Эффективное использование Dynamic Language Runtime

Карлен Симонян
разработчик

О чем мы поговорим

- DLR как платформа – зачем?
  - Динамизм в .NET вчера, сегодня, завтра
- Продолжите ряд: IronPython, IronRuby, IronC#?
- C# matters
- DLR Inside Out
  - Узлы вызовов aka callsites, PIC, кэширование, производительность и дополнительные затраты памяти, CPU, etc.
- DLR in the wild
- Infinite loop

DLR как платформа – Timeline

- 2004
  - отсутствуют публично выпущенные динамические языки для .NET
  - до CLR 2.0 еще два года
  - Jim Hugunin присоединяется к Microsoft (будучи автором Jython и уже разрабатывая IronPython)
  - На конференции OSCon 2004 показывается IronPython 0.6
- 2007
  - На конференции MIX’07 показывается IronPython 1.0, анонсируется IronRuby с выходами первых альфа-версий в 2008
  - Впервые вводится понятие Dynamic Language Runtime

DLR как платформа – Timeline

- 2008
  - DLR v0.9
  - IronPython 2.0
  - Две альфа версии IronRuby
- 2009
  - IronPython 2.6
  - IronRuby 0.9. Возможность запуска некоторых Rails-приложений
- 2010
  - C# 4
  - DLR v1.0
  - .NET 4
- 2011 - 2014
  - IronPython 2.7
  - IronRuby 1.1
  - Поддержка .NET 4.0+
  - Возможности рантайма и кодогенерации в составе .NET 4+

DLR как платформа

- Новый DLR добавляет важный набор возможностей в .NET, минуя изменения в CLR
  - Стандартную модель хостинга для динамических языков
  - Инфраструктуру для генерации *быстрого* кода
- .NET 3.5+ уже включает в состав System.Linq.Expressions – основу LINQ и кодогенерации DLR

Продолжите ряд: IronPython, IronRuby, IronC#?

- Несмотря на растущее количество динамических языков на .NET, эволюцию динамизма продолжил именно C#
- C# 4 является прямым продолжением развития темы Iron-языков, собрав в свойствах своего `dynamic` - возможности DLR

WEB ASP.NET FRAMEWORK

.NET BASE CLASS

SPEED LIBRARY GC

DATA WEBAPPS SYSTEM.DYNAMIC FAST CACHING

POLYMORPHIC INLINE CLR JIT AST Ruby On Rails

CALLSITES SYSTEM.LINQ.EXPRESSIONS DYNAMIC

DLR IronPython

EXPRESSIONS COMPILER

IronRuby POWERSHELL

FUNCTIONS CONTEXTS WPF

IronLua

FAST SECURITY RUNTIME NLua

CODE GENERATION EVERYWHERE

TODAY

C# UNIFIED DESIGN

HOSTING IronJS

TEXT COMMON TEXT CACHING WEB

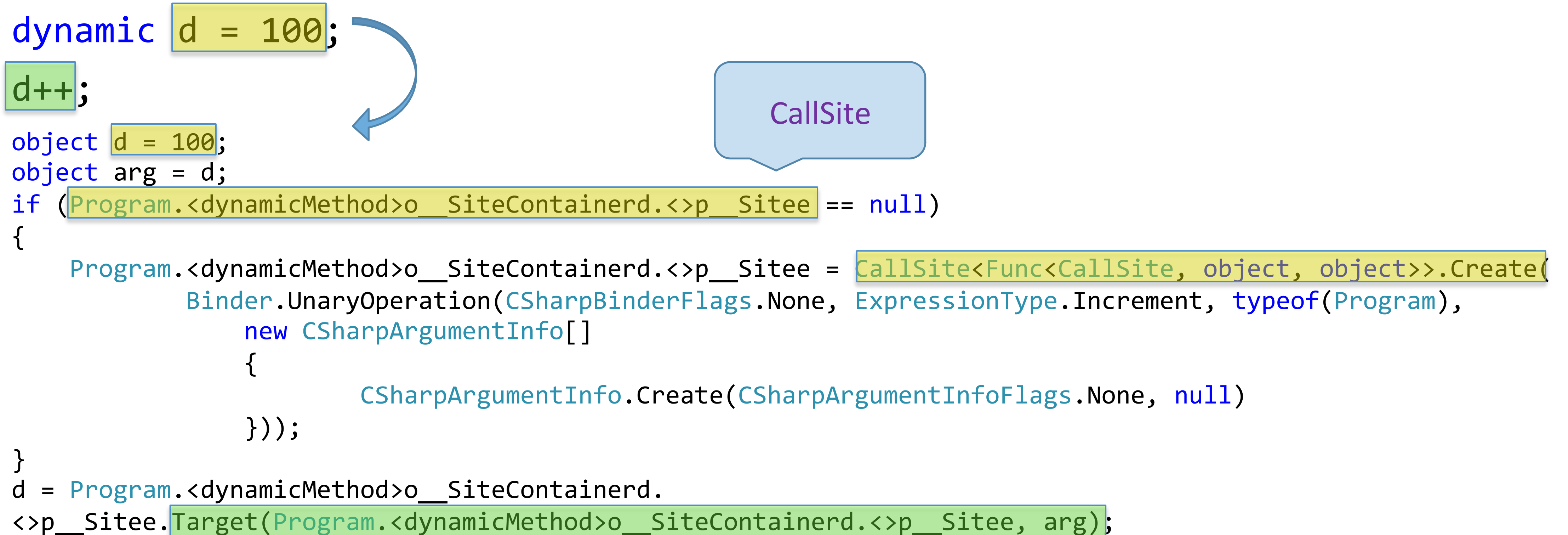
HOSTING POLYMORPHIC

INLINE RUNTIME BINDERS

C# Matters – Inside `dynamic` keyword

- Каждый `dynamic` параметр/переменная является `System.Object`
- Любые операции, связанные с использованием ключевого слова `dynamic` сопровождаются использованием узлов вызовов aka *callsites*.
- Callsite выступает в качестве:
  - Getter/Setter для:
    - Fields
    - Properties
    - Indexers
  - Method/Delegate invoker
  - Унарной/бинарной операции
  - Операции конвертации/приведения типов

