DOTNEXT

Extension Members:
новый уровень проектирования .NET API или новый уровень сложности?

Виктор Дзицкий
Руководитель группы разработки
Соларлаб

Вопрос в зал

Вопрос в зал

Кто активно использует extension-методы?

Главный тезис

тип + видимые расширения + правила поиска =
API, который выглядит как часть типа

Extension Members не меняют сам тип,
они расширяют API в текущем контексте.

В чём смысл?

Extension Members – это не просто развитие extension methods.

Это шаг к более богатой модели API:
код выглядит как работа с членами типа,
но разрешается через поиск на этапе компиляции.

О чём поговорим

- Как появились extension methods
- Что привнесли extension members
- Что происходит под капотом
- Где могут ждать неприятности
- Куда движутся extensions
- Практические выводы

Сквозной пример

Задача:
Подключить новый платежный шлюз *MyAwesomePay*.

Внутренняя модель платежей уже существует:
её используют API, отчётность, антифрод, сверки и аудит.

Нужно добавить новое поведение для конкретного шлюза рядом с интеграционным слоем, не раздувая домен.

Исходная модель

```
public readonly record struct Money(
    decimal Amount,
    string Currency
);
```

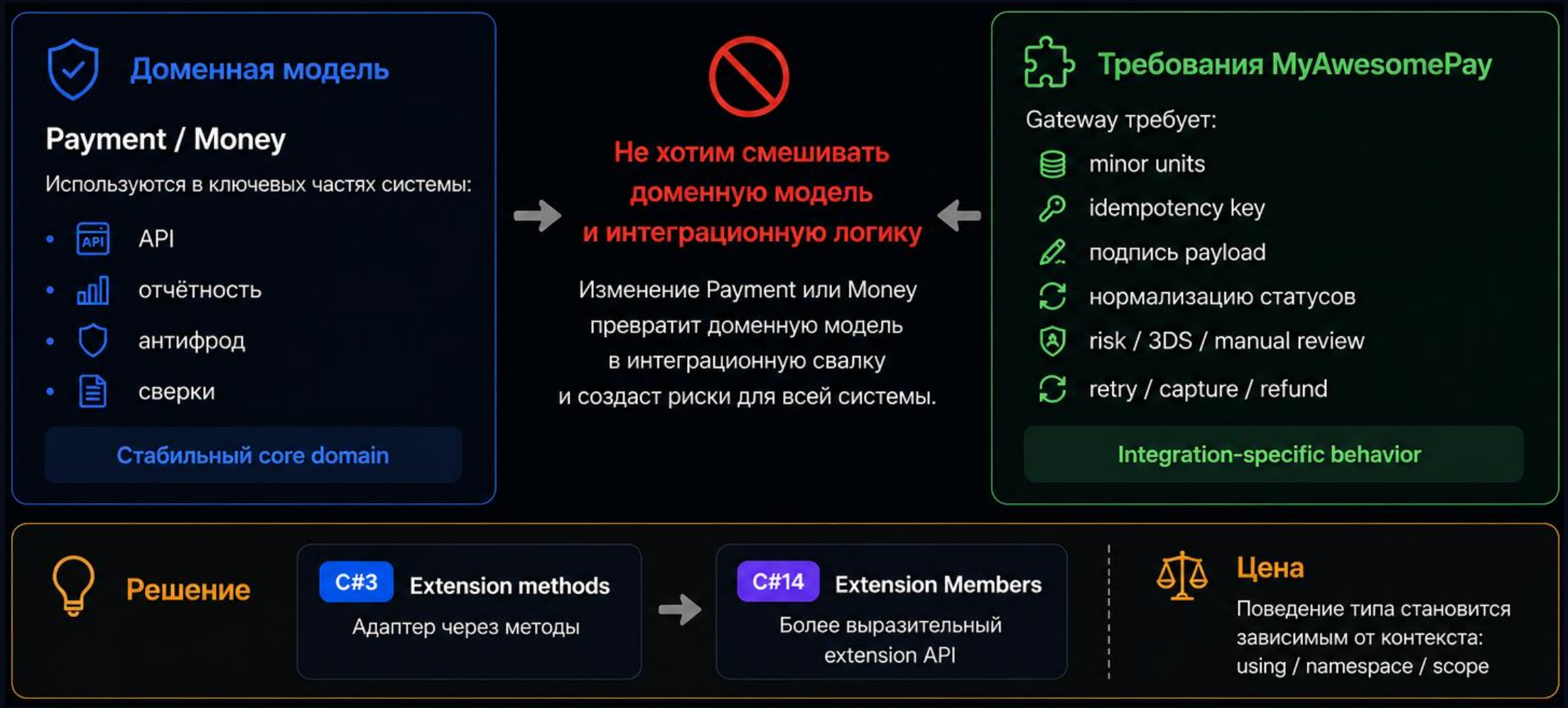
```
public sealed record Payment(
    Guid Id,
    string CustomerId,
    Money Amount,
    PaymentStatus Status,
    string ExternalOrderId,
    DateTimeOffset CreatedAt
);
```

Что делать с требованиями?

