Vagif Abilov

Типизированный или динамический API? Дайте два!

Коротко о себе

- Работаю в норвежской компании Miles
- Электронная почта: vagif.abilov@gmail.com
- Twitter: @ooobject
- GitHub: object
- Блог: http://vagifabilov.wordpress.com/
- Статьи на CodeProject
- Поддерживаю библиотеку Simple.OData.Client, участвую в других опенсорсных проектах

- Исходный код для этой презентации можно посмотреть здесь: https://github.com/object/HybridSqlCommandBuilder

Разработка современных API

Когда мы были молодыми,
трава была зеленее,
записи баз данных извлекались
посредством сохраненных процедур и
каждому элементу данных
обязательно назначался тип

- И посмотри, что ты сделал с мамой! Она состарилась, сопровождая твои прокси-классы!
- Чем тебе не нравятся мои прокси-классы, пацан? Я в жизни написал больше прокси-классов, чем у тебя было свиданий!

Элементы функционального программирования (2007)

- Существенные добавления в язык C# (мотивированные LINQ)
- Лямбда-выражения
- Методы расширения (extension methods)
- Анонимные типы
- Деревья выражений (expression trees)

Элементы динамических языков (2010)

- Ранние версии C#/.NET - статическое связывание (static binding)
- Полиморфизм - разрешение подтипов до времени исполнения программы
- Версия 4 - динамическое связывание
- Отложенное разрешение выбора методов

Компилятор, «Leave us kids alone!»

```
dynamic client = WeirdWildStuffFactory.GiveMeOneOfThose();

client.NowICanPretendAnySillySentenceIsAMethodCall();
client.AndICanSendAnyArguments(1, "2", new Stream[] {});

var result = client.MeaningOfLife * 42 * Guid.NewGuid();

Assert.AreEqual(42, result);
```

Пример: вызов COM-компонент

```
// Без динамического связывания

((Excel.Range)excelApp.Cells[1,1]).Value2 = "Name";
var range2008 = (Excel.Range)excelApp.Cells[1,1];

// С динамическим связыванием

excelApp.Cells[1,1].Value = "Name";
var range2010 = excelApp.Cells[1,1];
```

Пример: доступ к базе данных

```
// статические типы с Entity Framework
public User FindUserByEmail(string email)
{
    return _context
      .Users.Where(x => x.Email == email)
      .FirstOrDefault();
}

// динамические типы с Simple.Data
public User FindUserByEmail(string email)
{
    return Database.Open()
      .Users.FindAllByEmail(email)
      .FirstOrDefault();
}
```

Пример: реализация сценария языка Gherkin

Background: Validate users
Given the following users exist in the database:

Name	Birth date	Height
John	1940-10-09	1.80
Paul	1942-06-18	1.80
George	1943-02-25	1.77
Ringo	1940-07-07	1.68

```
[Given(@"the following users exist in the database:")]
public void GivenTheFollowingUsersExist(Table table)
{
    IEnumerable<dynamic> users = table.CreateDynamicSet();
    db.Users.Insert(users);
}
```

Популярные библиотеки, использующие динамические типы

- SignalR (~2,300,000 скачиваний с NuGet)
- Nancy (~700,000 скачиваний)
- Fasterflect (~230,000 downloads)
- Simple.Data (~170,000 downloads)
- ReflectionMagic (~130,000 downloads)
- ImpromptuInterface (~150,000 downloads)
- EasyHttp (~100,000 downloads)

Нужны ли вам динамические библиотеки?

- Эта презентация не сделает за вас выбор

Нужны ли вам динамические библиотеки?

- Эта презентация не сделает за вас выбор
- Ее цель - обозначить возможные варианты выбора

Нужны ли вам динамические библиотеки?

- Эта презентация не сделает за вас выбор
- Ее цель - обозначить возможные варианты выбора
- Даже больше - возможность поменять свой выбор с минимальными сложностями

Нужны ли вам динамические библиотеки?

- Эта презентация не сделает за вас выбор
- Ее цель - обозначить возможные варианты выбора
- Даже больше - возможность поменять свой выбор с минимальными сложностями
- Как писать библиотеки, поддерживающие типизированный и динамический API с единым интерфейсом

Нужны ли вам динамические библиотеки?

- Эта презентация не сделает за вас выбор
- Ее цель - обозначить возможные варианты выбора
- Даже больше - возможность поменять свой выбор с минимальными сложностями
- Как писать библиотеки, поддерживающие типизированный и динамический API с единым интерфейсом
- Гибридная упаковка гибридных API

Демо

Зачем может пригодиться динамический API?

При всем при этом…

- Использование динамических типов чаще оправдано при доступе к внешним сервисам
- Стыковка статических и динамических типов в C# может сопровождаться непредвиденными проблемами

```
dynamic results = db.Companies
            .FindAllByCountry("Norway")
            .FirstOrDefault();                  // EXCEPTION
```

- Будьте внимательны при использовании динамических объектов вперемежку со статически определенными типами, особенно при работе со списками и массивами

Возвращаясь к доступу к базам данных

```
var result = db.Companies
             .Where(x => x.CompanyName == "DynamicSoft")
             .Select( x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });
```

Должен динамический API выглядеть так?

```
var result = db.Companies
             .Where(x => x.CompanyName == "DynamicSoft")
             .Select( x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });

var result = db.Companies
             .FindByCompanyName("DynamicSoft")
             .SelectCompanyNameAndYearEstablished();
```

Или так?

```
var result = db.Companies
             .Where(x => x.CompanyName == "DynamicSoft")
             .Select( x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });

dynamic x = new DynamicQueryExpression();
var result = db.Companies
             .Where(x.CompanyName == "DynamicSoft")
             .Select(x.CompanyName, x.YearEstablished);
```

Выбор стиля API – сугубо личное дело

- Мой API - моя крепость, любой выбор наименований – дело личного предпочтения

Выбор стиля API – сугубо личное дело

- Мой API - моя крепость, любой выбор наименований – дело личного предпочтения
- Поэтому мы предоставим выбор имен авторам и зададим другой вопрос

Выбор стиля API – сугубо личное дело

- Мой API - моя крепость, любой выбор наименований – дело личного предпочтения
- Поэтому мы предоставим выбор имен авторам и зададим другой вопрос
- Вы хотите опубликовать один API или два API?

