generations.aspx`

Для всех этих областей сборщик мусора (GC) работает по-разному (причём, в зависимости от версии CLR). Так, например, в .NET 3.5 LOH не сжимается и не дефрагментируется, а SOH делится на три поколения (gen0, gen1, gen2) объектов, внутри которых GC так же работает по различным схемам. Объекты отличных типов так же хранятся по-разному. Принципиально отличается схема хранения массивов, строк и обычных объектов, так как первые два имеют переменный размер.

Был продемонстрирован алгоритм перечисления всех объектов в памяти:

- через WinAPI (доступен через механизм P/Invoke) получить все страницы памяти нашего процесса;
- интерпретировать все четвёрки байтов, пока не найдено что-то, что является .NET объектом (это можно сделать по адресам таблицы виртуальных функций, размещение которых известно — high frequency heap);
- найти размер объекта (расположен в описании класса рядом с таблицей виртуальных функций);
- так как SOH дефрагментирован, то следующий объект лежит по фиксированному смещению за текущим.

Главная проблема многих трюков докладчика — некоторым операциям необходима «заморозка» GC. Доклад очень вдохновил меня в предстоящем учебном году выдать студентам набор необязательных к решению задач повышенной сложности.

4 Сергей Шкредов. Управление зависимостями а.к.а «о сборке кучи продуктов из общей кодобазы».

Автор ранее уже выступал на конференции CodeFest 2014 — `http://2014.codefest.ru/lecture/842`. В докладе рассказывалось, как .NET-команда JetBrains организовала сборку линейки продуктов из единой кодобазы. Обсуждалось то, с чем постоянно сталкиваются устоявшиеся продуктовые компании/команды. Предметно: 40 (порядок точно правильный, а вот +−10 мог напутать, точно не записал) .NET программистов, плоская иерархия управления (утверждалось, что на всю эту толпу менеджеров нет вообще), четыре доступных клиентам продукта (ReSharper, dotPeek, dotTrace, dotCover), неизвестное количество продуктов в разработке.

.NET из коробки предоставляет механизм сборок. Он, хоть и сильно дальше ушёл от Java-овских jar-файлов (которые, кстати, даже не часть языка, а сбоку прикрученная функциональность среды выполнения), но всё равно не очень удобен в описанном выше случае:

- нет явного описания зависимостей;
- видимость «internal» не иерархична: есть сборка, есть под-сборки, из объемлющей сборки просто так нельзя получить доступ к под-сборкам, хотя они являются частями одного проекта;

- необходимость управлять зависимостями не только на уровне кода (где, есть уже отлаженные механизмы: inversion of control, различные DI-каркасы и пр.) но и на уровне сборок;
- фиксация независимости (выделение новой сборки) приводит к созданию большого графа со всеми проблемами, имеющимися в больших иерархиях классов.

Тут добавлю от себя примечание. В больших проектах возникает необходимость управлять кодом на уровне не классов, а групп классов — модулей. В то время, как для управления зависимостями между классами у программиста имеется множество инструментов на уровне языка (наследование, настройка инкапсуляции и пр.), для управления модулями данные механизмы отсутствуют.

Система пакетов NuGet так же имеет проблемы (как, кстати, и Maven и Gradle):

- никак не связана с языком — внешний по отношению к коду инструмент;
- сложна, как в освоении, так и в сопровождении.

Итог: парни написали свою систему построения приложения из компонентов (видимо, что-то похожее на Eclipse или OpenSim [3] на котором СофтЛаб [4] делает системы виртуальной реальности). Тезисно:

- принадлежность к компоненту в коде описывается с помощью custom attributes;
- приложение как таковое — система по управлению плагинами:
  - точка входа — это не классическое в DI место, которое «знает обо всём», а лишь код по загрузке и анализу модулей;
  - анализ модулей идёт, как по custom attributes, так и по интерфейсам типов данных и по naming convention;
  - дополнительно написан плагин (я не понял к ReSharper или VS) для анализа кода по вышеозначенным атрибутам;
  - дополнительно написана система сборки продуктов, которая включает в дистрибутив куски только того кода, который реально используется в продукте.

Господа обещают выдать систему для всеобщего использования, когда будет «достаточное количество заинтересованных в ней клиентов».

5 Кирилл Скрыган. Оптимизация .NET приложений.

Рассказывались случаи из опыта разработки продукта ReSharper. Цикл разработки у парней такой: программируют, в конце один месяц (в частности, по

[3] http://opensimulator.org/wiki/Main_Page
[4] http://www.softlab-nsk.com/

отзывам от пользователей) тратится на оптимизацию, потом релиз. Основных мест проседания производительности два: плохая асимптотика в алгоритмах ($O(n^k)$, где k>1), «memory traffic» (в простонародье — работа GC). В докладе описывались только случаи, ведущие к увеличению/проблемам с memory traffic, которые разработчикам приходилось исправлять. Далее я приведу случаи которыми делился докладчик:

5.1

Лямбда с захватом локальных переменных — тут понятно, что создаётся объект в поля которого прописывается локальный контекст.