Требование	Что за этим скрывается
MinorUnits	денежная семантика и валютные правила
IdempotencyKey	защита от двойного списания
NormalizeStatus	маппинг внешних статусов
Requires3DS	требования платёжных систем
CanRetryCapture	правила повторного проведения операций
SignPayload	безопасность и проверка подлинности запросов

Логику, которая начинает выглядеть как «обычные свойства и методы»

Почему нельзя просто менять домен?



Эволюция расширений

Зачем extension methods появились в C#?
Почему старого подхода стало недостаточно?

Контекст появления

C# 3: LINQ, лямбда-выражения, деревья выражений, синтаксис запросов

Методы расширения позволили LINQ расширять существующие типы без изменения их исходного кода.

LINQ как главный драйвер развития

```
var captured = payments
    .Where(p => p.Status == PaymentStatus.Captured)
    .Select(p => p.Amount)
    .ToList();
```

Форма extension method

```
public static class PaymentGatewayExtensions
{
    public static MyAwesomePayRequest ToMyAwesomePayRequest(
        this Payment payment,
        string merchantId) => ...;
}
```

Минимальная спецификация: static class + static method + this receiver

Вызов выглядел как instance method

```
var request = payment.ToMyAwesomePayRequest(merchantId);

Console.WriteLine(payment.Amount.ToDisplayAmount());
```

Классический extension method

```
public static class MoneyExtensionMethods
{
    public static long ToMinorUnits(this Money money) =>
        decimal.ToInt64(
            decimal.Round(money.Amount * 100m, 0));
}
```

Что дает такая модель?

Польза	Ограничение	Риск
Money и **Payment** адаптированы под требования шлюза	доступны только методы: *ToMinorUnits*, *CanBeCaptured*, *ToRequest*	бизнес-логика может уехать во внешний слой и стать менее заметной

Границы модели

```
// вызов как метода экземпляра
payment.ToMyAwesomePayRequest(merchantId);

// фактически это обычный статический метод
PaymentGatewayExtensions
        .ToMyAwesomePayRequest(payment, merchantId);
```

extension-style вызов всегда можно переписать как *static call*

Почему methods-only стало не хватать?

Не всё удобно выражать методами:

- Действие → метод
- Признак → свойство
- Фабрика → статический член
- Операция → оператор

Почему methods-only стало не хватать?

Fluent API – вызовы строятся цепочкой и читаются почти как естественный язык, но модель заставляла писать:

```
money.IsPositive()
```

вместо

```
money.IsPositive
```

```
MoneyFactory.Usd(100m)
```

вместо

```
Money.Usd(100m)
```

Почему methods-only стало не хватать?

DSL (Domain Specific Language) – API, который выглядит как язык предметной области и читается декларативно:

```
payment.Requires3DS
```

Но methods-only ломает «язык предметной области»:

```
PaymentGatewayExtensions.Requires3DS(payment)
```

Почему methods-only стало не хватать?

Для конфигурации, которая читается как декларативное описание системы:

```
services.AddPayments(options =>
{
    options.RetryPolicy = RetryPolicy.Exponential;
    options.Require3DS = true;
});
```

Pattern matching хочет работать со свойствами, а не с методами:

```
if (payment is { Requires3DS: true }) { RedirectToChallenge(); }
```

Почему methods-only стало не хватать?

Для доменных типов и объектов-значений особенно важна форма API:

```
MoneyOperations.Add(left, right);
MoneyFactory.Usd(100m);
```

естественные операции и фабрики делают код ближе к языку предметной области:

```
left + right;
Money.Usd(100m);
money.MinorUnits
```

Новые возможности

Extension blocks и новый уровень выразительности

C# 14 / .NET 10

Extension Members – одна из самых заметных языковых новинок C# 14.

Теперь, кроме методов, можно добавлять свойства, статические члены и операторы.

Extension block

```
public static class MoneyExtensionMembers
{
    extension(Money money)
    {
        // instance extension members
    }

    extension(Money)
    {
        // static extension members
    }
}
```

Extension property

```
extension(Money money)
{
    public long MinorUnits =>
        decimal.ToInt64(
            decimal.Round(money.Amount * 100m, 0));

    public string DisplayAmount =>
        $"{money.Amount:0.00} {money.Currency}";
}
```

Extension property: MinorUnits

```
extension(Money money)
{
    public long MinorUnits {
        get {
                var scale = money.Currency switch
                {
                    "JPY" => 0,
                    "KWD" => 3,
                    _ => 2
                };
            return (long)(money.Amount * (decimal)Math.Pow(10, scale));
            }
        }
}
```

Применение свойств

```
if (payment.Amount.IsPositive)
{
    var minorUnits = payment.Amount.MinorUnits;
    SendToGateway(minorUnits);
}
```

