Typed? Dynamic? Both!
Cross-platform DSLs in C#
About myself

• Consultant in Miles (several “Great Place to Work” awards in Norway in Europe)
• Mail: vagif.abilov@gmail.com
• Twitter: @ooobject
• GitHub: object
• BitBucket: object
• Blog: http://vagifabilov.wordpress.com/
• Articles: http://www.codeproject.com

• The source code for this presentation can be found at https://github.com/object/HybridSqlSqlCommandBuilder
Designing modern times API
When we were young the grass was greener, database records were retrieved using stored procedures and every single piece of data had a type resolved at compile time
Nowadays people didn’t just abandon their SQL skills in favour of statements generated by so-called «orms», they became so lazy they don’t define their entities relying instead on so-called «dynamic languages» wtf
-And look what you've done to mother! She's worn out fiddling with all your proxy classes.
- There's nowt wrong wi' proxy classes, lad! I've generated more proxy classes than you've had hot dinners!
Elements of functional programming (2007)

- Major additions to the C# language (driven by LINQ)
- Lambda expressions
- Extension methods
- Anonymous types
- Expression trees
Dynamic since 2010

• Earlier C#/.NET versions only supported static binding where method calls were resolved at compile time
• Subtype polymorphic resolution was delayed until run time
• Starting from version 4, in addition to static binding C# support dynamic binding where type member resolution occurs at run time
• Not only it’s now possible to choose at run time which method to execute, it doesn’t have to be a predefined method – any code will do
dynamic client = WeirdWildStuffFactory.GiveMeOneOfThose();

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

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

Assert.AreEqual(42, result);
Example: COM interop

// Without dynamic binding

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

// With dynamic binding

excelApp.Cells[1,1].Value = "Name";
var range2010 = excelApp.Cells[1,1];
Example: Accessing a database

// With Entity Framework

public User FindUserByEmail(string email)
{
    return _context
        .Users.Where(x => x.Email == email)
        .FirstOrDefault();
}

// With Simple.Data

public User FindUserByEmail(string email)
{
    return Database.Open()
        .Users.FindAllByEmail(email)
        .FirstOrDefault();
}
Example: implementing Gherkin scenario

**Background**: Validate users

**Given** the following users exist in the database:

<table>
<thead>
<tr>
<th>Name</th>
<th>Birth date</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1940-10-09</td>
<td>1.80</td>
</tr>
<tr>
<td>Paul</td>
<td>1942-06-18</td>
<td>1.80</td>
</tr>
<tr>
<td>George</td>
<td>1943-02-25</td>
<td>1.77</td>
</tr>
<tr>
<td>Ringo</td>
<td>1940-07-07</td>
<td>1.68</td>
</tr>
</tbody>
</table>

```csharp
public void GivenTheFollowingUsersExist(Table table)
{
    IEnumerable<dynamic> users = table.CreateDynamicSet();
    db.Users.Insert(users);
}
```
Popular libraries using C# dynamics

- SignalR (~400,000 downloads from NuGet)
- Nancy (~90,000 downloads)
- ReflectionMagic (~70,000 downloads)
- Fasterflect (~60,000 downloads)
- Simple.Data (~50,000 downloads)
- EasyHttp (~25,000 downloads)
- ImpromptuInterface (~25,000 downloads)
- Oak (~10,000 downloads)
So should you use dynamic libraries?

- This talk is not about making a choice on your behalf
So should you use dynamic libraries?

- This talk is not about making a choice on your behalf
- In fact, this talk is about leaving you multiple choices
So should you use dynamic libraries?

• This talk is not about making a choice on your behalf
• In fact, this talk is about leaving you multiple choices
• Moreover, this talk is about giving you an opportunity to change your mind later with minimum effort
So should you use dynamic libraries?

• This talk is not about making a choice on your behalf
• In fact, this talk is about leaving you multiple choices
• Moreover, this talk is about giving you an opportunity to change your mind later with minimum effort
• This talk is about creating libraries that would expose both typed and dynamic API – but not as two different APIs
So should you use dynamic libraries?

• This talk is not about making a choice on your behalf
• In fact, this talk is about leaving you multiple choices
• Moreover, this talk is about giving you an opportunity to change your mind later with minimum effort
• This talk is about creating libraries that would expose both typed and dynamic API – but not as two different APIs
• We will show how to make a single API that can be called from either static typed or dynamic client
So should you use dynamic libraries?

- This talk is not about making a choice on your behalf
- In fact, this talk is about leaving you multiple choices
- Moreover, this talk is about giving you an opportunity to change your mind later with minimum effort
- This talk is about creating libraries that would expose both typed and dynamic API – but not as two different APIs
- We will show how to make a single API that can be called from either static typed or dynamic client
- Our hybrid API will have hybrid packaging – it will disable dynamic support when installed on platforms that lack runtime method binding
So should you use dynamic libraries?

