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

Виктор Дзицкий
TeamLead .NET Developer
Solarlab

Вопрос в зал

Вопрос в зал

Кто активно использует extension methods?

Главный тезис

Extension Members не меняют тип —
они расширяют видимый API в текущем контексте.

type + visible extensions + lookup = member-like API

В чём смысл?

Это не просто развитие **extension methods**.

Это шаг к более богатой модели внешнего API:
код выглядит как работа с членами типа,
но разрешается через compile-time lookup.

О чём поговорим

- Зачем появились extension methods
- Что изменилось в C#14
- Что происходит под капотом
- Где могут ждать неприятности
- Как проектировать extension API
- Практические выводы

Сквозной пример

Задача:

Подключить новый платежный шлюз MyAwesomePay.

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

Нужно добавить gateway-specific поведение рядом с интеграционным слоем, не раздувая домен.

Будем работать с **Money** и **Payment**.

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);
```

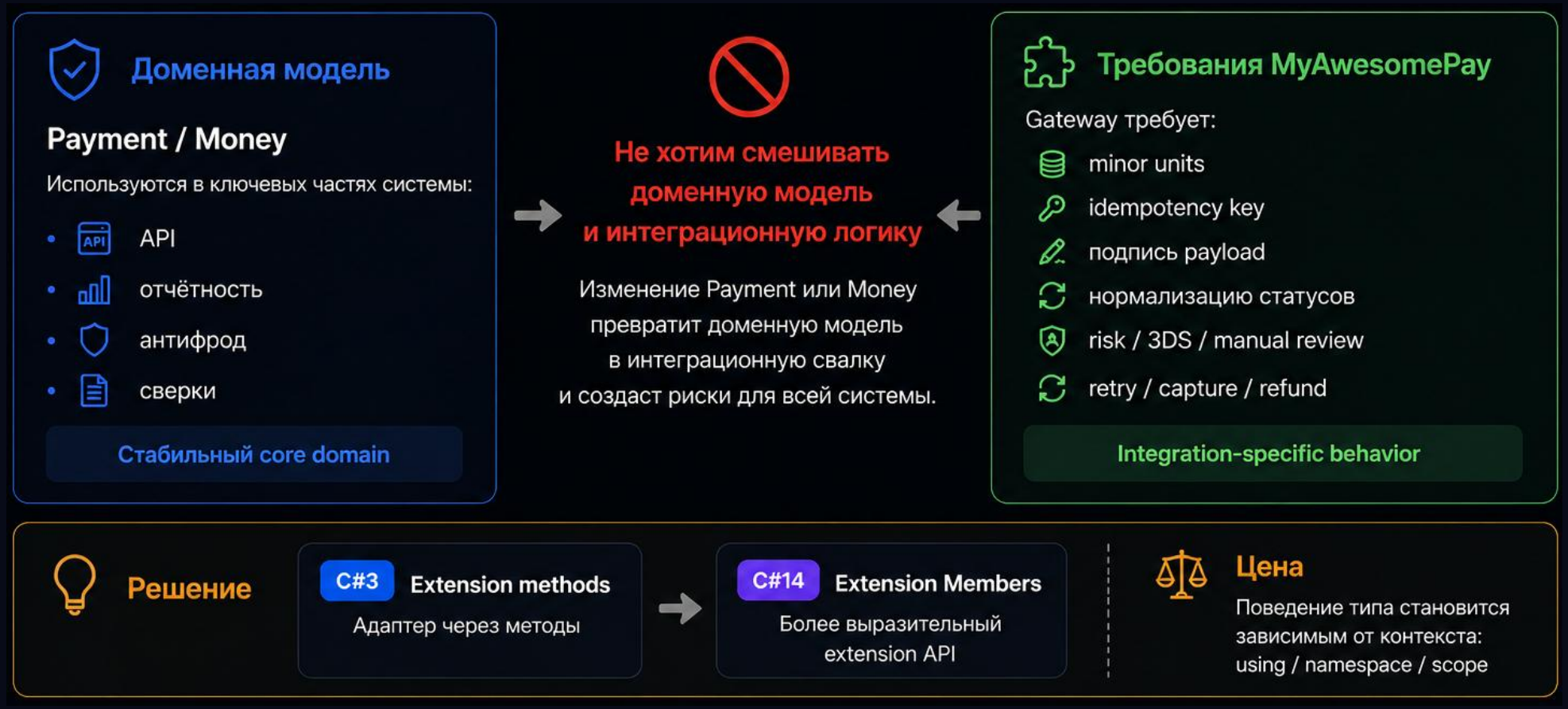
Что делать с требованиями MyAwesomePay

Gateway приносит сложную логику, которая начинает выглядеть как “обычные свойства и методы”.

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

И всё это легко спрятать в extensions.

Почему нельзя просто менять Payment и Money



Эволюция extension methods

Зачем они появились в C# и почему старого подхода со временем стало недостаточно.

C#3 • **LINQ** • **using System.Linq** • **methods only**

Контекст появления

C#3 принёс LINQ, лямбды, expression trees и query syntax.

Extension methods стали инфраструктурой, которая позволила LINQ расширять существующие типы через using System.Linq.

Главная задача

Расширять существующие типы без наследования, перекомпиляции и изменения исходного кода.

LINQ как главный драйвер

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

Форма extension method

Минимальная спецификация: static class + static method + this receiver

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

- объявляется в top-level static class
- сам метод тоже static
- this Payment payment задаёт receiver
- вызывается как instance method после using
- фактически остаётся static method

Классический extension method

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

Вызов выглядел как instance method

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

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

Польза / ограничение / риск

Польза	Ограничение	Риск
адаптируем Money к требованиям gateway	только методы: ToMinorUnits(), CanBeCaptured()	валютные правила могут спрятаться в helper-е
удобно для LINQ-like агрегаций	нет properties/static/operators	широкие extensions над IEnumerable могут стать магией
не меняем исходные типы	поведение приходит из namespace	business logic уезжает из доменной модели

Границы модели

Почему это был facade, а не изменение типа