вычисляемые свойства полезны,
пока они не скрывают критичную бизнес-логику

Pattern matching и extension properties

```
extension(Payment payment)
{
    public bool RequiresManualReview =>
        payment.Risk.RequiresManualReview;
}

if (payment is { RequiresManualReview: true })
{
    SendToManualReview(payment);
}
```

Extension property

Польза	Ограничение	Риск
читаемый код, удобно для DSL и сценариев обработки	свойство не является частью **Payment** и зависит от *using*	источник правила не виден в месте вызова

Static extension members

```
extension(Money)
{
    public static Money Usd(decimal amount) =>
        new(amount, "USD");

    public static Money FromMinorUnits(
        long minorUnits,
        string currency) =>
        new(minorUnits / 100m, currency);
}
```

Применение статических членов

```
var amount = Money.Usd(125.50m);
var fee = Money.Usd(1.25m);

var request = Payment.Pending(
    customerId,
    amount + fee,
    externalOrderId);
```

Операторы расширения

Операторы – это уже не просто новый способ добавить внешний член к типу.

Они позволяют менять сам стиль кода:

делать C# похожим на локальный предметный язык или даже на другой язык программирования.

Extension Operators

```
extension(Money)
{
    public static Money operator +(Money left, Money right)
    {
        left.EnsureSameCurrencyAs(right);
        return new Money(left.Amount + right.Amount, left.Currency);
    }
}

var total = payment.Amount + fee;
```

Статическая реализация

```
extension(Money)
{
    public static Money operator +(Money amount, GatewayFee fee) =>
        MoneyCalculator.Add(
            amount,
            fee.Amount,
            fee.Policy);
}

var total = payment.Amount + gatewayFee;

// Можно представить как
var total = MyAwesomePayOperators.op_Addition(payment.Amount, gatewayFee);
```

Составное присваивание

```
extension(PaymentBatchDraft draft)
{
    public void operator +=(Payment payment) => draft.Add(payment);

    public void operator +=(IEnumerable<Payment> payments) =>
        draft.AddRange(payments);
}

var draft = new PaymentBatchDraft();
draft += payment;
draft += retryQueue;
```

+= как локальный DSL

Операторы как DSL

```
extension(object)
{
    public static string operator ~(object payload) =>
        JsonSerializer.Serialize(payload);
}

var body = ~new
{
    payment.Id,
    amount = payment.Amount.MinorUnits,
    payment.Amount.Currency,
    signature = payment.SignPayload(secret)
};
```

Операторы как DSL

На примере MyAwesomePay можно представить компактный синтаксис для сборки тела запроса.

Можно реализовать:

- массив * число
- List<T> += item
- ~object → JSON
- writer << message

Операторы как DSL

На примере MyAwesomePay можно представить компактный синтаксис для сборки тела запроса.

Можно реализовать:

- массив * число
- List<T> += item
- ~object → JSON
- writer << message

Это мощный DSL-инструмент, но только если команда понимает локальные правила

Утиная типизация

Точки входа паттерна: foreach → GetEnumerator()

```
extension(MyAwesomePayBatch batch)
{
    public MyAwesomePayEnumerator GetEnumerator()
        => new(batch.Payments);
}

foreach (var payment in batch) { ... }
```

Утиная типизация

Точки входа паттерна: await → GetAwaiter()

```
extension(Payment payment)
{
    public GatewayAwaiter GetAwaiter()
        => gateway.SendAsync(payment).GetAwaiter();
}

await payment;
```

Утиная типизация

Extension Members не меняют систему типов и не делают тип реализацией нового интерфейса.

Но они могут влиять на доступность языковых конструкций в конкретном контексте компиляции.

Это не адаптация через интерфейс:

- batch не стал IEnumerable<Payment>
- payment не стал Task

Что изменилось в C# 14?

C# 14 дал мощный инструмент для выразительного API.

Но чем естественнее выглядит вызов, тем сложнее понять,
кто владеет поведением.

Что происходит под капотом

Исходный синтаксис, поиск, метаданные, рефлексия и Roslyn

Что пишет разработчик

```
public static class MoneyExtensions
{
    extension(Money money)
    {
        public long MinorUnits => ToMinorUnits(money);
    }
}

var minor = payment.Amount.MinorUnits;
```