Что если мы определим общее ядро API?

```
interface IQueryBuilder
{
    From(...);
    Where(...);
    OrderBy(...);
    OrderByDescending(...);
    Select(...);
}
```

- Этот интерфейс определяет основные операции нашей библиотеки (internal DSL) для типизированных и динамических клиентов
- Типизированные и динамические клиенты отличаются в том, какие параметры они передают методам API

Один API – две парадигмы

```
var result = db.Companies
             .Where(x => x.CompanyName > "D")
             .OrderBy(x => x.CompanyName);

dynamic x = new DynamicQueryExpression();
var result = db.Companies
             .Where(x.CompanyName > "D")
             .OrderBy(x.CompanyName);
```

- Разработка единого API со схожими параметрами методов для разных случаев унифицирует операции API
- Пользователи API могут достаточно безболезненно переходить от одной парадигмы к другой

Каких это требует дополнительных затрат?

- Дополнительный динамический слой не содержит реализации функционала, являясь адаптером преобразования параметров

Каких это требует дополнительных затрат?

- Дополнительный динамический слой не содержит реализации функционала, являясь адаптером преобразования параметров
- Размер кода динамического слоя не зависит от сложности предметной области

Каких это требует дополнительных затрат?

- Дополнительный динамический слой не содержит реализации функционала, являясь адаптером преобразования параметров
- Размер кода динамического слоя не зависит от сложности предметной области
- Код примера для этой презентации:
  - Основаная библиотека: 20 классов, 681 строк кода
  - Библиотека динамических расширений: 3 класса, 76 строк кода

Каких это требует дополнительных затрат?

- Дополнительный динамический слой не содержит реализации функционала, являясь адаптером преобразования параметров
- Размер кода динамического слоя не зависит от сложности предметной области
- Код примера для этой презентации:
  - Основаная библиотека: 20 классов, 681 строк кода
  - Библиотека динамических расширений: 3 класса, 76 строк кода
- Реальный проект (Simple.OData.Client, версия 3):
  - Основная библиотека: 96 классов, 2760 строк кода
  - Библиотека динамических расширений : 6 классов, 59 строк кода

Стратегия гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic

Стратегия гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic
- В основе API – DSL, основанный на LINQ выражениях

Стратегия гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic
- В основе API – DSL, основанный на LINQ выражениях
- Для каждого метода API, содержащего параметр типа LINQ выражение, есть аналогичный метод с типом кастомного выражения - его-то и выбирает компилятор для динамических типов

Стратегия гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic
- В основе API – DSL, основанный на LINQ выражениях
- Для каждого метода API с параметром типа LINQ выражение есть аналогичный метод с типом кастомного выражения
- Динамические расширения реализуют IDynamicMetaObjectProvider

Стратегия гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic
- В основе API – DSL, основанный на LINQ выражениях
- Для каждого метода API с параметром типа LINQ выражение есть аналогичный метод с типом кастомного выражения
- Динамические расширения реализуют IDynamicMetaObjectProvider
- Динамические расширения упакованы в отдельный модуль (assembly), который устанавливается на платформах с поддержкой DLR

Пример

Finder.dll

```
public interface IFinder
{
    Result Find<T>(Expression<Func<T>,bool>> query);
    Result Find(QueryExpression query);
}
```

```
public class QueryExpression
{
    // нет публичного конструктора
}
```

Finder.Dynamic.dll

```
public class DynamicQueryExpression
: QueryExpression, IDynamicMetaObjectProvider
{
    public DynamicQueryExpression() {}
}
```

Код клиента

Клиент со статическими типами

```
var finder = new Finder();
var result = finder.Find<Companies>().Where(x =>
             x.CompanyName.StartsWith("D") &&
             x.YearEstablished > 2000);
```

Клиент с динамическими типами

```
dynamic x = new DynamicQueryExpression();
var finder = new Finder();
var result = finder.Find(x.Companies).Where(
             x.CompanyName.StartsWith("D") &&
             x.YearEstablished > 2000);
```

План на оставшуюся часть доклада

- Применить метод на примере гибридного построителя комманд SQL
- Итеративно усложнять дизайн:
  - нетипизированный (untyped) построитель комманд
  - типизированный с LINQ выражениями
  - динамический
- Расширить реализацию, добавив симулятор процессора команд SQL

Примерный проект

Гибридный построитель команд SQL

Напишем нечто похожее на провайдер LINQ

```
var result = db.Companies
             .Where(x => x.CompanyName.StartsWith("D")
             .OrderBy(x => x.CompanyName)
             .Select( x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });
```

Поддерживаемый список комманд SQL

```
interface ICommandBuilder
{
    From(...)
    Where(...)
    OrderBy(...)
    OrderByDescending(...)
    Select(...)
}
```

Разминочная версия – только строки

```
interface ICommandBuilder
{
    ICommandBuilder From(string tableName);
    ICommandBuilder Where(string condition);
    ICommandBuilder OrderBy(params string[] columns);
    ICommandBuilder OrderByDescending(params string[] columns);
    ICommandBuilder Select(params string[] columns);

    Command Build();
}
```

Пример использования

```
var command = new CommandBuilder()
    .From("Companies")
    .Where("YearEstablished>2000 AND NumberOfEmployees<100")
   .OrderBy("Country")
   .Select("CompanyName", "Country", "City")
   .Build();
```

Поля класса Command

```
public class Command
{
    private string _table;
    private string _where;
    private List<string> _selectColumns;
    private List<KeyValuePair<string, bool>> _orderByColumns;

    ...
}
```

Command.Format()

```
private string Format()
{
    var builder = new StringBuilder();

    builder.AppendFormat("SELECT {0} FROM {1}",
       _selectColumns.Any() ?
       string.Join(",", _selectColumns) : "*", _table);

    if (!string.IsNullOrEmpty(_where))
       builder.AppendFormat(" WHERE {0}", _where);

    if (_orderByColumns.Any()) {
        builder.AppendFormat(" ORDER BY {0}",
           string.Join(",", _orderByColumns.Select(
               x => x.Key + (x.Value ? " DESC" : string.Empty))));
    }

    return builder.ToString();
}
```