C# Matters – dynamic Operation Compilation



C# Matters – Purpose of Callsites

- Callsite обеспечивает динамическую диспетчеризацию методов
- Каждый callsite является самообучаемым
- DLR сам по себе реализует многоуровневый кэш для типов объектов по отношению к которым производит операцию
- Существует 2 реализации кэша: для Iron-языков и C#
  - Но обо всем по порядку ☺

C# Matters - Semantics

- Диспетчеризация методов с использованием callsite сохраняет семантику компилятора C# для разрешения конфликтов при вызове метода, а также выборе перегрузок метода, что означает:
  - Возможность работы с *optional/default* параметрами
  - Поддержку использования *implicit conversion operator* для переданных параметров
  - Специализация *generic-методов* происходит во время исполнения, а не компиляции
  - Использование правил явной, неявной конвертации для числовых типов
    - Да-да, InvalidCastException имеет место быть

DLR Inside Out - Goals

- “Ручное” использование DLR API дает нам возможность:
  - Заменить использование Reflection на более быстрый API (сравнение производительности и ответ на вопрос “почему” будет далее)
  - Перестать пользоваться Reflection emit для создания оберток и т.п., что дает нам легкоподдерживаемый и читаемый код
  - Лучше понять и знать свой инструмент

DLR Inside Out - Infrastructure of `dynamic`

- Компилятор и среда исполнения используют следующие сборки и пространства имен для инфраструктуры:
  - System.Dynamic (System.Core.dll)
  - System.Linq.Expressions (System.Core.dll)
  - System.Runtime.CompilerServices (mscorlib.dll)
  - Microsoft.CSharp (Microsoft.CSharp.dll)

DLR Inside Out – CallSite<T> compile-time

```
dynamic d = 100;
d++;
```

CallSite

```
object d = 100;
object arg = d;
if (Program.<dynamicMethod>o__SiteContainerd.<>p__Sitee == null)
{
    Program.<dynamicMethod>o__SiteContainerd.<>p__Sitee = CallSite<Func<CallSite, object, object>>.Create(
            Binder.UnaryOperation(CSharpBinderFlags.None, ExpressionType.Increment, typeof(Program),
                new CSharpArgumentInfo[]
                {
                        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null)
                }));
}
d = Program.<dynamicMethod>o__SiteContainerd.
<>p__Sitee.Target(Program.<dynamicMethod>o__SiteContainerd.<>p__Sitee, arg);
```

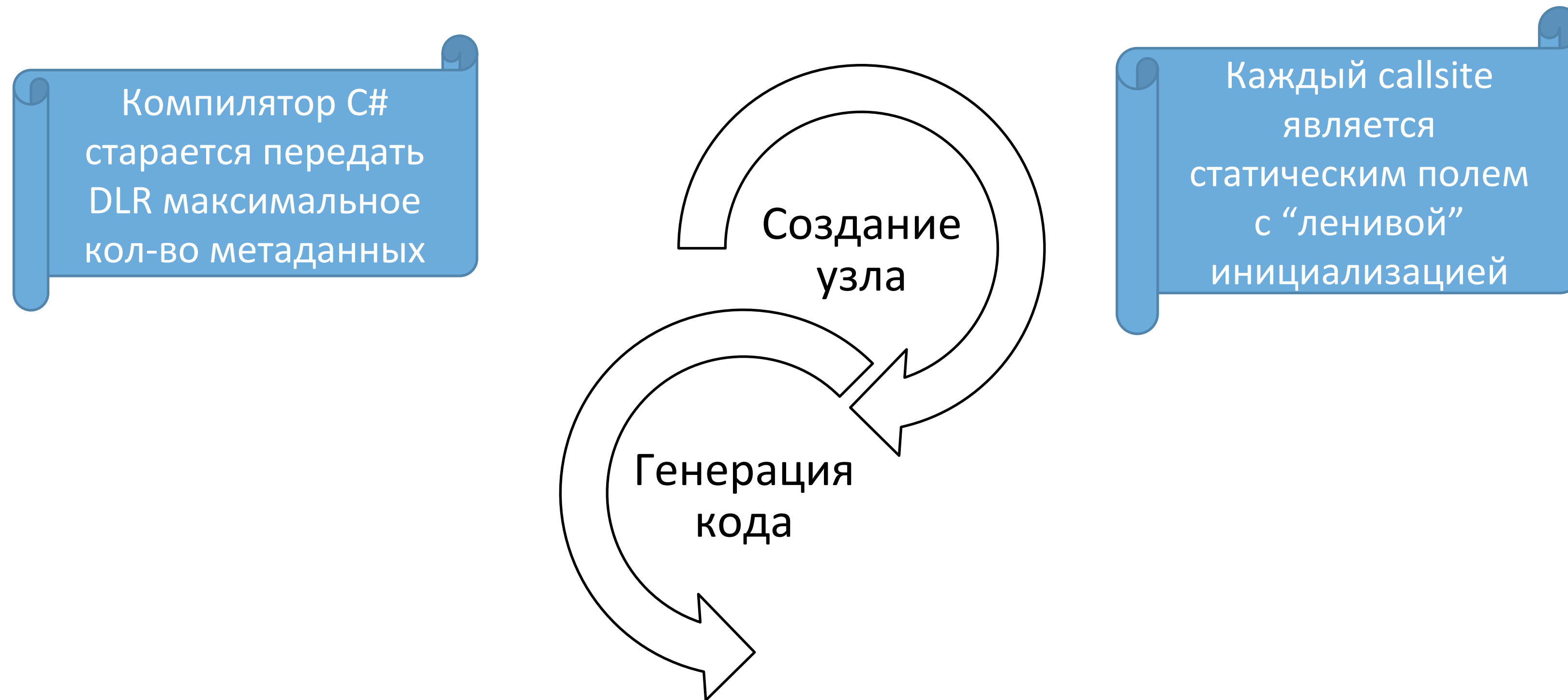
DLR Inside Out – CallSite<T>

```
namespace System.Runtime.CompilerServices.CallSite
{
    public sealed class CallSite<T> : CallSite where T : class
    {
        public T Target;
        public T Update { get; }
        public static CallSite<T> Create(CallSiteBinder binder);
    }
}
```