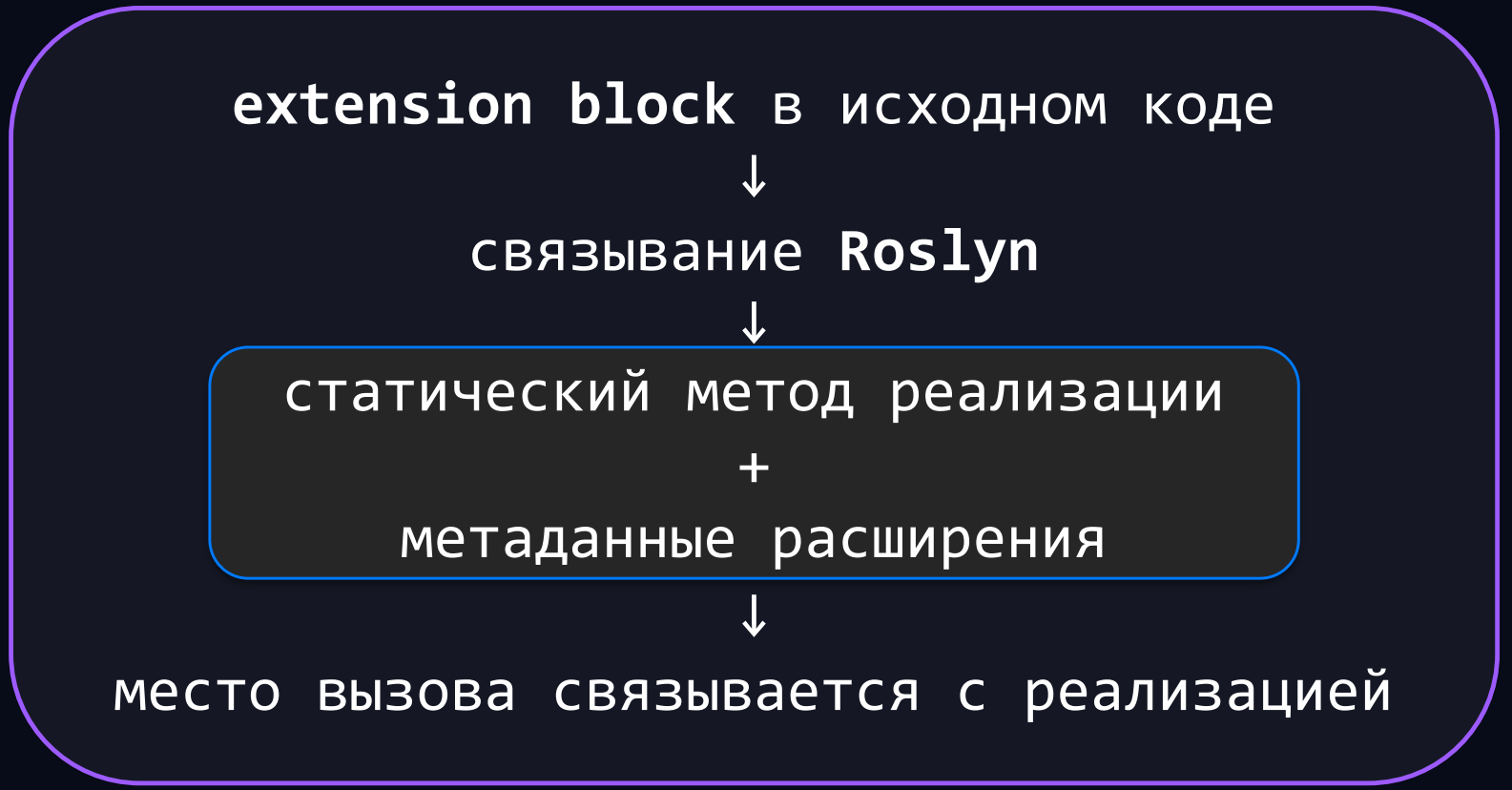
Как это нужно понимать

```
var minor = payment.Amount.MinorUnits;

// Можно представить как
var minor = MoneyExtensions.GetMinorUnits(payment.Amount);
```

- Money не получает новый настоящий член
- Компилятор находит вызов среди видимых расширений
- Реализация остаётся во внешнем статическом классе
- Вызов выглядит как API типа, но принадлежит внешнему коду

Что попадает в сборку



Упрощённый пайплайн Roslyn

Parse	→	синтаксическое дерево
Bind	→	объект вызова + видимые расширения
Resolve	→	разрешение перегрузок
Emit	→	статическая реализация + метаданные

ExtensionMarkerNameAttribute

```
extension(Money money)
{
    public long MinorUnits => ...;
}

// кодирование компилятора
[ExtensionMarkerName("...")]
// служебный член

// статический метод реализации
```

Маркер метаданных для компилятора и инструментов (связывает служебный член и реализацию)

Что покажет рефлексия

```
typeof(Money)
    .GetMembers()
    .Any(m => m.Name == "MinorUnits");

// false: это не член исходного типа

typeof(MoneyExtensions)
    .GetMembers(...);

// реализация / метаданные
```