Двигаемся к типизированной версии

```
interface ICommandBuilder
{
    ICommandBuilder<T> From<T>();
}

interface ICommandBuilder<T>
{
   ICommandBuilder<T> Where(Expression<Func<T, bool>> expression);
   ICommandBuilder<T> OrderBy(Expression<Func<T, object>> expression);
   ICommandBuilder<T> OrderByDescending(
                          Expression<Func<T, object>> expression);
   ICommandBuilder<T> Select(Expression<Func<T, object>> expression);

    Command Build();
}
```

Пример использования

```
var command = new CommandBuilder()
    .From<Companies>()
    .Where(x => x.YearEstablished > 2000 && x.NumberOfEmployees < 100)
    .OrderBy(x =>x.Country)
    .Select(x => new {
         x.CompanyName,
         x.Country,
         x.City })
    .Build();
```

Добро пожаловать в мир LINQ выражений!

Add
AddAssign
AddAssignChecked
AddChecked
And
AndAlso
AndAssign
ArrayIndex
ArrayLength
Assign
Block
Call
Coalesce
Conditional
Constant
Convert
ConvertChecked
DebugInfo
Decrement
Default
Divide
DivideAssign
Dynamic
Equal
ExclusiveOr
ExclusiveOrAssign
Extension
Goto
GreaterThan
GreaterThanOrEqual
Increment
Index
Invoke
IsFalse
IsTrue
Label
Lambda
LeftShift
LeftShiftAssign
ListInit
Loop
MemberAccess
MemberInit
Modulo
ModuloAssign
Multiply
MultiplyAssign
MultiplyAssignChecked
MultiplyChecked
Negate
NegateChecked
New
NewArrayBounds
NewArrayInit
Not
NotEqual
OnesComplement
Or
OrAssign
OrElse
Parameter
PostDecrementAssign
PostIncrementAssign
Power
PowerAssign
PreDecrementAssign
PreIncrementAssign
Quote
RightShift
RightShiftAssign
RuntimeVariables
Subtract
SubtractAssign
SubtractAssignChecked
SubtractChecked
Switch
Throw
Try
TypeAs
TypeEqual
TypeIs
UnaryPlus
Unbox

Никто не ожидает разбора деревьев LINQ выражений!

Памятка для выживания среди деревьев LINQ выражений

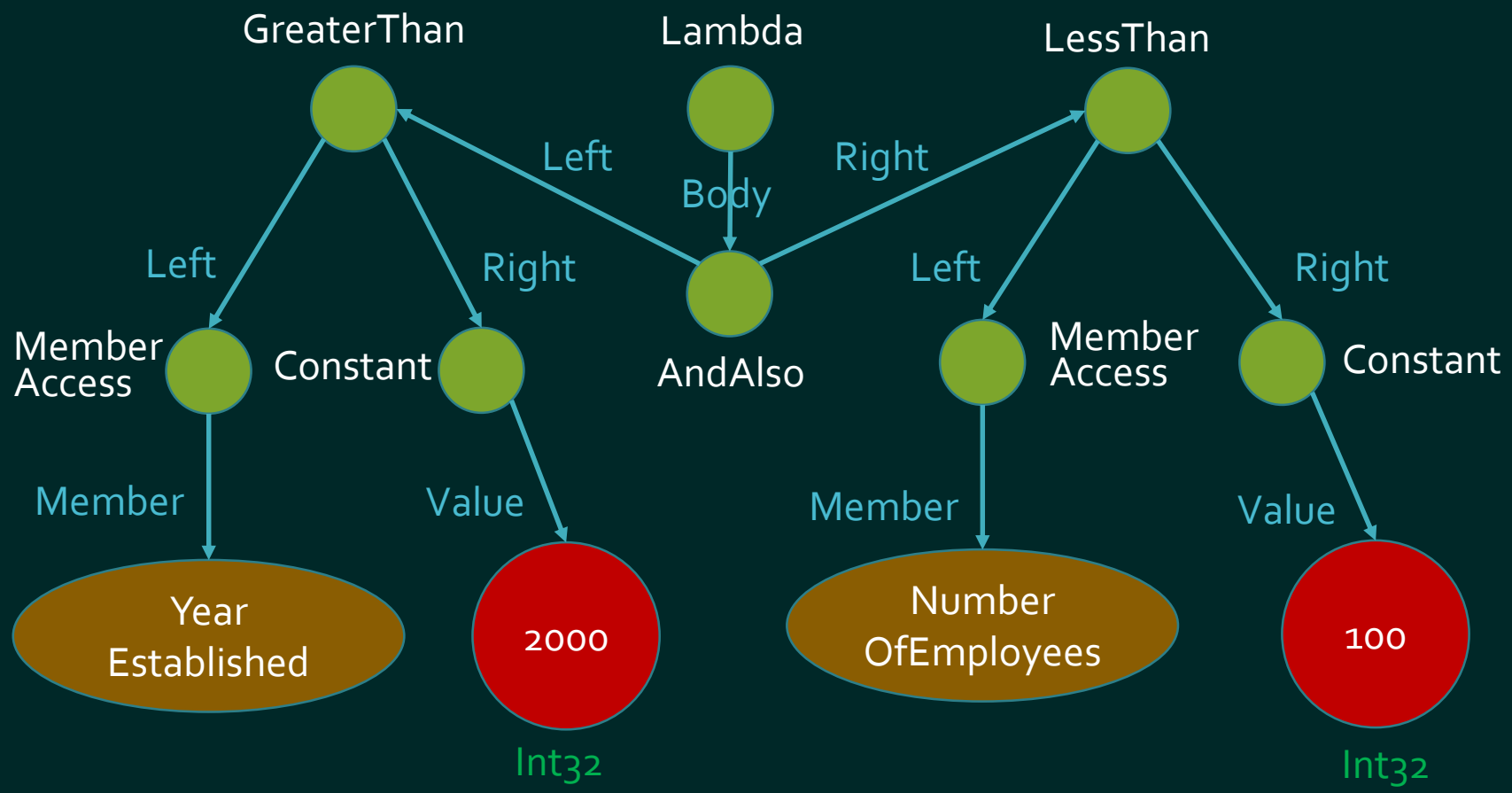
1. Не пугайтесь
2. Не берите плату за разбор первого дерева
3. Пользуйтесь опенсорсным кодом
4. Наслаждайтесь!