5.2

Интересное! Есть код:

```
void Bar()
{
    int[] a = new int[10000];
    int b = 10;
    fun(() => a[0], () => b + 1);
}
```

Парни, как умные Маши, даже обнуляли ссылку на уже ненужный делегат внутри Fun. Но не тут-то было. ILDASM показал, что компилятор на обе лямбды создавал один объект для поддержки замыкания!

5.3

Переменное число аргументов в коде выглядит безобидно, но создаёт ненужный объект даже в случае вызова функции без аргументов. Боролись просто — создавали перегруженные функции под каждое кол-во аргументов.

```
void Bar(params string list)
{
}
void BarFast()
{
}
void BarFast(string argument)
{
}
void Zig()
{
    Bar(); //new object allocation!
    Bar(""); //new object allocation!
    BarFast(); //ok, no objects allocation
    BarFast(""); //ok, no objects allocation
}
```

5.4

Оператор yield return, понятно, тоже создаёт лишний объект — state machine для поддержки continuation.

5.5

В коде ReSharper'а часто применяется шаблон приспособленец (flyweight[5]). К примеру, рукописные реализации «пустых» IEnumerable со статическим пустым итератором (стоит заметить, что парни пишут на .NET 3.5 из-за необходимости поддерживать старые версии Visual Studio).

5.6

Оператор foreach имеет утиную типизацию [6], а использование с ним типов через интерфейс IEnumerable ведёт к созданию в куче объекта итератора. Поэтому используются везде, где можно, конкретные типы коллекций (не IList, а List, к примеру), так как все контейнеры стандартной библиотеки имеют struct-based реализации итераторов, что не приводит к созданию нового объекта в куче.

5.7

А треперь совсем нетривиальное. SOH в .NET разбит на 3 поколения. Работают две вычислительные нити. Одна заканчивает работу. GC собирает все объекты с её стека, а объекты со стека второй (она ещё работает) уходят в gen1, не смотря на то, что они временные и дальше своих stack-frame'ов не собираются жить. Кстати, схема возможна и в Java.

5.8

В продолжении темы шаблона приспособленец, у парней написано множество собственных функций для переиспользования строк (substring и пр.). Дело в том, что из-за необходимости interop'а с Win API в .NET строки в памяти — честные null-terminated C-style строки, а не «ropes», как в Java до версии 1.6 включительно.

5.9

Так как с LOH бывают проблемы (не сжимается, не дефрагментируется), то ребяты написали ChunkedList и прочих, как они сами говорят, «велосипедов». Однако, тот же ChunkedList хоть и уменьшает нагрузку на LOH, но при этом увеличивает memory traffic.

5.10

Запросы к HDD-кэшу данных объединяются в большие транзакции (кстати, на HDD в ReSharper вынесены некоторые структуры данных, как раз что б ни нагружать сборщик мусора).

[5] http://en.wikipedia.org/wiki/Flyweight_pattern

[6] http://ericlippert.com/2014/01/02/what-is-duck-typing/

5.11

Для работы UI-нити с фоновыми создана InteruptableActivity — это конструкция, которая захватывает lock но периодически проверяет, если его кто-то попытался захватить (UI-нить), то она тут же останавливается и отпускает lock.

6 Владислав Чистяков. Nitra.

Для читателей, которые знают, что такое MPS (`http://www.jetbrains.com/mps/`), всё просто: Nitra — это MPS, работающий на платформе .NET и не имеющий его (MPS) странностей (языки как обычно описываются в текстовых файлах, никаких вам concepts, projection editors и прочего футуристического). Для остальных поясню подробнее. Nitra — это каркас для создания любых языков. Основной вариант использования для пользователей: иметь возможность в Visual Studio бесшовно создавать свои расширения к C# и различные domain specifiec languages. Основной вариант использования для самих JetBrains, как я понимаю, — унифицировать в компании создание парсеров/анализаторов языков программирования. На данный момент у парней готово: простая IDE с подсветкой кода, folding/unfolding кода, восстановление парсера от ошибок, квазицитирование (возможность задавать язык не только в виде ast, но и в виде готовых кусков/шаблонов кода). Интересно у них сделана система восстановления от ошибок: парсер начинает выкидывать правила вывода из грамматики и получает множество грамматик, в которых данный текст допустим. Далее всякими эвристиками множество уменьшается и по нему строится для пользователя набор предложений, как исправить ошибку, ну и, собственно, дальше происходит переход к следующей единице разбора (statement). Я раньше не знал, как такое делается. Нам в университете не рассказывали, как восстанавливаться от ошибок. Проект выглядит очень круто, и если у парней всё получится, то у MPS будет серьёзный конкурент.