Что покажет IDE

Visual Studio / Roslyn

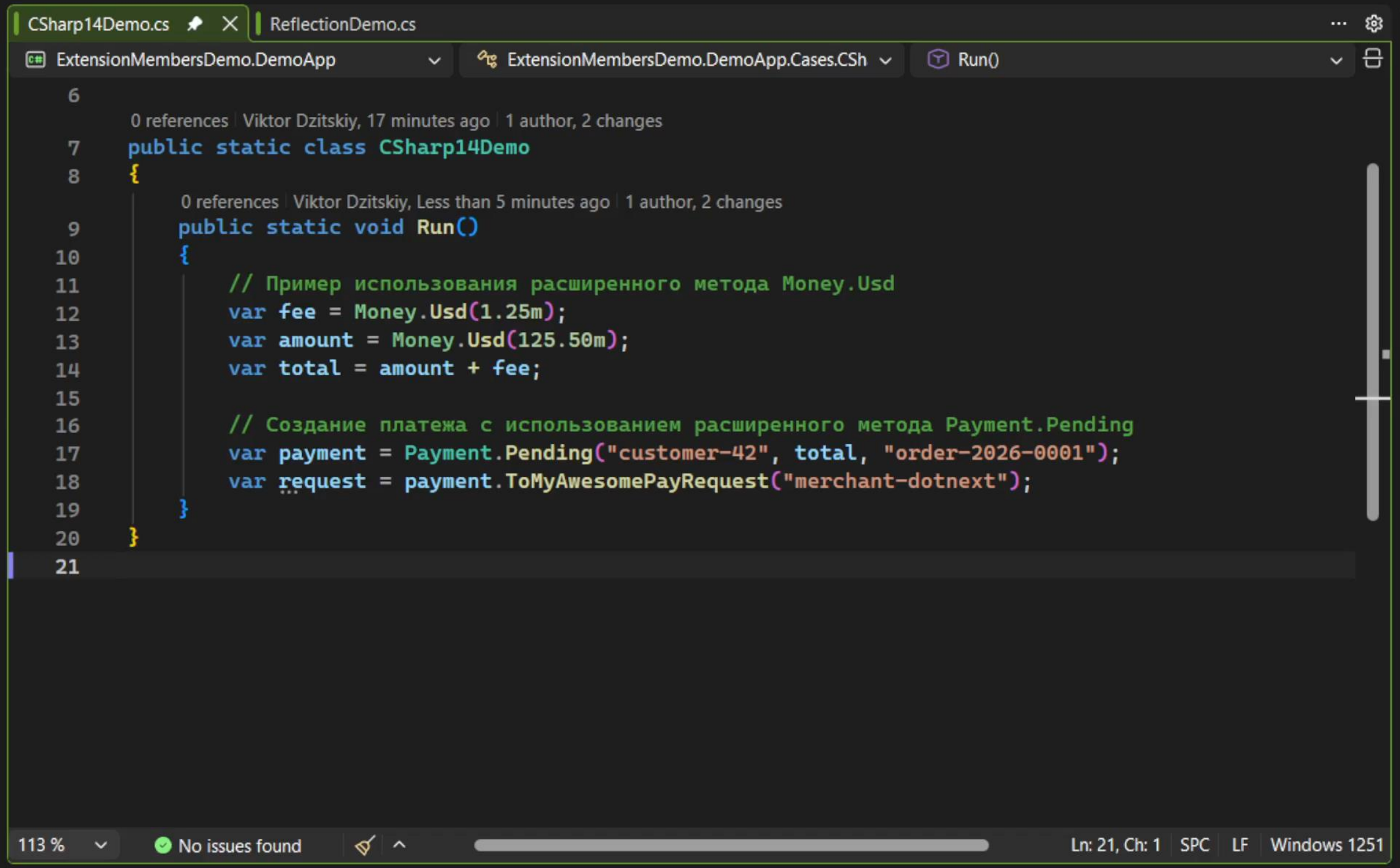
- синтаксис и подсказки зависят от версии SDK и IDE
- окончательное решение принимает компилятор

Rider / ReSharper / VS Code

- поддержка extension members и операторов расширения зависит от версии
- отдельно проверяем навигацию, автодополнение и рефакторинг

Рассмотрим пример

 Dzitskiy/ExtensionMembersDemo


CSharp14Demo.cs
ReflectionDemo.cs
ExtensionMembersDemo.DemoApp
ExtensionMembersDemo.DemoApp.Cases.CSh
Run()
0 references | Viktor Dzitskiy, 17 minutes ago | 1 author, 2 changes
public static class CSharp14Demo
{
0 references | Viktor Dzitskiy, Less than 5 minutes ago | 1 author, 2 changes
public static void Run()
{
// Пример использования расширенного метода Money.Usd
var fee = Money.Usd(1.25m);
var amount = Money.Usd(125.50m);
var total = amount + fee;
// Создание платежа с использованием расширенного метода Payment.Pending
var payment = Payment.Pending("customer-42", total, "order-2026-0001");
var request = payment.ToMyAwesomePayRequest("merchant-dotnext");
}
}
113 %
No issues found
Ln: 21, Ch: 1
SPC
LF
Windows 1251

Чем Extension Members не являются

Extension Members – это не:

- реализация интерфейса
- *virtual / override*
- динамическая диспетчеризация
- привилегированный доступ к закрытому состоянию
- настоящие члены исходного типа

Можно ли было сделать иначе?