Выразительность DSL

```
var command = new CommandBuilder()
    .From<Companies>()
    .Where(x => x.YearEstablished > 2000 && x.NumberOfEmployees < 100)
    .OrderBy(x => x.Country)
    .Select(x => new {
        x.CompanyName,
        x.Country,
        x.City })
    .Build();
```

Пример дерева LINQ выражений

```
Expression<Func<Companies, bool>> expression = x =>
    x.YearEstablished > 2000 && x.NumberOfEmployees < 100;
```



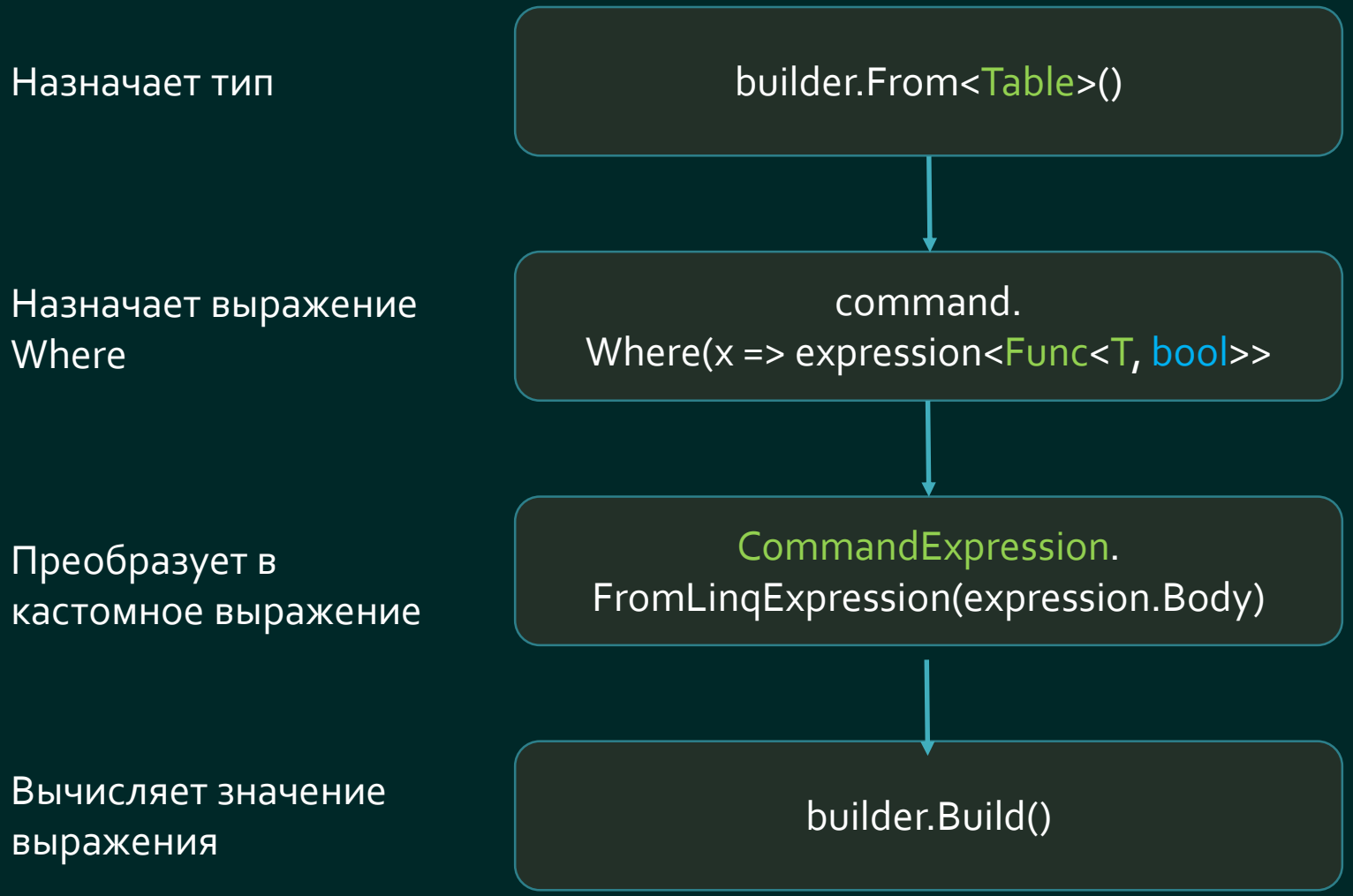
Памятка для разбора LINQ выражений

- Определите кастомный тип для ваших выражений
- Напишите парсер LINQ выражений для конвертации в объекты кастомного типа
- Напишите преобразование выражений LINQ Call в операции вашей предметной области
- Определите унарные и бинарные операторы (при необходимости – переопределите другие операторы C#) для кастомных выражений
- Напишите интерпретатор кастомных выражений, в случае, когда на выходе – текст, он может сводится к простому переназначению метода ToString()

Демо

Типизированный построитель комманд SQL

Фазы работы типизированного построителя команд



Где мы и что дальше?