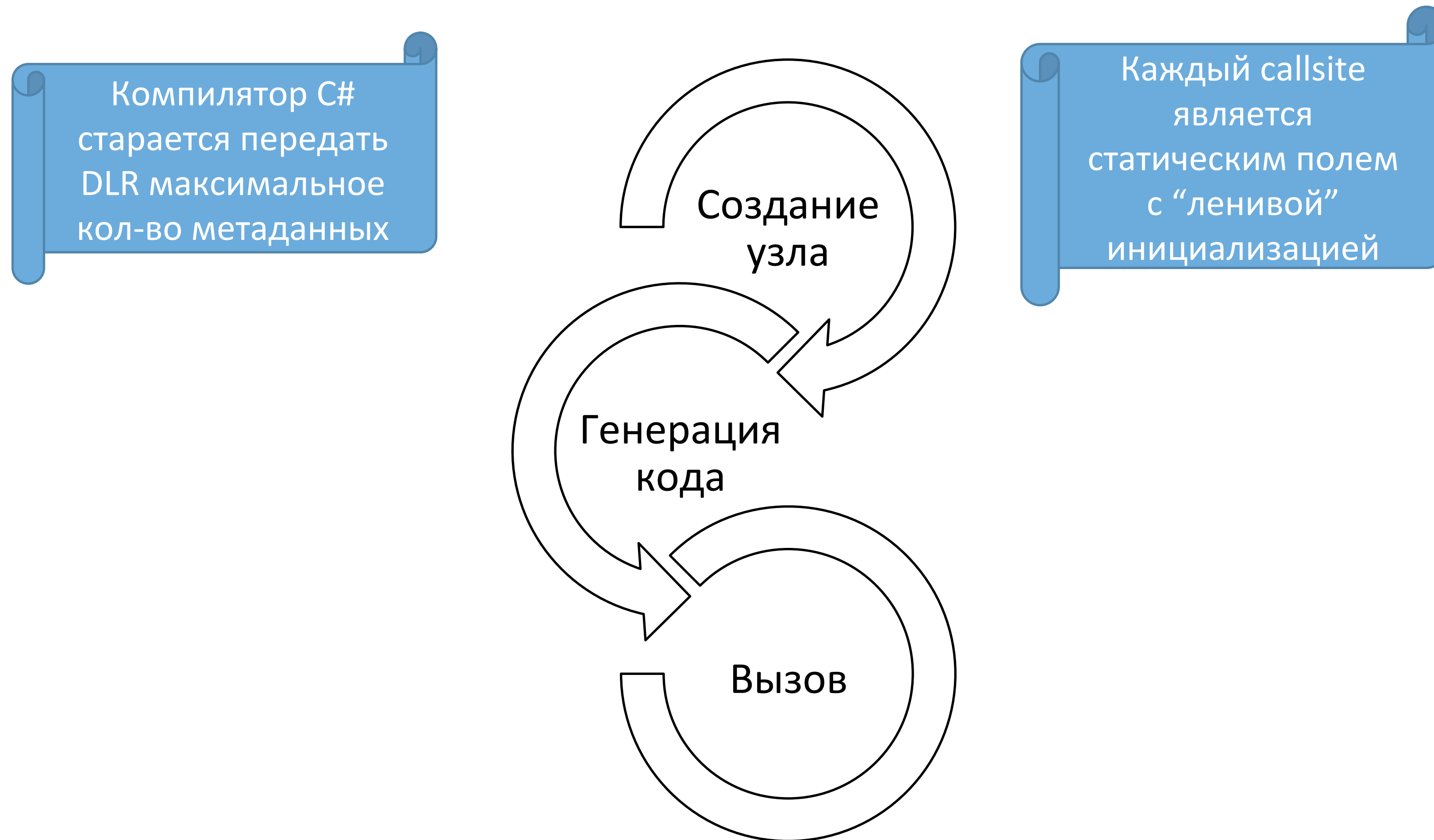
DLR Inside Out – Callsite’s Life Cycle

Каждый callsite является статическим полем с “ленивой” инициализацией

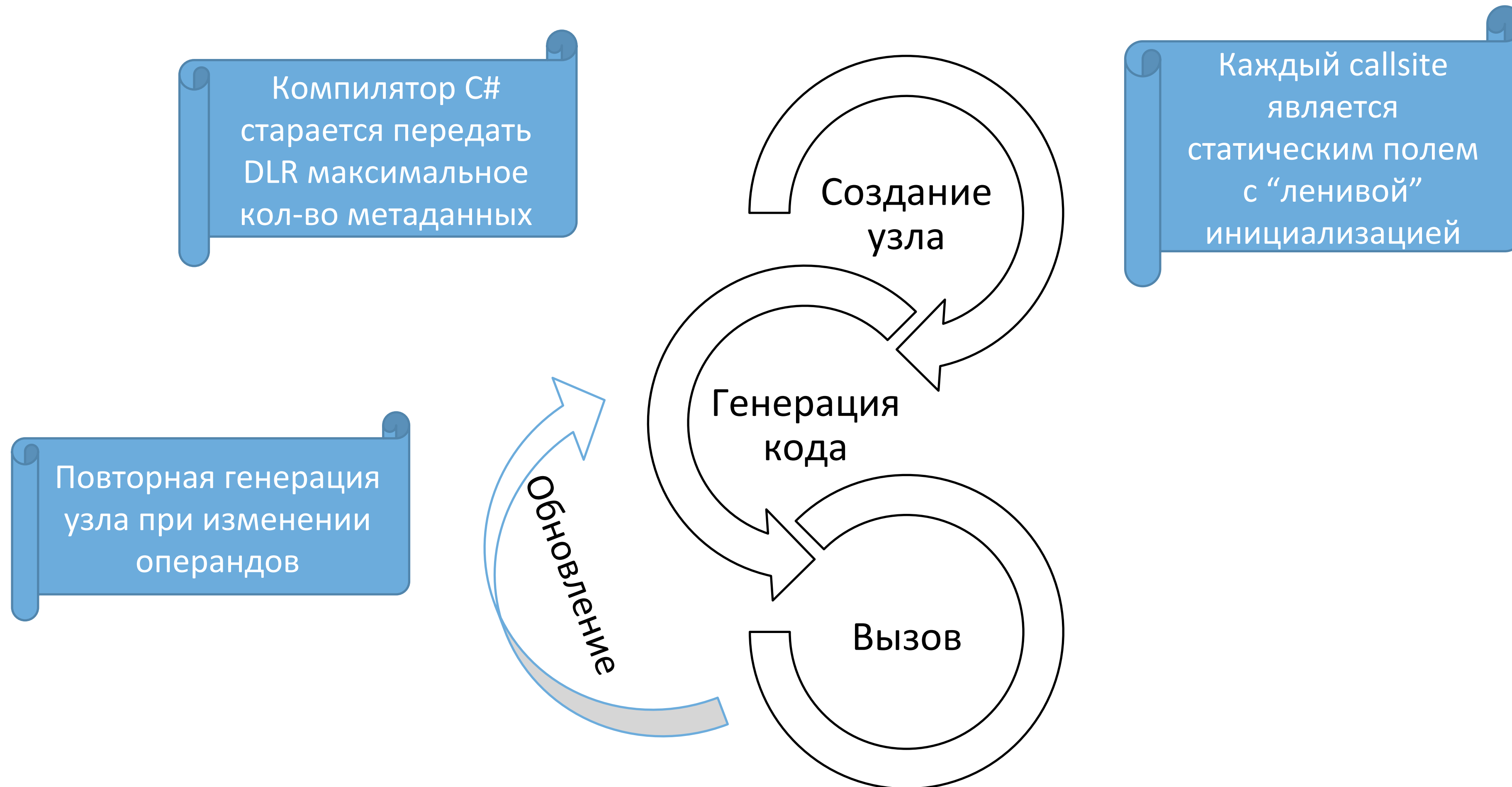
DLR Inside Out – Callsite's Life Cycle



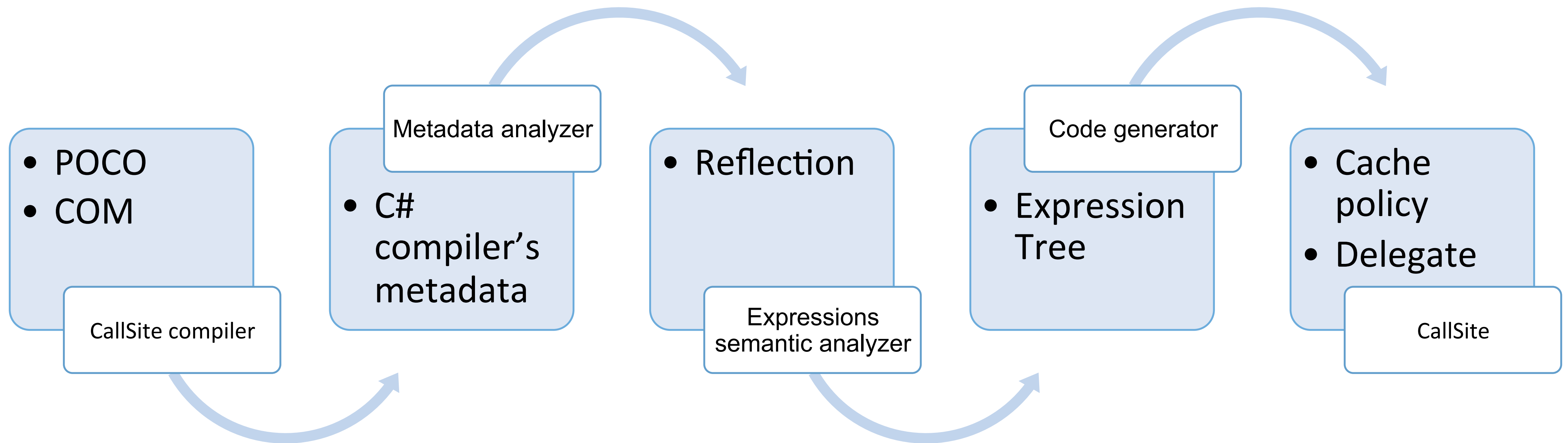
DLR Inside Out – Callsite's Life Cycle



DLR Inside Out – Callsite’s Life Cycle



DLR Inside Out – Callsite generation process



DLR Inside Out – PIC

- DLR реализует известную технику PIC, используемую в Google V8, JavaScriptCore, Self VM, Oracle HotSpot VM
- Polymorphic. Узел вызова может иметь несколько состояний, исходя из типов объектов, используемых в динамической операции
- Inline. Жизненный цикл экземпляра класса CallSite проходит именно в месте самого вызова
- Cache. Работа основана на переиспользовании сгенерированного кода благодаря многоуровневому кэшу

DLR Inside Out – Cache

Level 2

- Used by all callsites
- Adaptive cache method signature
- 64-128 callsites per operation type and signature