Можно ли было сделать иначе?

C# Extension Members
вызов как член типа + поиск на этапе компиляции → тип не меняется

Kotlin extensions
статически разрешаемые расширения → ближайшая модель

Rust traits
поведение участвует в системе типов → контракты и ограничения

Go interfaces
интерфейс реализуется неявно → по набору методов

Extension Members – это не...

- **Not C++ friend** – нет привилегированного доступа к private/protected
- **Not traits / type classes** – нет нового контракта и generic-ограничений
- **Not mixins** – не подмешивают состояние и настоящие члены
- **Not monkey patching** – нет изменения типов во время выполнения

Ближайшие по ощущению идеи: Kotlin extensions, Swift extensions

Выглядит как член типа. Компилируется как расширение. Принадлежит контексту.

Практические сценарии

Что может пойти не так?

Реальные практические кейсы

Конкретные сценарии отказов для SDK и production-кода:

- Обновление зависимости → новый набор кандидатов
- *using / global using* → другой набор видимых расширений
- Разрешение перегрузок → другой выбор или неоднозначность
- Повторная компиляция → возможна смена поведения
- Операторы → критичные правила могут быть скрыты
- Во время выполнения расширений нет → последствия есть, но динамического поиска

Кейс 1: пакет принёс новый extension

Библиотека A

```
namespace Finance.Extensions;

public static class FinanceMoneyExtensions
{
    extension(Money money)
    {
        public string GatewayAmount() =>
      $"{money.Amount:0.00} {money.Currency}";
    }
}
```

Код клиента

```
using Finance.Extensions;

var amount = Money.Usd(125.50m);

Console.WriteLine(
    amount.GatewayAmount());
```

Money не изменился. Но у компилятора появился новый кандидат.

Кейс 1: пакет принёс новый extension

Библиотека B

```
namespace Reporting.Extensions;

public static class ReportingMoneyExtensions
{
    extension(Money money)
    {
        public string GatewayAmount() =>
       $"{money.MinorUnits}:{money.Currency}";
    }
}
```

Код клиента

```
using Finance.Extensions;
using Reporting.Extensions;

var amount = Money.Usd(125.50m);

// CS0121: call is ambiguous
Console.WriteLine(
    amount.GatewayAmount());
```

Конфликт приходит из контекста, а не из самого Money.

Кейс 1: пакет принёс новый extension

Подключение библиотеки может сломать сборку.

Не из-за изменения Money.
Не из-за ошибки в Payment.

А из-за нового extension member, появившегося в видимой области.

Кейс 2: меняется выбор перегрузки

Широкое расширение

```
extension(object value)
{
    public string ToGatewayLogLine() =>
        $"[object] {value}";
}

extension(Payment payment)
{
    public string ToGatewayLogLine() =>
        $"[payment] {payment.Id:N}";
}
```

Код клиента

```
// v1: только object-extension
Payment payment =
PaymentSamples.PaymentUsd();

Console.WriteLine(
    payment.ToGatewayLogLine()
);

// [object] Payment { ... }
```

Кейс 2: меняется выбор перегрузки

Специфичное расширение

```
// v2: добавили более специфичный
extension
extension(Payment payment)
{
    public string ToGatewayLogLine() =>
        $"[payment] {payment.Id:N};
{payment.Amount}";
}
```

Код клиента

```
// v1: только object-extension
Payment payment =
PaymentSamples.PaymentUsd();

Console.WriteLine(
    payment.ToGatewayLogLine())
;

// [object] Payment { ... }
```

Код не изменился, но после пересборки компилятор может выбрать более специфичную перегрузку.

Кейс 2: меняется выбор перегрузки

Код не изменился.
Изменился набор видимых расширений.

После пересборки компилятор может выбрать другую перегрузку.

Это уже вопрос семантической совместимости, а не только бинарной.

Кейс 3: Семантическая совместимость

Совместимость поведения

```
// Library v2
extension(Payment payment)
{
    public string
ToGatewayLogLine() =>
        $"[payment]
{payment.Id:N}; {payment.Amount}";
}
```

Код может компилироваться, но начать означать другое

```
// тот же клиентский код
Payment payment =
PaymentSamples.PaymentUsd();

Console.WriteLine(
    payment.ToGatewayLogLine());

// [payment] b4c441bb...; 125.50
USD
```

Binary compatibility ≠ Source compatibility ≠ Behavior compatibility

Кейс 3: Семантическая совместимость

Бинарная совместимость, совместимость исходного кода и совместимость поведения – это разные вещи.

Extension Members делают эту разницу заметнее, потому что API выглядит как часть типа, но приходит из контекста и версии библиотеки.