```
// extension-style вызов
payment.ToMyAwesomePayRequest(merchantId);

// тот же метод можно вызвать как static method
PaymentGatewayExtensions.ToMyAwesomePayRequest(
    payment, merchantId);
```

- добавляются только методы
- это внешний static API
- нет extension properties
- нет static members на типе
- нет extension operators
- обычные members типа приоритетнее

Чем Extension Members не являются

Extension Members — это не:

- interface implementation
- virtual / override
- dynamic dispatch
- доступ к private state
- настоящий member в reflection исходного типа

Почему methods-only стало не хватать?

- не всё удобно выражать методами
- вычисляемые признаки лучше читаются как свойства
- фабрики естественнее выглядят как статическое API типа
- операции над value object хочется писать как операторы
- pattern matching удобнее работает со свойствами

Почему methods-only стало не хватать?

Fluent API - вызовы строятся цепочкой и читаются почти как естественный язык (как “родной API типа”).

Но methods-only модель заставляла писать:

```
money.IsPositive()
```

вместо

```
money.IsPositive
```

и

```
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?

Domain primitives - это value objects, которые моделируют бизнес-сущности.

helper-style API:

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

на “естественном” API будет выглядеть более лаконично:

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

Почему methods-only стало не хватать?

Для конфигурации, которая читается как декларативное описание системы:

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

С **Pattern matching** можно описывать логику через patterns, а не через imperative conditions:

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

Что добавил C# 14

Extension blocks, properties, static members, operators
и новый уровень выразительности.

extension blocks • **properties** • **static members** • **operators**

C# 14 / .NET 10

C#14 добавляет **extension blocks**.

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

Это не отменяет C#3-модель, а достраивает её.