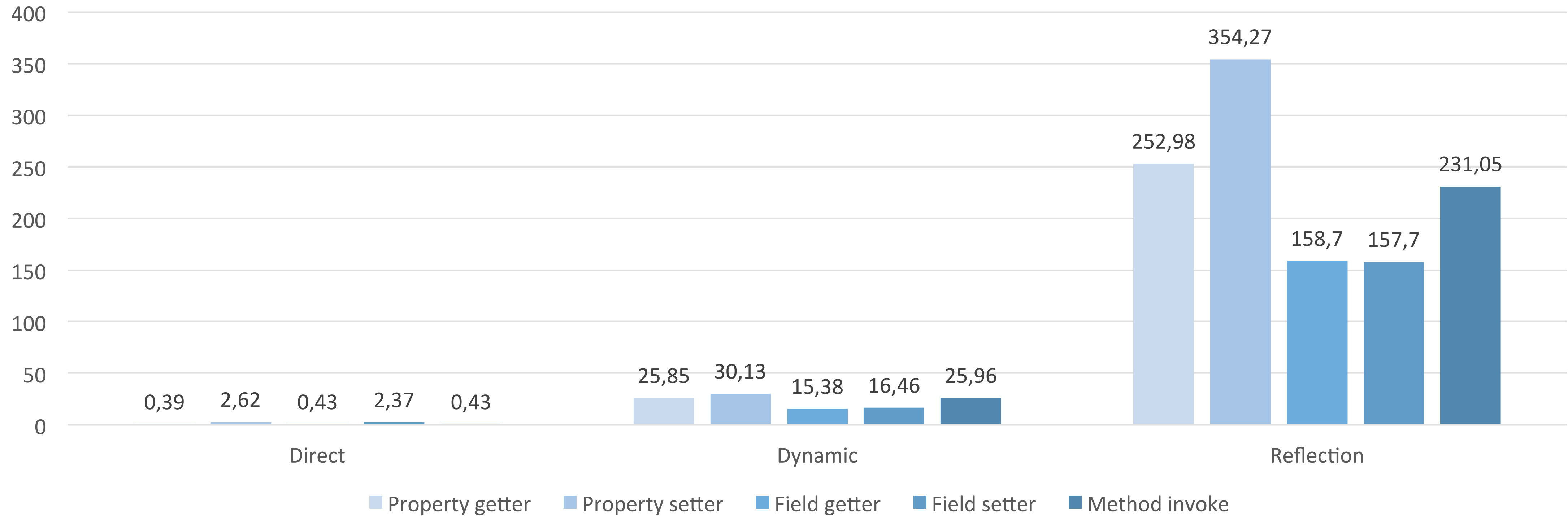
Level 1

- Method's signature-based cache
- Up to 10 cached callsites

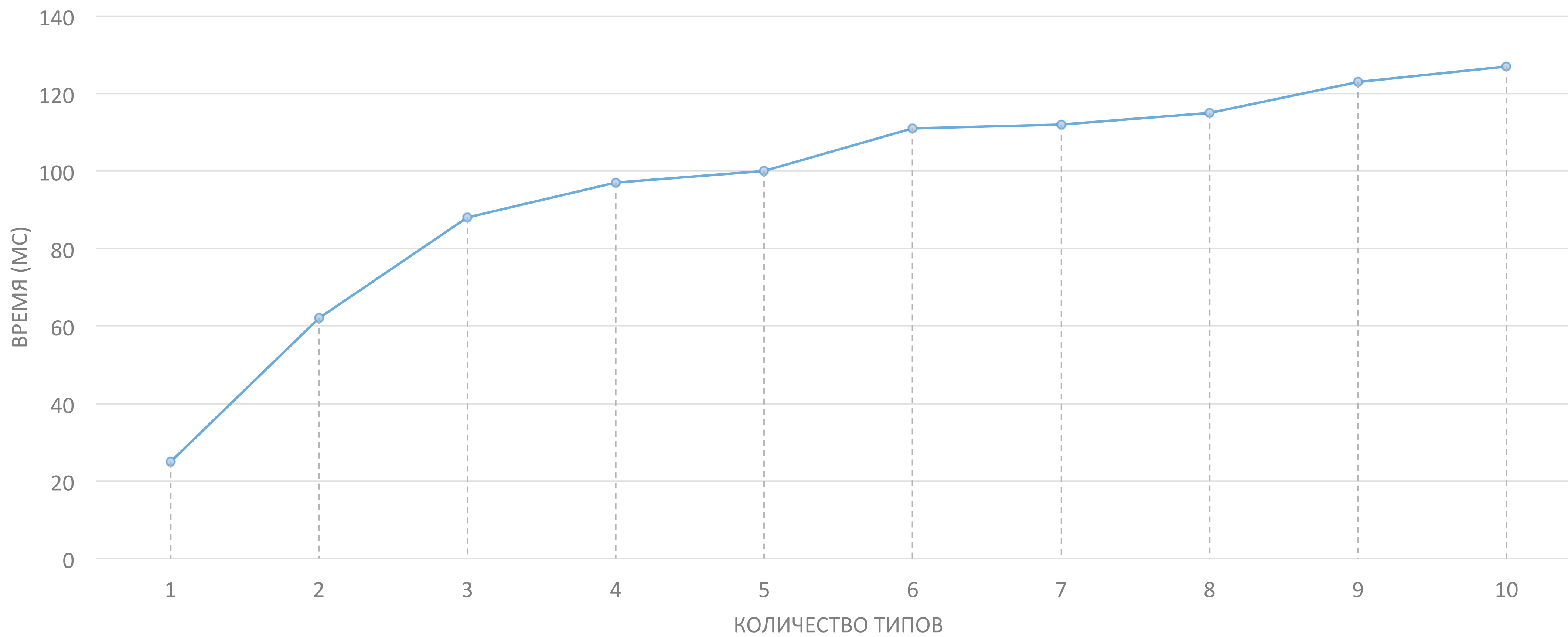
Level 0

- Context Type
- Operation type

DLR Inside Out – Monomorphic Performance



DLR Inside Out – Polymorphic Performance



DLR Inside Out – Field/Property getter

```
CallSite<Func<CallSite, object, object>>.Create(
    Binder.GetMember(CSharpBinderFlags.None,
        "PropertyName", typeof (Context),
        new CSharpArgumentInfo[]
        {
            // CSharpArgumentInfo
            // represents object instance
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null)
        })
);
```