- Определили интерфейсы ICommandBuilder и ICommandBuilder<T>
- Реализовали CommandBuilder
- Реализовали CommandExpression
  - парсер LINQ выражений
  - вычисление значения выражений

- Изменить интерфейсы ICommandBuilder и ICommandBuilder<T> для поддержки динамических объектов (не добавляя зависимость от System.Dynamic)
- Определить DynamicCommandExpression (наследует CommandExpression)
- Реализовать конверторы DynamicCommandExpression

Типизированный построитель команд

```
interface ICommandBuilder
{
    ICommandBuilder<T> From<T>();
}

interface ICommandBuilder<T>
{
   ICommandBuilder<T> Where(Expression<Func<T, bool>> expression);
   ICommandBuilder<T> OrderBy(Expression<Func<T, object>> expression);
   ICommandBuilder<T> OrderByDescending(
                          Expression<Func<T, object>> expression);
   ICommandBuilder<T> Select(Expression<Func<T, object>> expression);

    Command Build();
}
```

Реализовано

```
var result = db.From<Companies>()
             .Where(x => x.CompanyName == "DynamicSoft")
             .Select(x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });

dynamic x = new DynamicQueryExpression();
var result = db.From(x.Companies)
             .Where(x.CompanyName == "DynamicSoft")
             .Select(x.CompanyName, x.YearEstablished);
```

Осталось реализовать

```
var result = db.From<Companies>()
             .Where(x => x.CompanyName == "DynamicSoft")
             .Select(x =>
                new
                {
                   c.CompanyName,
                   c.YearEstablished
                });

dynamic x = new DynamicQueryExpression();
var result = db.From(x.Companies)
             .Where(x.CompanyName == "DynamicSoft")
             .Select(x.CompanyName, x.YearEstablished);
```

Гибридный построитель команд

```
interface ICommandBuilder
{
    ICommandBuilder<T> From<T>();
    ICommandBuilder<object> From<T>(CommandExpression expression);
}
```

Гибридный построитель команд

```
interface ICommandBuilder<T>
{
    ICommandBuilder<T> Where(Expression<Func<T, bool>> expression);
    ICommandBuilder<T> Where(CommandExpression expression);
    ICommandBuilder<T> OrderBy(Expression<Func<T, object>> expression);
    ICommandBuilder<T> OrderBy(params CommandExpression[] columns);
    ICommandBuilder<T> OrderByDescending(
                            Expression<Func<T, object>> expression);
    ICommandBuilder<T> OrderByDescending(
                            params CommandExpression[] columns);
    ICommandBuilder<T> Select(Expression<Func<T, object>> expression);
    ICommandBuilder<T> Select(params CommandExpression[] columns);
    Command Build();
}
```

Демо

Динамический построитель комманд SQL

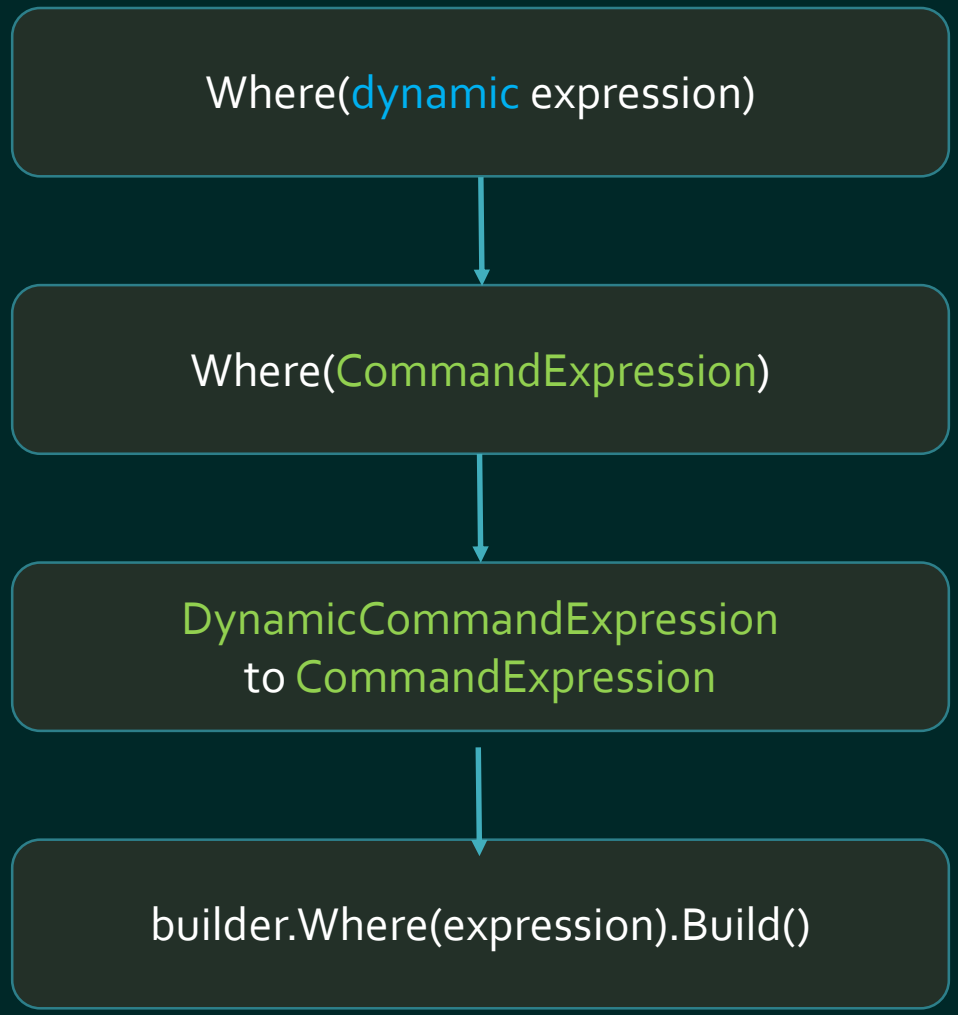
Фазы работы динамического построителя команд

Вызов с динамическим параметром

Найти подходящий метод класса

Преобразовать в типизированное выражение

Далее следовать фазам работы типизированного построителя



У нас все?