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}";
}
```

Применение свойств

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

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));
        }
    }
}
```

Читается как свойство, но содержит правила

Польза: читабельность.

Опасность: валютные правила выглядят как простое свойство.

Payment property

```
extension(Payment payment)
{
    public bool CanRetryGatewayCall =>
        payment.Status is PaymentStatus.Created
            or PaymentStatus.Failed;

    public bool RequiresManualReview =>
        payment.Risk.RequiresManualReview;
}
```

Pattern matching и extension properties

Property pattern начинает видеть внешний API

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

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

Польза:
читаемый код, удобно использовать для DSL и workflow

Ограничение:
property не часть Payment, зависит от using

Опасность:
источник правила не виден в месте вызова, скрытая бизнес-логика в pattern matching

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 static API

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

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

Extension operators

- operators: + почти полностью скрывает источник поведения
- составное присваивание: += может стать локальным DSL
- DSL-примеры: ~payload, writer << message, list += item
- duck typing: foreach / await / deconstruct через pattern lookup

Extension operator: выразительно, но требует дисциплины

```
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;
```

Хорошо:
код близок к доменному языку,
удобно для value object,
легко читать happy path.

Плохо:
источник оператора не в Money,
другая библиотека может принести
другой оператор/правила,
критичная политика может стать
невидимой.

Иногда явный API лучше красивого оператора

```
var fee = Money.Usd(1.25m);
var total = payment.Amount +
fee;

Console.WriteLine(
total.DisplayAmount);
```

```
// Более честный API
для критичных правил
var total =
MoneyCalculator.Add(
    payment.Amount,
    fee,
    CurrencyPolicy.Strict);
```

Extension operator: syntax → static implementation

```
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);
```

Что важно:
operator приходит из extension scope, а не из самого Money.

Почему это опаснее, чем property:
символ + почти полностью скрывает источник поведения.

Для денег это означает:
правила валют, округления и комиссий могут спрятаться за одним символом.

Compound assignment: += как локальный DSL

```
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;
```

В MyAwesomePay можно реализовать:
- draft += payment
- draft += retryQueue

Но это compile-time sugar:
- оператор найден через scope/using
- поведение не в Payment
- публичный API должен быть очевиден
- для value types важен ref receiver

Операторы как DSL: можно, но не всегда нужно

```
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)
};
```

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

На примере MyAwesomePay можно представить компактный синтаксис для сборки тела запроса, но оператор должен быть очевиден всей команде. Иначе это не DSL, а скрытый локальный язык.

Pattern-based API: duck typing без интерфейса

Extension member может включить языковую конструкцию

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

foreach (var payment in batch)
{
    await gateway.SendAsync(payment);
}

// batch не стал IEnumerable<Payment>
// но foreach увидел pattern.
```

Pattern-based точки входа:

- foreach → GetEnumerator()
- await → GetAwaiter()
- deconstruct → Deconstruct()
- initializer → Add()

Контекст компиляции может сделать тип “похожим” для конкретной языковой конструкции.

Что изменилось в C#14?

C# 14 дал мощный инструмент для выразительного API.

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

Можно ли было сделать иначе?

C# Extension Members

member-like syntax + compile-time lookup
не меняют систему типов

Kotlin extensions
похожая идея: удобный вызов без настоящего override

Rust traits / type classes
поведение участвует в типизации и constraints

Go interfaces
тип может удовлетворять интерфейсу структурно

Что происходит под капотом

От кода до ILSpy:
как компилятор связывает extension member

member-like syntax • static implementation • compile-time lookup

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 member
- Компилятор находит вызов среди видимых расширений
- Реализация остаётся во внешнем static-классе
- Вызов выглядит как API типа, но принадлежит внешнему коду

Что попадает в сборку

```
extension block in source
        ↓
   Roslyn binding
        ↓
static implementation member
        +
   extension metadata
        ↓
call site binds to implementation
```

Что покажет Reflection

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

// false: это не член исходного типа

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

// implementation / metadata
```