DLR Inside Out – Field/Property setter

```
CallSite<Func<CallSite, object, object, object>>.Create(
    Binder.SetMember(CSharpBinderFlags.None,
        "PropertyName", typeof (Context),
        new CSharpArgumentInfo[]
        {
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null),
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null)
        })
    );
```

DLR Inside Out – Indexer getter

```
CallSite<Func<CallSite, object, object, object>>.Create(
    Binder.GetIndex(CSharpBinderFlags.None,
        "PropertyName", typeof (Context),
        new CSharpArgumentInfo[]
        {
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null),
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null)
        })
);
```

DLR Inside Out – Indexer setter

```
CallSite<Func<CallSite, object, object, object, object>>.Create(
    Binder.SetIndex(CSharpBinderFlags.None,
        "PropertyName", typeof (Context),
        new CSharpArgumentInfo[]
        {
            // three CSharpArgumentInfo
            // instance[index] = value
        })
    );
```

DLR Inside Out – Method invocation

```
CallSite<Action<CallSite, object...[optional args]>>.Create(
    Binder.InvokeMember(CSharpBinderFlags.ResultDiscarded,
        "MethodName", null, typeof (Context),
        new CSharpArgumentInfo[]
        {
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null),
            [optional, args] CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.None, null)
        })
);
```

DLR Inside Out – Boxing/Unboxing

- CallSite<Action<CallSite, object...[optional args]>>.Create() может принимать на “вход” как Action<…>, так и Func<…>
- Generic-параметрами Action<…> и Func<…> могут быть любые типы, которые компилятор сможет вывести еще на этапе компиляции, благодаря чему вместо, например, Action<object, object, object> мы можем получить Action<int, DateTime, string>, что позволит избежать лишней упаковки
- Если не удается вывести типы параметров операции (что весьма маловероятно), то произойдет fallback к Action<object, object, object>, например.

DLR in the wild – C# `dynamic` Usage Goals

- Использование C# `dynamic` дает нам возможность:
  - Взаимодействовать с динамическими языками
  - Упростить COM-interop
  - ...
  - Привнести новые паттерны в мир C# (~~использовать~~ заменить double dispatch на эффективный multiple dispatch)
  - Упростить иерархию классов
  - Уменьшить императивный код

DLR in the wild –Direct API Usage Goals

- “Ручное” использование DLR API дает нам возможность:
  - Заменить использование Reflection на более быстрый API
  - Перестать пользоваться Reflection emit для создания оберток и т.п., что дает нам легкоподдерживаемый и читаемый код

DLR in the wild – Example (Multiple Dispatch pattern with C# `dynamic`)

- Заменим Exception Handling Block библиотеки Enterprise Library на собственный с помощью C# `dynamic` и паттерна multiple dispatch
- Основными примитивами для обработки исключений в Enterprise Library являются:
  - ExceptionManager
  - ExceptionPolicyDefinition
  - ExceptionPolicyEntry
  - IExceptionHandler

Microsoft.Practices.EnterpriseLibrary.ExceptionHandling
- { } Microsoft.Practices.EnterpriseLibrary.Common.Configuration
- { } Microsoft.Practices.EnterpriseLibrary.Common.Configuration
- { } Microsoft.Practices.EnterpriseLibrary.ExceptionHandling
  - ExceptionFormatter
  - ExceptionHandlingException
  - ExceptionManager
  - ExceptionPolicy
  - ExceptionPolicyDefinition
  - ExceptionPolicyEntry
  - ExceptionPolicyFactory
  - ExceptionUtility
  - IExceptionHandler
  - PostHandlingAction
  - ReplaceHandler
  - TextExceptionFormatter
  - WrapHandler
  - XmlExceptionFormatter
- { } Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Conf
- { } Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Prop

Enterprise Library – Pros/Cons

- Pros:
  - Enterprise Library предоставляет приложениям любого типа функционал организации единой схемы обработки исключений. (*Примечание*: Написание собственного pipeline а-ля ASP.NET & co. требует много усилий и затрат, что не всегда оправдано)
- Cons:
  - Настройка блока обработки исключений требует относительно большого количества кода для setup
  - Количество policy, handler и т.п. сущностей растет пропорционально количеству используемых типов исключений

Enterprise Library – Typical code for exception handling

```
var policies = new List<ExceptionPolicyDefinition>();
var myTestExceptionPolicy = new List<ExceptionPolicyEntry>
{
    {
        new ExceptionPolicyEntry(typeof (InvalidCastException), PostHandlingAction.NotifyRethrow,
            new IExceptionHandler[] {new LoggingExceptionHandler(...),})
    },
    {
        new ExceptionPolicyEntry(typeof (Exception), PostHandlingAction.NotifyRethrow,
            new IExceptionHandler[] {new ReplaceHandler(...)})
    }
};
policies.Add(new ExceptionPolicyDefinition("MyTestExceptionPolicy", myTestExceptionPolicy));
ExceptionManager manager = new ExceptionManager(policies);
try
{
    // code to throw exception
}
catch (Exception e)
{
    manager.HandleException(e, "Exception Policy Name");
}
```

Really??