• This talk is not about making a choice on your behalf
• In fact, this talk is about leaving you multiple choices
• Moreover, this talk is about giving you an opportunity to change your mind later with minimum effort
• This talk is about creating libraries that would expose both typed and dynamic API – but not as two different APIs
• We will show how to make a single API that can be called from either static typed or dynamic client
• Our hybrid API will have hybrid packaging – it will disable dynamic support when installed on platforms that lack runtime method binding
Having said all that...

• The benefits from bringing dynamic objects to C# often come from interop with external services that don’t expose .NET API

• However interop between dynamic and statically typed objects in C# may outweigh the benefits brought by dynamic typing

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

• Beware ad-hoc use of dynamic objects in statically typed code, especially when dealing with collections
Database access revisited

```csharp
var result = db.Companies
    .Where(x => x.CompanyName == "DynamicSoft")
    .Select(x =>
        new {
            c.CompanyName,
            c.YearEstablished
        });
```
Should dynamic API look like this?

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

var result = db.Companies
            .FindByCompanyName("DynamicSoft")
            .SelectCompanyNameAndYearEstablished();
```
Or like this?

```csharp
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);
```
API style is a fairly opinionated question

- My API is my castle, any naming scheme is a matter of a personal preference
API style is a fairly opinionated question

• My API is my castle, any naming scheme is a matter of personal preference
• So we will leave method naming to personal opinion and ask a different question
API style is a fairly opinionated question

- My API is my castle, any naming scheme is a matter of personal preference
- So we will leave method naming to personal opinion and ask a different question
- Do you want to publish one API or two APIs?
API style is a fairly opinionated question

- My API is my castle, any naming scheme is a matter of personal preference
- So we will leave method naming to personal opinion and ask a different question
- Do you want to publish one API or two APIs?
What if we have common API core?

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

- This interface defines core operation set of our internal DSL and is shared by static typed and dynamic clients.
- Typed and dynamic clients differ in what kind of arguments they send to the operations: static typed or dynamic respectively.
One API – two paradigms

```csharp
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);
```

- Exposing single API with two similar looking parameter syntax flavors unifies API operations across paradigms
- API users learn the same API operations no matter what paradigm they choose
- API users can switch paradigms at relatively low costs
What are the extra costs?

• Additional dynamic layer doesn’t re-implement functionality, just servers as an adapter
• Code size of the dynamic layer isn’t affected much by choice of domain
• Example project:
  • Core library: 20 classes, 681 lines of code
  • Dynamic extensions library: 3 classes, 76 lines of code
• Real world project (Simple.OData.Client):
  • Core library: 96 classes, 2760 lines of code
  • Dynamic extensions library: 6 classes, 59 lines of code
Strategy for a hybrid API

- Core API is static typed and packaged in assembly that doesn’t reference types from System.Dynamic namespace
- API exposes a DSL based on LINQ expressions
- API defines custom expression type without a public constructor
- For every API method that accepts LINQ expression parameter there is another method accepting custom expression type
- Dynamic extensions include custom expression subtype that is derived from custom expression and implements IDynamicMetaObjectProvider
- Dynamic extensions for the API are packaged in a different assembly that is deployed on platforms with DLR support
public class DynamicQueryExpression : QueryExpression, IDynamicMetaObjectProvider
{
    public DynamicQueryExpression()
    {
    }
}

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

public class QueryExpression
{
    // no public constructor
}

public class DynamicQueryExpression : QueryExpression, IDynamicMetaObjectProvider
{
    public DynamicQueryExpression()
    {
    }
}
Client code

Static typed client

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

Dynamic client

```csharp
dynamic x = new DynamicQueryExpression();
var finder = new Finder();
var result = finder.Find(x.Companies).Where(
    x.CompanyName.StartsWith("D") &&
    x.YearEstablished > 2000);
```
Plan for the rest of the talk

- Go through the case study for a hybrid SQL command builder
- The implementation reflects the design of the open source library Simple.OData.Client but replaces OData queries with SQL queries that should be familiar for everyone
- Grow the implementation incrementally starting with primitive untyped builder, followed by typed builder based on LINQ expressions, and completed with dynamic support
- Extend the implementation by adding fake SQL command processor to demonstrate how to handle typed and dynamic return results
Case study

Hybrid SQL Command Builder
Let’s write something like LINQ provider

```csharp
var result1 = from c in db.Companies
               where c.CompanyName.StartsWith("D")
               orderby c.CompanyName
               select new
               {
                   c.CompanyName,
                   c.YearEstablished
               };

var result2 = db.Companies
               .Where(x => x.CompanyName.StartsWith("D"))
               .OrderBy(x => x.CompanyName)
               .Select(x =>
                           new
                           {
                               c.CompanyName,
                               c.YearEstablished
                           });
```
Here’s a supported subset of SQL commands

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