Что делает Roslyn

Parse → syntax tree
Bind → receiver + visible extensions
Resolve → overload resolution
Emit → static implementation + metadata

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 operators зависит от версии
- отдельно проверяем навигацию, автодополнение и рефакторинг

Рассмотрим пример

Money.cs

```
1  using System;
2
3  namespace ExtensionMembersDemo.Domain;
4
5  public readonly record struct Money(decimal Amount, string Currency)
6  {
7      public override string ToString() => $"{Amount:0.00} {Currency}";
8  }
9
```

115 % No issues found Ln: 1, Ch: 1 SPC LF Windows 1251

Практические сценарии

Что может сломаться в реальном проекте:
package, using, global using и recompile

ambiguity • **hidden logic** • **semantic compatibility** • **versioning**

Кейс 1: поставили пакет — появился 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());
```

Кейс 1: добавили using/global using — получили ambiguity

Библиотека B

```
namespace Reporting.Extensions;

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

Добавили using

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

var amount = Money.Usd(125.50m);

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

Вывод кейса 1

Подключение библиотеки может сломать сборку.

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

А из-за нового extension member в visible scope.

Кейс 2: package update меняет overload resolution

Широкое расширение

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

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

Клиентский код v1

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

Console.WriteLine(
    payment.ToGatewayLogLine());

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

Кейс 2: тот же source code выбирает другой candidate

Более специфичное расширение

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

Клиентский код v1

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

Console.WriteLine(
    payment.ToGatewayLogLine());

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

Вывод кейса 2

Код не изменился.

Изменился набор доступных extensions.

После пересборки может выбраться другой кандидат.

Это уже **semantic compatibility**, а не только **binary compatibility**.

Кейс 3: виды совместимости

Binary compatibility

Сборка может продолжать загружаться.
Но это ещё не значит, что source code после пересборки выберет тот же extension.

Source compatibility

Код может перестать компилироваться из-за неоднозначности или начать резолвиться иначе после обновления пакета.

Кейс 3: semantic compatibility

Behavior compatibility

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

Тот же код может продолжать компилироваться, но означать другое

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

Console.WriteLine(
    payment.ToGatewayLogLine());

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

Вывод кейса 3

Binary compatibility
≠
Source compatibility
≠
Behavior compatibility

Кейс 4: operator и бизнес-правила

```
var total = amount + fee;

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

Для сложных доменных правил лучше заменить на explicit 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, shared project, SDK-style шаблона или infrastructure package.

Кейс 6: runtime без runtime lookup

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

// call-site не изменился
// lookup всё ещё compile-time
```

Код компилируется. Gateway отклоняет запрос: изменилась policy подписи.

```
// Package v2
extension(Payment payment)
{
    public string SignPayload(string secret) =>
        Hmac.Sign(JsonSerializer.Serialize(payment), secret);
}
// gateway: signature mismatch
```

Как снижать риск в production

```
// так лучше не делать
global using MyAwesomePay.Extensions;

// локальный scope
using MyAwesomePay.GatewayFacade;

// DSL — только там, где он ожидаем
using MyAwesomePay.ExperimentalDsl;
```

Правила:

- no global using для gateway extensions
- namespace per bounded context
- security/money/idempotency — explicit API
- recompile tests для SDK
- analyzer/CI для опасных namespaces
- review: кто владеет API?

Extension Everything и C# 15

Обозначим, что уже известно про фичи C#15, а также обсуждения extensions, roles и design space.

C#15 • union types • roles • design space

Extension Everything

Идея:

Расширять не только methods, но разные сущности языка.

Важно:

Properties, operators и static members уже в C#14.
Всё остальное — design direction, а не обещанный синтаксис C#15.

C#15: не новые Extension Members

Официальные preview-фичи C#15 нужно отделять от design discussions.