Enterprise Library - IExceptionHandler

```
namespace Microsoft.Practices.EnterpriseLibrary.ExceptionHandling
{
    public interface IExceptionHandler
    {
        Exception HandleException(Exception ex,
                                  Guid handlingInstanceID);
    }
}
```

Enterprise Library - ExceptionPolicyDefinition

```
private ExceptionPolicyEntry FindExceptionPolicyEntry(Type exceptionType)
{
    ExceptionPolicyEntry policyEntry = null;
    while (exceptionType != typeof(object))
    {
        policyEntry = this.GetPolicyEntry(exceptionType);
        if (policyEntry != null)
        {
            return policyEntry;
        }
        exceptionType = exceptionType.BaseType;
    }
    return policyEntry;
}
```

Enterprise Library - ExceptionPolicyEntry

```
public bool Handle(Exception exceptionToHandle)
{
    if (exceptionToHandle == null)
    {
        throw new ArgumentNullException("exceptionToHandle");
    }
    Guid handlingInstanceID = Guid.NewGuid();
    Exception chainException = this.ExecuteHandlerChain(exceptionToHandle,
handlingInstanceID);
    return this.RethrowRecommended(chainException, exceptionToHandle);
}
```

Enterprise Library - ExceptionPolicyEntry

```
private Exception ExecuteHandlerChain(Exception ex, Guid handlingInstanceID)
{
    string name = string.Empty;
    try
    {
        foreach (IExceptionHandler handler in this.handlers)
        {
            name = handler.GetType().Name;
            ex = handler.HandleException(ex, handlingInstanceID);
        }
    }
    catch (Exception exception)
    {
        // rest of implementation
    }
    return ex;
}
```

DLR in the wild – Own Custom Exception handler interfaces

```
public interface IExceptionHandler
{
    void HandleException<T>(T exception) where T : Exception;
}

public interface IExceptionHandler<T> where T : Exception
{
    void Handle(T exception);
}
```

DLR in the wild – Own Custom Exception handler implementation

```
public class DefaultExceptionHandler : IExceptionHandler, IExceptionHandler<Exception>
{
    public void HandleException<T>(T exception) where T : Exception
    {
        var handler = this as IExceptionHandler<T>;

        if (handler != null)
            handler.Handle(exception);
        else
            this.Handle(exception as dynamic);
    }

    public void Handle(Exception exception)
    {
        OnFallback(exception);
    }

    protected virtual void OnFallback(Exception exception)
    {
        // rest of implementation
    }
}
```

```
public class FileSystemExceptionHandler :
             DefaultExceptionHandler,

             IExceptionHandler<IOException>,

             IExceptionHandler<FileNotFoundException>
{
    public void Handle(IOException exception)
    {
        // rest of implementation
    }
}
```

DLR in the wild – Custom Exception handler usage

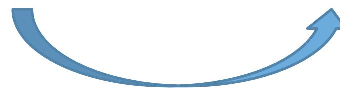
```
IExceptionHandler defaultHandler = new FileSystemExceptionHandler();
defaultHandler.HandleException(new IOException()); // Handle(IOException) overload
defaultHandler.HandleException(new FileNotFoundException()); // Handle(IOException) overload
defaultHandler.HandleException(new FormatException()); // Handle(Exception) => OnFallback
```

DLR in the wild – Example (Reflection method invoke vs CallSite API)

- Сравним возможности вызова метода с помощью Reflection и с помощью CallSite API
  - static-методов
  - instance-методов
  - Создание делегатов на методы
  - Применим частичное применение функции (без тавтологии никак ☺)

DLR in the wild – Delegate using Reflection (Static methods)

```
public static T CreateStaticMethodDelegate<T>(string methodName) where T : class
{
    MethodInfo targetMethodInfo = typeof(Calc).GetMethod(methodName,
        BindingFlags.Static | BindingFlags.Public,
        null,
        GetTypeParameters(typeof(T)),
        null);

    T target = (T) (object) Delegate.CreateDelegate(typeof(T), targetMethodInfo);

    return target;
}
```



```
private static Type[] GetTypeParameters(Type delegateType)
{
    return delegateType.GetMethod("Invoke")
                       .GetParameters()
                       .Select(x => x.ParameterType)
                       .ToArray();
}
```

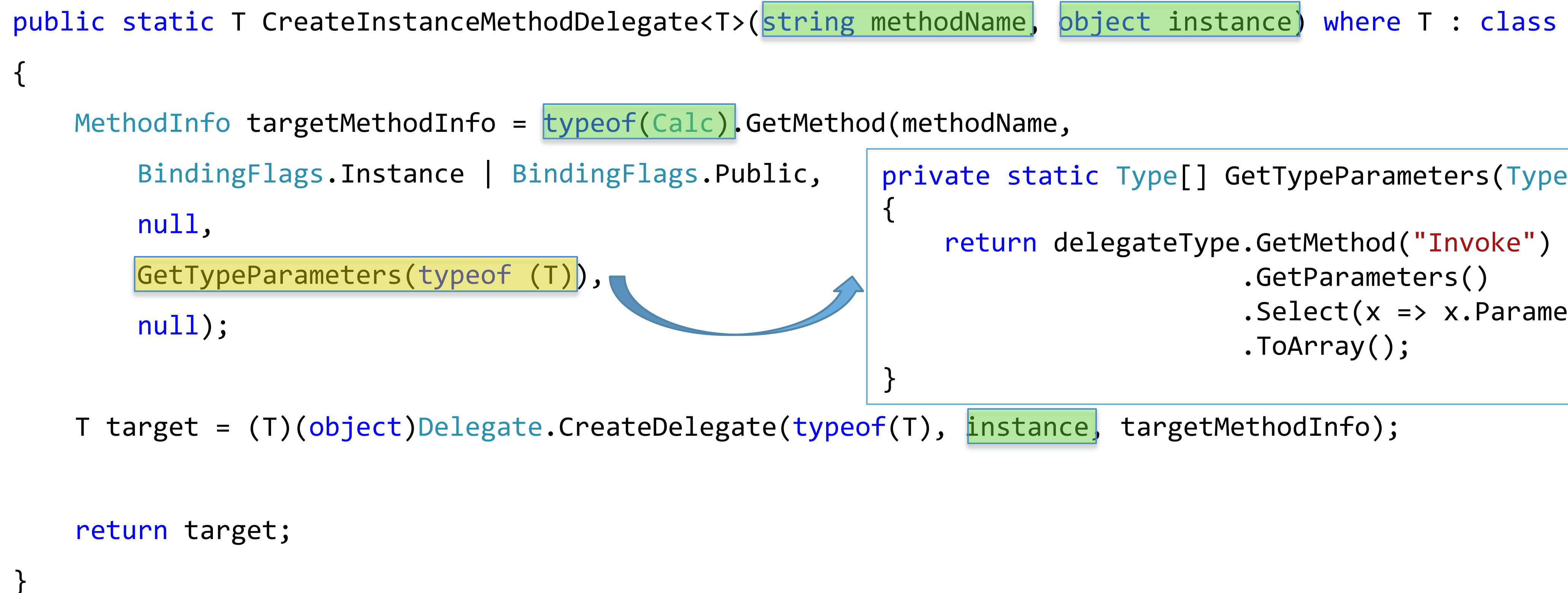
DLR in the wild – CallSite Delegate (Static methods)

```
public static CallSite<T> CreateDynamicStaticMethodDelegate<T>(string methodName) where T : class
{
    return CallSite<T>.Create(Binder.InvokeMember(CSharpBinderFlags.None,
        methodName,
        null,
        typeof(Program),
        GetCSharpArgumentInfos(typeof(T), true)));
}
```