```csharp
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();
}

// Usage example

var command = new CommandBuilder()
    .From("Companies")
    .Where("YearEstablished>2000 AND NumberOfEmployees<100")
    .OrderBy("Country")
    .Select("CompanyName", "Country", "City")
    .Build();
```
public class Command
{
    private string _table;
    private string _where;
    private List<string> _selectColumns;
    private List<KeyValuePair<string, bool>> _orderByColumns;

    ...
}

Command class fields
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();
}
Moving on to typed version

```csharp
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();
}
```
Usage example

```csharp
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();
```
Enter LINQ expression trees

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
Nobody expects to parse LINQ expression trees!
LINQ expression trees survival guide

1. Don’t be scared
2. Don’t get paid for your first tree
3. Learn some tricks from OSS
4. Have fun!
Expressiveness of DSL

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

Lambda
AndAlso
GreaterThan
Member
Access
Constant
Value
Member
Access
Member
Value
Int32
Int32
LINQ expression processing checklist

- Create custom expression type
- Implement LINQ expression parser that converts expression trees to instances of the custom expression type
- Create function maps to map LINQ Call expressions to your domain operations
- Define unary and binary operators that can be applied to custom expressions
- Overload C# operators for custom expressions
- Implement custom expression evaluation, in case the output is a text string it can be as simple as overloaded ToString() method
Demo

Typed SQL command builder
Typed command builder workflow

Assigns the type: `builder.From<Table>()`

Assigns Where expression: `command.Where(x => expression<Func>T, bool>)`

Converts to custom expression: `CommandExpression.FromLinqExpression(expression.Body)`

Evaluates the expression: `builder.Build()`
Where are we and what’s next?

- Defined ICommandBuilder and ICommandBuilder<T> interfaces
- Implemented CommandBuilder
- Implemented CommandExpression
  - LINQ expression parsing
  - Expression evaluation

- Revise ICommandBuilder and ICommandBuilder<T> interfaces to accept dynamic objects (without adding dependency on System.Dynamic)
- Define DynamicCommandExpression (derived from CommandExpression)
- Implement DynamicCommandExpression conversion
**Typed command builder**

```csharp
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();
}
```
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);
To be implemented

```csharp
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);
```
Recalling our strategy

- Core API is static typed and packaged in assembly that doesn’t reference types from System.Dynamic namespace
- API exposes a DSL based on LINQ expressions
- API defines custom expression type without a public constructor
- For every API method that accepts LINQ expression parameter there is another method accepting custom expression type
- Dynamic extensions include custom expression subtype that is derived from custom expression and implements IDynamicMetaObjectProvider
- Dynamic extensions for the API are packaged in a different assembly that is deployed on platforms with DLR support
Hybrid command builder

```csharp
interface ICommandBuilder
{
    ICommandBuilder<T> From<T>();
    ICommandBuilder<object> From<T>(
        CommandExpression expression);
}
```
Hybrid command builder

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();
}
interface ICommandBuilder<T>
{
    ... 
    ICommandBuilder<T> Where(
        Expression<Func<T, bool>> expression);  
    ICommandBuilder<T> Where(
        CommandExpression expression);  
    ... 
}

// Typed client
builder.Where(x => x.CompanyName == "DynamicSoft");

// Dynamic client
x = new DynamicCommandExpression();
builder.Where(x.DynamicName == "DynamicSoft");
interface ICommandBuilder<T>
{
    ...

    ICommandBuilder<T> OrderBy(
        Expression<Func<T, object>> expression);
    ICommandBuilder<T> OrderBy(
        params CommandExpression[] columns);

    ...
}

// Typed client
builder.OrderBy(x => x.CompanyName);

// Dynamic client
x = new DynamicCommandExpression()
builder.OrderBy(x.CompanyName);
interface ICommandBuilder\<T\>
{
    ...

    ICommandBuilder\<T\> Select(
        Expression<Func\<T, object\>> expression);
    ICommandBuilder\<T\> Select(
        params CommandExpression[] columns);

    ...
}

// Typed client
builder.Select(x => new { x.Country, x.CompanyName });

// Dynamic client
x = new DynamicCommandExpression()
builder.Select(x.Country, x.CompanyName);
Adding method overloads is trivial