Union types и **collection expression arguments** полезны для выразительного API, но это не новые виды Extension Members.

Extension Members в C#15: таблица статусов

Сущность	Статус
union types	официальная C# 15 preview-фича
collection expression arguments	официальная C# 15 preview-фича
extension constructors / indexers / events	design space / proposals
roles / extension everything	design discussion

C#15 preview

Официально: union types и collection expression arguments

```
public union GatewayResult(
    GatewayApproved,
    GatewayDeclined,
    GatewayTimeout);

List<Payment> retryQueue =
    [with(capacity: 32), payment];
```

- union types позволяют моделировать фиксированные варианты результатов
- collection expression arguments делают API более декларативным
- это не новые Extension Members
- C# 15 важен как направление развития языка
- язык становится более контекстным и выразительным

Microsoft Learn — Collection expression arguments

C#15: union type для результата шлюза

```
public sealed record GatewayApproved(string Id);
public sealed record GatewayDeclined(string Reason);
public sealed record GatewayTimeout(TimeSpan
Timeout);

public union GatewayResult(
    GatewayApproved,
    GatewayDeclined,
    GatewayTimeout);
```

C#15: collection expression arguments

```
List<Payment> retryQueue =
    [with(capacity: 32), payment];

HashSet<string> idempotencyKeys =
    [with(StringComparer.OrdinalIgnoreCase),
     "KEY-1", "key-1"];
```

Design direction

В csharplang обсуждениях extensions рассматриваются шире, чем конкретная фича C#14.

Здесь важно увидеть компромиссы.

Extensions as design space

Ось выбора	Что выигрываем	Что платим
выразительность	API ближе к предметной области	хуже видно источник поведения
локальность	gateway-логика рядом с integration layer	поведение распределено по namespace
совместимость	не меняем исходные типы	semantic compatibility сложнее binary
расширяемость	можно расширять чужие типы	using становится частью модели API

Design space

Что обсуждается в design space extensions:

- generic non-method members
- receiver passing semantics
- nullable receiver
- grouping extensions
- compatibility with classic extension methods

Roles and extensions

Roles — это design discussion, а не то, что делает C#14.

Главная мысль

C#14 не меняет систему типов.

Он меняет контекст вызова:
внешний API может выглядеть как member-like API.

Именно это нужно учитывать при проектировании библиотек и долгоживущих систем.

Extensions

Версия	Что добавляет
C# 3	extension methods
C# 14	properties, static members, operators
C# 15 preview	union types, collection expression arguments
Design discussions	roles / extension everything

Практические выводы

Как использовать Extension Members осознанно

правила • рекомендации

Что мы получили

- Выразительность
- fluent API
- DSL
- member-like facade поверх чужих типов
- более естественный синтаксис

За счет чего

- простота определения владельца
- часть предсказуемости
- прозрачность источника поведения
- безопасная эволюция публичных библиотек

Где Extension Members действительно полезны

- member-like facade для внешних gateway API
- локальные DSL внутри bounded context
- computed properties без критичных правил
- test helpers и фабрики тестовых данных
- миграция от C#3 extension methods к extension blocks

Где Extension Members могут вызвать трудности

- публичные библиотеки и shared SDK
- security / signature / idempotency / risk logic
- финансовые правила, комиссии, rounding policy
- слишком широкие receiver-типы: object, string, IEnumerable<T>
- operators с неочевидной семантикой
- global using и скрытый extension scope

Практические правила

1. Критичное поведение должно оставаться явным
2. Не каждый хелпер должен выглядеть как часть типа
3. Extension scope должен быть предсказуемым и контролируемым
4. Operators допустимы только при очевидной семантике
5. Source of truth — compile-time lookup

Вывод

Extension Members — не способ изменить тип.

Проектировать нужно не только типы, но и visible extension context.

type + visible extensions + compile-time lookup = member-like API

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

[████████████████████] Completed!