Кейс 4: Оператор и бизнес-правила

```
var total = amount + fee;

// Выглядит как безопасная доменная арифметика.
// Но оператор подключён извне через using.
```

Для сложных доменных правил лучше использовать явный API:

```
var total = MoneyCalculator.Add(amount, fee, CurrencyPolicy.Strict);

var total = amount.AddFee(fee);
```

Кейс 5: Global using

```
// GlobalUsings.cs
global using Finance.Extensions;
global using Reporting.Extensions;

// Любой файл проекта получает оба набора
// расширений и потенциальную неопределенность.
```

Конфликт может появиться не в файле, который вы редактируете. Он может прийти из GlobalUsings.cs, общего проекта, SDK или инфраструктурного пакета.

Кейс 6: расширения не ищутся во время выполнения

```
var signature = payment.SignPayload(secret);
await gateway.SendAsync(payment, signature);

// место вызова не изменилось, поиск всё ещё на этапе компиляции
```

```
// Пакет v2
extension(Payment payment)
{
    public string SignPayload(string secret) =>
        Hmac.Sign(JsonSerializer.Serialize(payment), secret);
} // шлюз: подпись не совпала
```

Код компилируется. Шлюз отклоняет запрос: изменилась политика подписи.

Как снизить риск?

Необходимо следовать правилам:

- не использовать **global using** для важных расширений;
- разделять расширения по пространствам имён и ограниченным контекстам;
- безопасность, деньги и идемпотентность — через явный API;
- пересобирать и прогонять тесты при обновлении SDK и зависимостей;
- использовать анализаторы и проверки CI для опасных пространств имён;
- на ревью задавать вопрос: кто владеет этим API?

А что дальше?

Куда движутся Extensions?

C# 14 был только началом

- **C# 14**: методы, свойства, статические члены и операторы-расширения
- **C# 15**: индексаторы-расширения

Активная работа над компилятором: константы-расширения, расширения без явно заданного типа получателя
Прорабатываемые идеи: конструкторы, события, адаптация через интерфейсы

Важно: не все идеи входят в официальную дорожную карту – нужно различать реализованные возможности, активную разработку и предложения

Модель расширений развивается в двух направлениях: появляются новые виды членов и углубляется их интеграция с синтаксисом и языковыми конструкциями C#.

C# 15: Индексаторы

Как работают индексаторы-расширения:

- Индексатор объявлен вне типа
- Вызов выглядит как обычный *obj[index]*
- Получатель должен быть именованным
- Тип не получает настоящий индексатор
- Поиск выполняется на этапе компиляции

Поиск индексатора: у кого приоритет?

Собственный API типа сильнее внешних расширений:

- обычный индексатор экземпляра
- неявный индексатор экземпляра
- индексатор-расширение
- неявный индексатор-расширение

Собственный индексатор типа имеет приоритет над расширением.

Extension Members расширяют видимый API, но не вытесняют собственные члены типа.

C# 15: Индексаторы

```
public static class BatchExtensions
{
    extension(MyAwesomePayBatch batch)
    {
        public Payment this[int index] =>
            batch.Payments[index];
    }
}

var payment = batch[0];
```

Синтаксически – часть API типа. Реализация – вне типа.

Поиск индексатора: у кого приоритет?

Собственный API типа сильнее внешних расширений:

- обычный индексатор экземпляра
- неявный индексатор экземпляра
- индексатор-расширение
- неявный индексатор-расширение

Собственный индексатор типа имеет приоритет над расширением.

Extension Members расширяют видимый API, но не вытесняют собственные члены типа.

Extensions входят в языковые конструкции

Индексаторы-расширения и языковые конструкции:

- доступ по индексу: obj[index]
- Index и Range
- списочные шаблоны
- шаблоны с проверкой количества элементов
- тип по-прежнему не реализует новый интерфейс

Extensions входят в языковые конструкции

```
extension(MyAwesomePayBatch batch)
{
    public int Length => batch.Payments.Count;

    public Payment this[Index index] =>
        batch.Payments[index];

    public IEnumerable<Payment> this[Range range] =>
        batch.Payments[range];
}

var last = batch[^1];
var firstTwo = batch[..2];
```

Extension влияет уже не только на obj.Member, но и на то, как язык умеет работать с объектом.

Над чем продолжается работа?

Активная разработка: константы-расширения

```
extension(Money)
{
    public const int DefaultScale = 2;
}

const int scale = Money.DefaultScale;

void CreateAmount(int scale = Money.DefaultScale) { }
```

- статическое свойство ≠ константа
- нужна величина, известная на этапе компиляции
- используется в атрибутах, параметрах по умолчанию
- пока не часть C# 15, но находится в активной разработке

Активная разработка: получатель без явно заданного типа

```
// сегодня это проблемное место:
[1, 2, 3].ToImmutableArray();

// выражение коллекции не имеет собственного типа
// поиск расширения начинается с определения типа получателя
```

- поиск расширений может участвовать в выводе типа
- особенно важно для выражений коллекций
- потенциально – для лямбда-выражений и условных выражений
- пока это проектирование и работа над компилятором

Следующий уровень: расширение может участвовать в определении типа самого получателя.