7 Евгений Кошкин. О том, как живёт команда разработки TeamCity.

Наиболее познавательное и поучительное в докладе — это доведение командой TeamCity принципа «eating own dogfood» до грани абсурда :) TeamCity из коробки поддерживает работу со множеством систем контроля версий (далее VCS): Git, Hg, TFS, SVN, Perforce. Парни взяли и попилили исходный код на куски, разместив их во всех, поддерживаемых TeamCity VCS. Перед таким стремлением сделать классный продукт я снимаю шляпу. Далее тезисно:

- не используют ветки в коде (код раскидан по разным VCS);
- используют схему «branching by abstraction» (a.k.a. boolean флажки в коде) для ненадёжного функционала:
  - если у клиентов после выката что-то не работает, то поддержка сообщает как выключить новый функционал, приведший к сбою;

  - набор флажков не публичен, но конкретные выдаются по факту проблем;
  - со временем флажки убирают из кода.

- всего два тестировщика на проект: очень много автоматизированного тестирования;
- внутренняя установка TeamCity: около 1000 комитов в день, около 1500 сборок в день, среднее время ожидания сборки в очереди — час, ферма из 140 агентов сборки.

Хоть TeamCity и программируется на языке Java, докладчик рассказывал разные случаи из жизни .NET-команды JetBrains, так как конференция по .NET-технологиям всё-таки:

- очень долго не использовали ветки, так как сидели на SVN;
- используют Hg:
  - на момент выбора клиент Git под Windows оставлял желать лучшего;
  - набор команд очень похож на SVN.
- периодически всплывают проблемы «Huge Merge» (долгая разработка в отдельной ветке, а затем слияние с основной);
- для разделения кода между продуктами используют ветки, но думают перейти на бинарную поставку кода в виде сборок.

Последний пункт стоит осветить подробнее. У них есть проект с библиотечным кодом .NET Platform (далее просто «платформа»). Её используют все .NET-продукты. Репозиторий данной библиотеки имеет ветки: а) стабильную б) разрабатываемую в) по 2 ветки на продукт. Периодически ветки продуктов сливаются в основную ветку, принося туда новый функционал. Ветки на продукты нужны так как платформа постоянно меняется и текущий код продукта может быть несовместим с нововведениями в платформе.

8 Станислав Сидристый. Xamarin.

Рассказывалось про Xamarin, платформу разработки на C# под операционные системы iOS и Android. Тезисно:

- 1000$ в год (если необходима поддержка написания кода в Visual Studio).
- продукт стабилен и имеет хорошую поддержку (поддержка iOS 7.1 была заявлена через день после выхода оной);
- отражение (binding) в C# API конкретной платформы генерируется автоматически;
- решаются различные проблемы командного характера:

  - C# программистов найти легче, чем iOS;
  - взаимозаменяемость людей (программист Android-части может легко заменить разработчика iOS-части);
  - общая бизнес-логика (потенциально разделяемая с проектами для Windows Phone).

Недостатки:

- примерно 3 дополнительных мегабайта в дистрибутив приложения на среду выполнения;
- не всегда корректно строится отображение API;
- при падении приложения (не самого Xamarin) иногда бывает «потеря» call stack;
- иногда отладчик отцепляется от процесса.

В целом, докладчик был очень позитивно настроен относительно Xamarin.

9 Антон Оникийчук. Применение TDD в WPF-программах.

Докладчик в живую продемонстрировал применение TDD (test-driven development, `http://en.wikipedia.org/wiki/Test-driven_development`) при разработке MVVM-приложения (MVVM — основной шаблон проектирования в разработке приложений на каркасе WPF). Утверждается (неверно, на мой взгляд), что для WPF нет систем автоматического взаимодействия с UI. Поэтому предлагается писать тесты на View Model:

1. создать прототип UI на XAML;
2. создать View Model;
3. написать тесты на варианты использования для View Model, так как нет возможности тестировать UI напрямую (тут ещё раз скажу, что автор явно «сидит» на .NET 3.5 — `http://msdn.microsoft.com/en-us/library/ms753107.aspx`);
4. далее — классический цикл TDD: «red — green — refactoring»
   (a) написать тесты (они не проходят);
   (b) писать код, пока тесты не будут проходить;
   (c) провести переработку кода.

Очень живой доклад получился, так как Антон вывел на проектор Visual Studio и почти всё отведённое время демонстрировал осуществление одной итерации цикла.