```csharp
public ICommandBuilder<T> Where(
    Expression<Func<T, bool>> expression)
{
    _command.Where(
        CommandExpression.FromLinqExpression(expression.Body));
    return this;
}

public ICommandBuilder<T> Where(
    CommandExpression expression)
{
    _command.Where(expression);
    return this;
}
```
Adding dynamic support is more demanding

- The easiest (and widely used) method to implement a class with dynamic support is to derive it from `DynamicObject`.
- This method won’t work because our dynamic class needs to be derived from `CommandExpression`.
- We will have to take a more complicated path and implement `IDynamicMetaObjectProvider` interface (in addition to deriving class from `CommandExpression`).
- Being a meta object provider, `DynamicCommandExpression` will need to create instances of another custom class that is derived from `DynamicMetaObject`.
- `DynamicMetaObject-derived` class will provide method overloads that implement custom behavior with runtime binding.
Demo

Dynamic SQL command builder
Dynamic command builder workflow

- Call with dynamic argument
- Finds suitable method overload
- Converts to typed expression
- Follow typed builder workflow

Are we finished?

• No, if everything has been going smooth we should have about 15 minutes left
• We have now successfully implemented an API with operations that accept both static typed and dynamic input arguments
• We haven’t tested how client code deals with API operations return values in case these values represent instances of dynamic objects
• We will now extend our implementation with a trivial CommandProcessor that executes SQL commands and returns either static typed or dynamic results (consisting of single rows or collections of rows)
Command processor usage

// Typed

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

// Dynamic

```csharp
var x = new DynamicCommandExpression();
var command = commandBuilder
    .From(x.Companies)
    .Build();
var commandProcessor = new CommandProcessor(command);
var row = commandProcessor.FindOne(x.Companies);
```
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();
}
Returning typed results

Single rows:
- Extension method for `IDictionary<string, object>`
  - `ToObject<T>`: returns T
- Extension method for `object`
  - `ToDictionary`: returns `IDictionary<string, object>`

Collections:
- Extension method for `IEnumerable<IDictionary<string, object>>`
  - `ToObject<T>`: returns T
- Extension method for `object`
  - `ToEnumerable`: returns `IEnumerable<IDictionary<string, object>>`
Dynamic clients require result cast

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

// Single row

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

// Collection

var result = commandProcessor.FindAll();
Assert.AreEqual(2, result.Count());
Assert.AreEqual("DynamicSoft", result.First()["CompanyName"]);
Assert.AreEqual("StaticSoft", result.Last()["CompanyName"]);
Revised ICommandProcessor

```csharp
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 and DynamicResultCollection

```csharp
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)
    {
    }
    ...
}
```
Dynamic client is now fine

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);
Demo

Command processor
Typed command processor workflow

Call to a generic method overload

`processor.FindAll<T>(x => expression<T>)`

Executes SQL command

`processor.Execute()`

Returns native result

`IEnumerable<IDictionary<string, object>>`

Converts results to typed objects

`ToObject<T>()`
Dynamic command processor workflow

1. Call to a non-generic method overload: `processor.FindAll(dynamic)`
2. Follows typed execution path: `processor.Execute()`
3. Converts results to typed objects: `ToObject<DynamicResultCollection>()`
4. Converts results to typed objects: `IEnumerable<T> results`
Cross-platform portability

• Both core and dynamic extensions library are implemented as portable class library (PCL) with support for the following platforms:
  • .NET 4 or higher
  • Silverlight 5
  • Windows Phone 8
  • Windows Store apps (Windows 8)
  • Xamarin iOS
  • Xamarin Android

• Core library can also target legacy platforms, such as Silverlight 4 or Windows Phone 7 that don’t offer support for dynamic objects

• Some platforms have more strict type visibility constraints
Demo

Android and iOS 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
Conclusion

- It takes some extra effort to design API properly
- Such additional effort may not always be justified when the API is used internally
- Making API consistent and easy to install and use can be crucial for its success in a large development community
- Support for static and dynamic method binding in a single API gives developers a choice to stick to preferred programming style
- The cost of adding dynamic binding on the top of static type library is negligible
The moral

MAKE HYBRID API NOT HYBRID WAR
And now for something completely different

I Broke My Static Types
probably by 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!
Remember hybrid API strategy

- Core API is static typed and packaged in assembly that doesn’t reference types from System.Dynamic namespace
- API exposes a DSL based on LINQ expressions
- API defines custom expression type without a public constructor
- For every API method that accepts LINQ expression parameter there is another method accepting custom expression type
- Dynamic extensions include custom expression subtype that is derived from custom expression and implements IDynamicMetaObjectProvider
- Dynamic extensions for the API are packaged in a different assembly that is deployed on platforms with DLR support
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!
Thank you!

- I am Vagif Abilov, consultant in Miles
- Mail: vagif.abilov@gmail.com
- Twitter: @ooobject
- GitHub: object
- BitBucket: object
- Blog: http://vagifabilov.wordpress.com/
- Articles: http://www.codeproject.com

- The source code for this presentation can be found at https://github.com/object/HybridSqlCommandBuilder