- Нет, если все прошло нормально, у нас еще должно оставаться около 15 минут
- Мы успешно реализовали API с операциями, принимающими входные аргументы как статически заданных типов, так и динамические
- Мы не протестировали, как будут себя вести операции API при возврате значений, присваемых динамическим объектам
- Мы покажем, как расширить нашу реализацю тривиальным процессором команд CommandProcessor

Использование процессора команд

```
// Типизированный
var command = commandBuilder
      .From<Companies>()
      .Build();
var commandProcessor = new CommandProcessor(command);
var result = commandProcessor.FindOne<Companies>();

// Динамический
var x = new DynamicCommandExpression();
var command = commandBuilder
      .From(x.Companies)
      .Build();
var commandProcessor = new CommandProcessor(command);
var row = commandProcessor.FindOne(x.Companies);
```

Типизированный CommandProcessor

```
public interface ICommandProcessor
{
    T FindOne<T>();
    IEnumerable<T> FindAll<T>();
}

public abstract class CommandProcessor : ICommandProcessor
{
    protected readonly Command _command;

    protected CommandProcessor(Command command)
    {
        _command = command;
    }
    protected abstract IEnumerable<IDictionary<string, object>> Execute();
}
```

Возврат типизированных результатов

Единичные строки:

- Метод расширения (extension method) для IDictionary<string, object>
  - ToObject<T>: возвращает T
- Метод расширения для object
  - ToDictionary: возвращает IDictionary<string, object>

Массивы:

- Метод расширения для IEnumerable<IDictionary<string, object>>
  - ToObject<T>: возвращает T
- Метод расширения для object
  - ToEnumerable: возвращает IEnumerable<IDictionary<string, object>>

Пользоваться динамическим клиентом пока громоздко

```
dynamic x = new DynamicCommandExpression();
var command = SelectAllCommand();
var commandProcessor = new FakeCommandProcessor(command);

// Единичный результат
var result = commandProcessor.FindOne();
Assert.AreEqual("DynamicSoft", result["CompanyName"]);

// Множественный результат
var result = commandProcessor.FindAll();
Assert.AreEqual(2, result.Count());
Assert.AreEqual("DynamicSoft", result.First()["CompanyName"]);
Assert.AreEqual("StaticSoft", result.Last()["CompanyName"]);
```

Пересмотренный ICommandProcessor

```
public interface ICommandProcessor
{
    T FindOne<T>() where T : class;
    ResultRow FindOne();
    ResultRow FindOne(CommandExpression expression);
    IEnumerable<T> FindAll<T>() where T : class;
    IEnumerable<ResultRow> FindAll();
    ResultCollection FindAll(CommandExpression expression);
}
```

DynamicResultRow и DynamicResultCollection

```
public class DynamicResultRow :
    ResultRow, IDynamicMetaObjectProvider
{
    internal DynamicResultRow(IDictionary<string, object> data)
        : base(data)
    {
    }
}

public class DynamicResultCollection :
    ResultCollection, IDynamicMetaObjectProvider
{
    internal DynamicResultCollection(IEnumerable<ResultRow> data)
        : base(data)
    {
    }
}
```

Динамический клиент становится удобным

```
dynamic x = new DynamicCommandExpression();
var command = SelectAllCommand();
var commandProcessor = new FakeCommandProcessor(command);

// Dynamic result
// Requires BindGetMember overload
var result = commandProcessor.FindOne();
Assert.AreEqual("DynamicSoft", result.CompanyName);

// Typed result
// Requires BindConvert overload
Companies result = commandProcessor.FindOne();
Assert.AreEqual("DynamicSoft", result.CompanyName);
```

Демо

Процессор команд

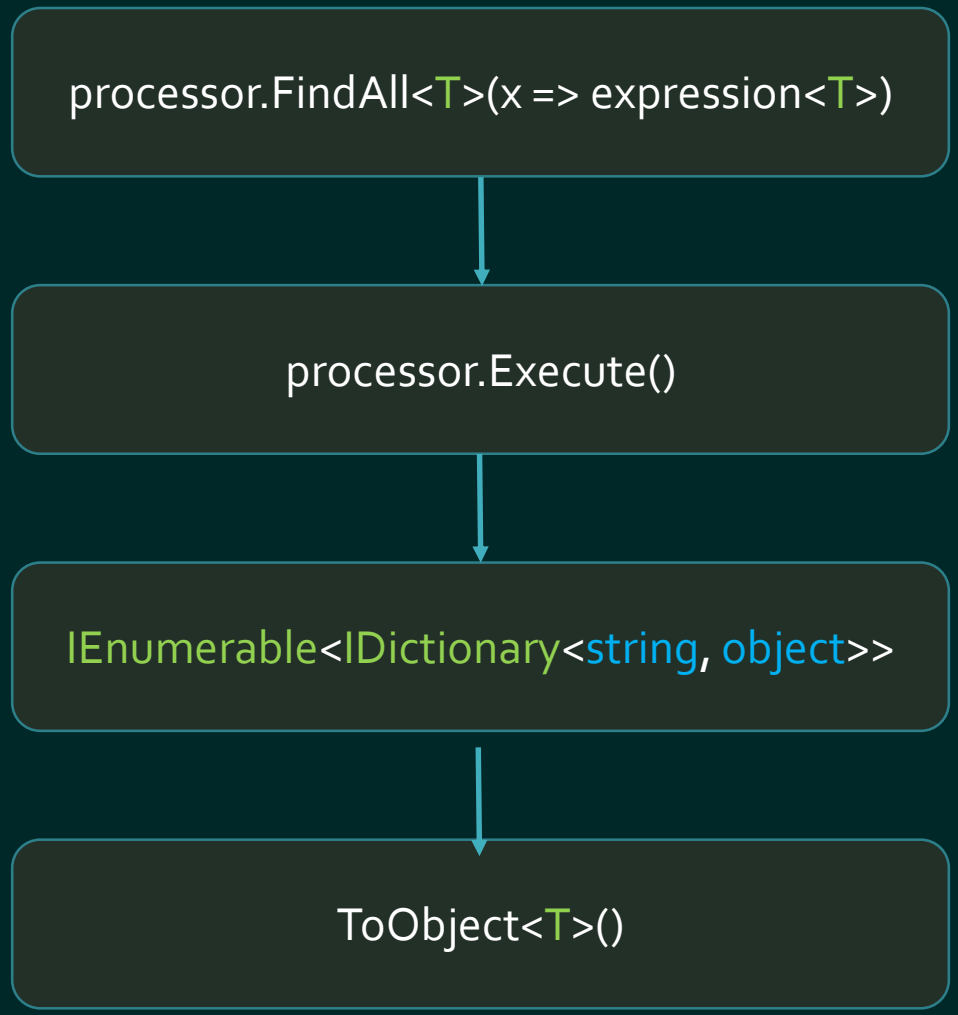
Фазы работы типизированного процессора команд