Вопрос проектирования: Конструкторы-расширения

```
// обычный extension:
extension(Money money)
{
    public long MinorUnits => ...;
}
// receiver уже существует

// идея конструктора:
new ExternalType(...)
// receiver ещё не существует
```

- кто здесь выступает получателем?
- кто отвечает за создание объекта?
- это всё ещё конструктор или уже фабрика?
- как это соотносится с моделью конструкторов CLR?

Конструктор – первый случай, где получателя ещё не существует.

События-расширения: где хранить состояние?

```
extension(Control control)
{
    public event EventHandler
DoubleClicked
    {
        add    => ...;
        remove => ...;
    }
}

button.DoubleClicked += handler;
```

События-расширения упираются уже не столько в синтаксис, сколько в состояние и время жизни:

- где хранить список подписчиков?
- кому принадлежит время жизни подписки?
- возможно ли автоматическое хранение обработчиков?

Внешний API может выглядеть как член типа, но это ещё не решает проблему владения состоянием.

Граница: API или система типов?

```
// сегодня:
payment.Requires3DS
// API, похожий на член типа

// другой уровень:
extension(int) : IPrintable
{
    void IPrintable.Print() => ...;
}
// реализация интерфейса
// ограничения / диспетчеризация
```

В C# 14 тип определяет применимые расширения.

В следующей модели расширения могут участвовать в выводе самого типа.

Следующая граница – момент, когда расширения начинают участвовать не только в поиске членов, но и в типизации.

Куда движутся расширения?

больше видов членов + глубже интеграция с языком

Уже есть:

- **C# 14**: методы, свойства, статические члены, операторы
- **C# 15**: индексаторы-расширения

В работе:

константы-расширения, получатель без явно заданного типа

Область проектирования:

конструкторы, события, адаптация через интерфейс

Пока не входит в планы:

Roles (исторический эксперимент с моделью языка)

Extension Everything (направление развития всей модели расширений)

Где должны находиться расширения?

Место расширения должно соответствовать владельцу поведения:

- Доменные расширения – рядом с предметной моделью
- Интеграционные расширения – рядом с кодом шлюза или SDK
- Тестовые расширения – только в тестовых проектах

Зона повышенного риска – деньги, безопасность, идемпотентность, критичные бизнес-правила

Главная мысль

C# 14 не меняет систему типов.

Он меняет контекст вызова:
внешний API может выглядеть как API, похожий на член типа.

Именно это нужно учитывать при проектировании библиотек и долгоживущих систем.

Практические выводы

Как использовать Extension Members осознанно?

Что мы получили?

- выразительный API
- цепочки вызовов
- предметно-ориентированный синтаксис
- фасад над чужими типами, выглядящий как их собственный API
- более естественный и читаемый код

За счёт чего?

- простое определение владельца API
- более предсказуемое поведение
- прозрачность источника логики
- безопасная эволюция публичных библиотек

Где Extension Members будут полезны

- фасад, похожий на собственный API типа, для внешних шлюзов и SDK
- локальный предметно-ориентированный синтаксис внутри ограниченного контекста
- вычисляемые свойства без критичной бизнес-логики
- вспомогательные расширения для тестов и фабрики тестовых данных

Где Extension Members могут вызвать трудности

- публичные библиотеки и общие SDK
- безопасность, подпись, идемпотентность, оценка рисков
- финансовые правила, комиссии и правила округления
- слишком широкие типы получателя: *object, string, IEnumerable<T>*
- операторы с неочевидной семантикой
- *global using* и скрытая область видимости расширений

Практические правила

- Критичное поведение должно оставаться явным
- Не каждый вспомогательный метод должен выглядеть как часть типа
- Область видимости расширений должна быть предсказуемой и контролируемой
- Операторы допустимы только при очевидной семантике
- Источник истины – поиск на этапе компиляции

Главный вывод

тип + видимые расширения + поиск на этапе компиляции =
API, который выглядит как часть типа

Extension Members – это не способ изменить тип.

Это способ изменить видимый API для типа в конкретном контексте компиляции.

Проектировать нужно не только типы, но и контекст, в котором видны расширения.

Материалы и источники

Официальная документация

- Microsoft Learn: Extension methods / extension members
- Microsoft Learn: C# 14 — What's new
- Microsoft Learn: C# 14 extension declarations / proposal
- Microsoft Learn: C# 15 — What's new
- Microsoft Learn: C# 15 union types
- Microsoft Learn: collection expression arguments

Блоги и анонсы

- .NET Blog / Microsoft materials: extension members and design updates
- .NET Blog: Union types in C# 15

Design discussions

- csharplang #5496: Roles and extensions
- csharplang #8548: The design space for extensions
- Microsoft Learn: C# 14 extension declarations / extension operators / design space

Мои контакты

@dzitskiy_dev
dzitskiy@gmail.com