DLR in the wild – Reflection vs. CallSite Usage (Static methods)

```
class Program
{
    static void Main(string[] args)
    {
        Func<int, int, int> addDelegate = CreateStaticMethodDelegate<Func<int, int, int>>("Sum");
        Console.WriteLine("Static delegate: {0}", addDelegate(2, 2));

        var addCallSite = CreateDynamicStaticMethodDelegate<Func<CallSite, object, int, int, object>>("Sum");
        Console.WriteLine("Static callsite: {0}", addCallSite.Target(addCallSite, typeof(Calc), 4, 4));

        addDelegate = (a, b) => (int) addCallSite.Target(addCallSite, typeof (Calc), a, b);

        Console.WriteLine("Partial applied delegate with callsite: {0}", addDelegate(5, 5));

        // Static delegate: 4
        // Static callsite: 8
        // Partial applied delegate with callsite: 10
    }
}
```

DLR in the wild – Delegate using Reflection (Instance methods)



DLR in the wild – CallSite Delegate (Instance methods)

```
public static CallSite<T> CreateDynamicInstanceMethodDelegate<T>(string methodName) where T : class
{
    return CallSite<T>.Create(Binder.InvokeMember(CSharpBinderFlags.None,
        methodName,
        null,
        typeof (Program),
        GetCSharpArgumentInfos(typeof (T), false)));
}
```

DLR in the wild – Reflection vs. CallSite Usage (Instance methods)

```
class Program
{
    static void Main(string[] args)
    {
        var calc = new Calc();

        Func<int, int, int> addDelegate = CreateInstanceMethodDelegate<Func<int, int, int>>("Add", calc);
        Console.WriteLine("Delegate: {0}", addDelegate(1, 1));

        var addCallSite = CreateDynamicInstanceMethodDelegate<Func<CallSite, object, int, int, object>>("Add");
        Console.WriteLine("CallSite: {0}", addCallSite.Target(addCallSite, calc, 3, 3));

        // OR
        MethodInfo addMethodInfo = typeof (Calc).GetMethod("Add", BindingFlags.Instance | BindingFlags.Public,
                                                          null, new Type[] {typeof (int), typeof (int)}, null);

        Console.WriteLine("Reflection: {0}", addMethodInfo.Invoke(calc, new object[] {1, 1}));

        // Delegate: 2
        // CallSite: 6
        // Reflection: 2
    }
}
```

DLR in the wild – Reflection vs. CallSite

- Создание делегатов через Reflection для instance-методов неэффективно, т.к. “захватывает” экземпляр объекта
- Reflection и CallSite предоставляют схожие возможности для делегатов статичных методов

Infinite loop

```
public class LambdaInstance<T> : LambdaInstance<T, T>
{
}

public class LambdaInstance<T, TPluginType>
                              : ExpressedInstance<LambdaInstance<T, TPluginType>, T, TPluginType>
{
}

public abstract class ExpressedInstance<T>
{
}

public abstract class ExpressedInstance<T, TReturned, TPluginType> : ExpressedInstance<T>
{
}
```

Infinite loop

```
class Program
{
    private static LambdaInstance<object> ShouldThrowException(object argument)
    {
        throw new NotImplementedException();
    }

    static void Main(string[] args)
    {
        // will be an exception thrown?
        ShouldThrowException((dynamic)new object());
    }
}
```

Infinite loop

```
class Program
{
    private static LambdaInstance<object> ShouldThrowException(object argument)
    {
        throw new NotImplementedException();
    }

    static void Main(string[] args)
    {
        // will never end
        ShouldThrowException((dynamic)new object());
    }
}
```

Infinite loop

```
public class LambdaInstance<T> : LambdaInstance<T, T>
{
}

public class LambdaInstance<T, TPluginType>
                              : ExpressedInstance<LambdaInstance<T, TPluginType>, T, TPluginType>
{
}

public abstract class ExpressedInstance<T>
{
}

public abstract class ExpressedInstance<T, TReturned, TPluginType> : ExpressedInstance<T>
{
}
```

Infinite loop

```
public class LambdaInstance<T> : LambdaInstance<T, T>
{
}

public class LambdaInstance<T, TPluginType>
                    : ExpressedInstance<LambdaInstance<T, TPluginType>, T, TPluginType>
{
}

public abstract class ExpressedInstance<T>
{
}

public abstract class ExpressedInstance<T, TReturned, TPluginType> : ExpressedInstance<T>
{
}
```

DLR - Summary

- ✓ Динамизм для .NET
- ✓ Эффективный API - быстрее чем Reflection
  - ✓ Приемлемая и достижимая производительность
  - ✓ Без затрат на инфраструктуру
  - ✓ Легкоподдерживаемый код
- ✓ Новые паттерны для C#
  - ✓ Mutiple dispatch (“скажем пока” double dispatch, boilerplate-коду и большой иерархии классов)
- ✓ It’s cool ☺

Спасибо!

Q&A

Контакты

- Email: szkarlen@gmail.com, KSimonyan@luxoft.com
- Twitter, Habr, GitHub: @szKarlen