Вызов метода API с LINQ выражением

Выполнить команду языка SQL

Вернуть нативный результат

Преобразовать результат в типизированный объект



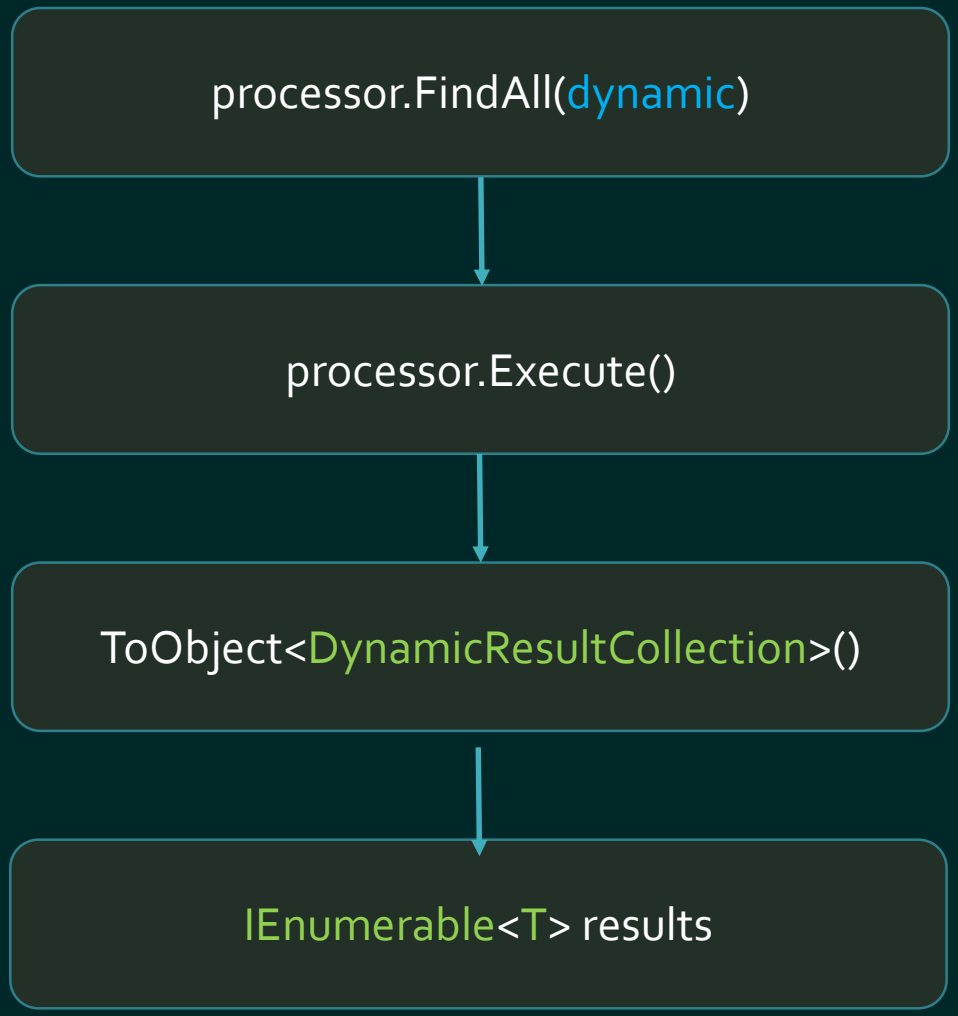
Фазы работы динамического процессора команд

Вызов метода API с динамическим выражением

Следовать фазам работы типизированного процессора

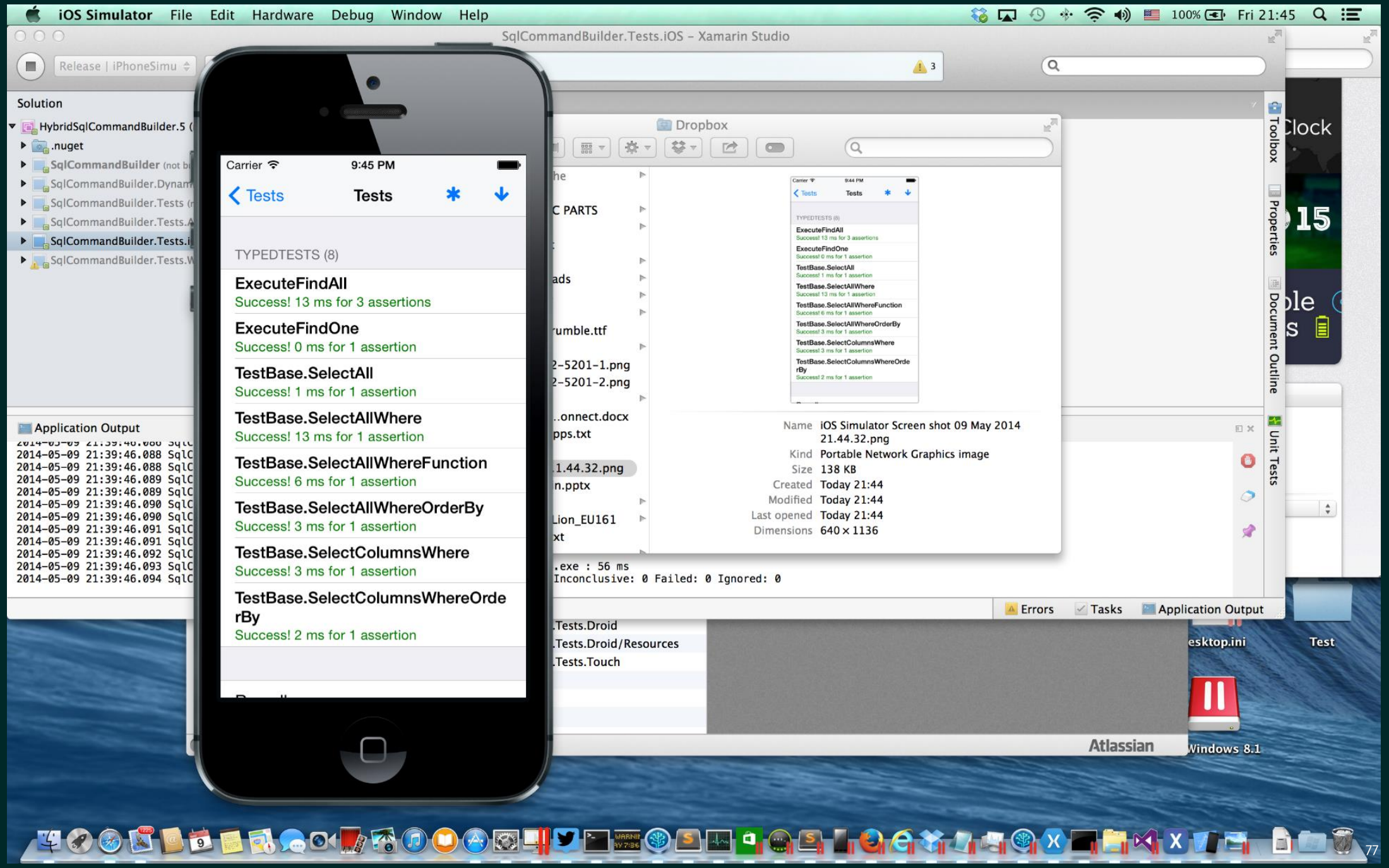
Преобразовать нативный результат в динамический

Преобразовать динамический результат во множественный



Поддержка различных платформ

- Как типизированный, так и динамический API реализуются в виде портативных библиотек (PCL) с поддержкой современных платформ
- Типизированная библиотека может поддерживать устаревшие платформы, такие как Silverlight 4 или Windows Phone 7, для которых отсутствует поддержка динамических типов
- Поддержка новых платформ (.NET Core)


iOS Simulator File Edit Hardware Debug Window Help
SqlCommandBuilder.Tests.iOS - Xamarin Studio
Carrier 9:45 PM
Tests
Tests
TYPEDTESTS (8)
ExecuteFindAll
Success! 13 ms for 3 assertions
ExecuteFindOne
Success! 0 ms for 1 assertion
TestBase.SelectAll
Success! 1 ms for 1 assertion
TestBase.SelectAllWhere
Success! 13 ms for 1 assertion
TestBase.SelectAllWhereFunction
Success! 6 ms for 1 assertion
TestBase.SelectAllWhereOrderBy
Success! 3 ms for 1 assertion
TestBase.SelectColumnsWhere
Success! 3 ms for 1 assertion
TestBase.SelectColumnsWhereOrderBy
Success! 2 ms for 1 assertion
Dropbox
Name iOS Simulator Screen shot 09 May 2014 21.44.32.png
Kind Portable Network Graphics image
Size 138 KB
Created Today 21:44
Modified Today 21:44
Last opened Today 21:44
Dimensions 640 × 1136
Application Output
Errors
Tasks
Application Output
Atlassian
Windows 8.1

Заключение

- Гибкий дизайн API требует дополнительных усилий

Заключение

- Гибкий дизайн API требует дополнительных усилий
- Дополнительных затраты могут оказаться чрезмерными, если API создается для внутреннего пользования

Заключение

- Гибкий дизайн API требует дополнительных усилий
- Дополнительных затраты могут оказаться чрезмерными, если API создается для внутреннего пользования
- Поддержка статических и динамических типов в одном и том же API дает разработчикам возможность использования API, не меняя своих предпочтений

Заключение

- Гибкий дизайн API требует дополнительных усилий
- Дополнительных затраты могут оказаться чрезмерными, если API создается для внутреннего пользования
- Поддержка статических и динамических типов в одном и том же API дает разработчикам возможность использования API, не меняя своих предпочтений
- Единожды разобравшись с техникой написания динамических расширений, в дальнейшем их можно применять для новых библиотек с минимальными дополнительными затратами

Песня для усвоения пройденного

I Broke My Static Types

похоже на Queen

When compiler acts rough and ruthless
And the meaning is oh so clear
One thousand and one nasty errors
Begin to dance in front of you - oh dear

Are they trying to tell you something?
Your proxy classes are out of date
Or you merged your code with a wrong branch
Or maybe it is just getting late?

I broke my static types

I broke my static types

It finally happened - happened

It finally happened - oh yeah

It finally happened - I broke my types

Oh dear!

I'm one step from the roof edge
I know I'm not quite sublime
And I have only one chance left
Bind all my code at runtime

It won't be run until Monday
And by that time I'll be free
Running through yellow daffodils
Climbing on a banana tree

I broke my static types

But I wrote dynamic code

It finally happened - happened

It finally happened - aha

It finally happened - dynamic code!

В проигрыше вспоминаем стратегию гибридного API

- Основной API использует статические типы и не содержит ссылок на System.Dynamic
- В основе API – DSL, основанный на LINQ выражениях, они транслируются в кастомные выражения
- Для каждого метода API, содержащего параметр типа LINQ выражение, есть аналогичный метод с типом кастомного выражения
- Динамические расширения транслируют динамические расширения в подтип кастомного выражения, реализующий IDynamicMetaObjectProvider
- Динамические расширения API упакованы в отдельный модуль (assembly), который устанавливается на платформах с поддержкой DLR

I'm calling now funny methods
My properties’re weird it’s true
I import System.Dynamic these days
But my dear how about you?

I wrote dynamic code

I wrote dynamic code

It finally happened

It finally happened oh yes

It finally happened - dynamic code!

It’s all dynamic code!

And there you have it!

Спасибо!

- Работаю в норвежской компании Miles
- Электронная почта: vagif.abilov@gmail.com
- Twitter: @ooobject
- GitHub: object
- Блог: http://vagifabilov.wordpress.com/
- Статьи на CodeProject
- Поддерживаю библиотеку Simple.OData.Client, участвую в других опенсорсных проектах

- Исходный код для этой презентации можно посмотреть здесь: https://github.com/object/HybridSqlCommandBuilder