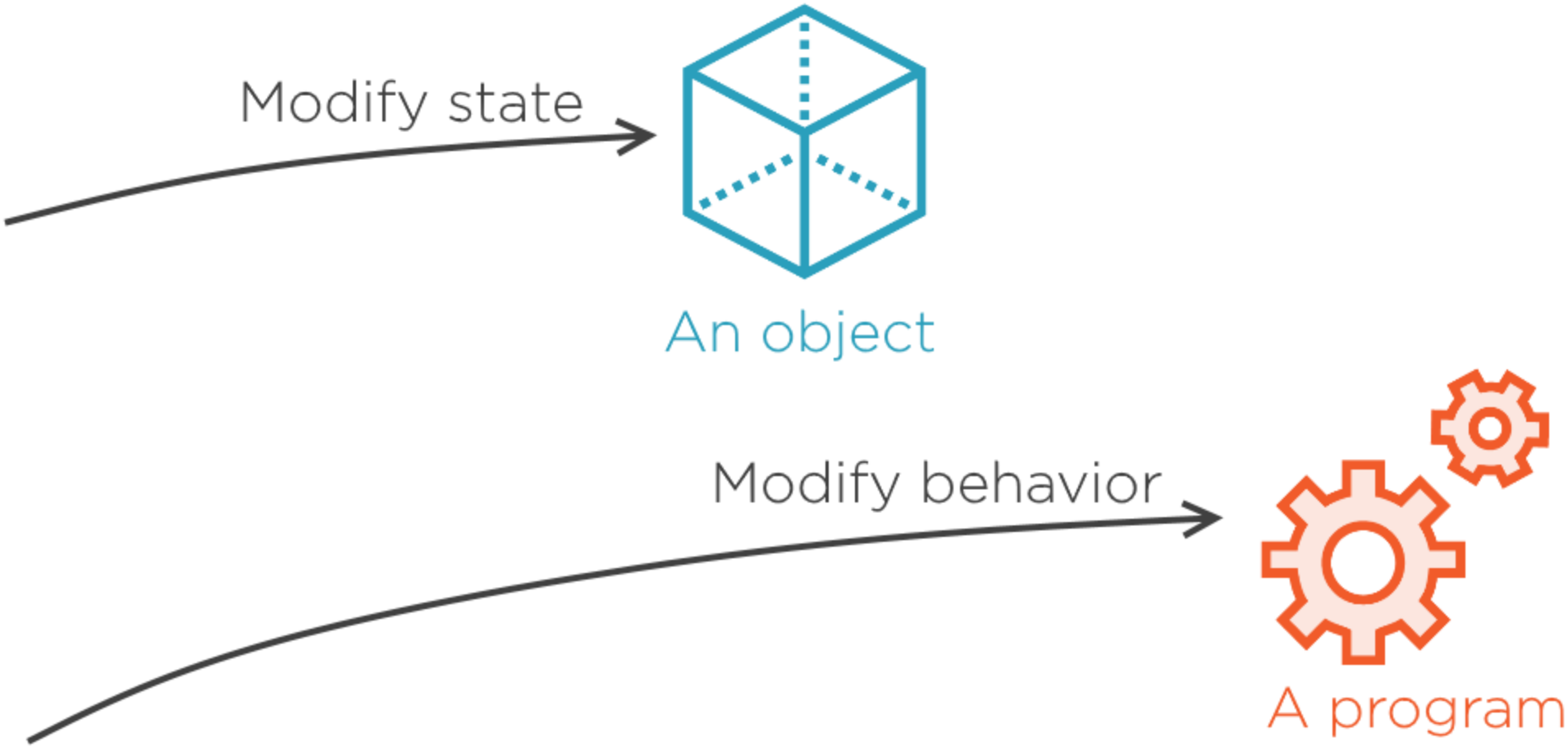
Metaprogramming with Extension Methods

Zoran Horvat

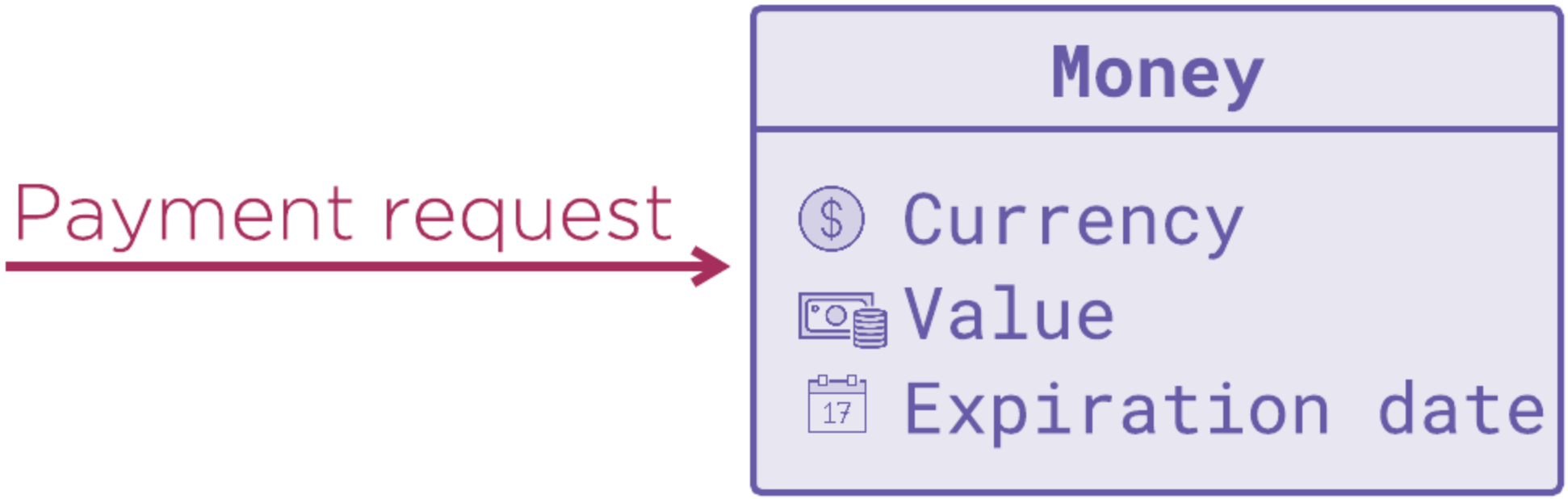
CEO AT CODING HELMET

@zoranh75 http://csharpmentor.com

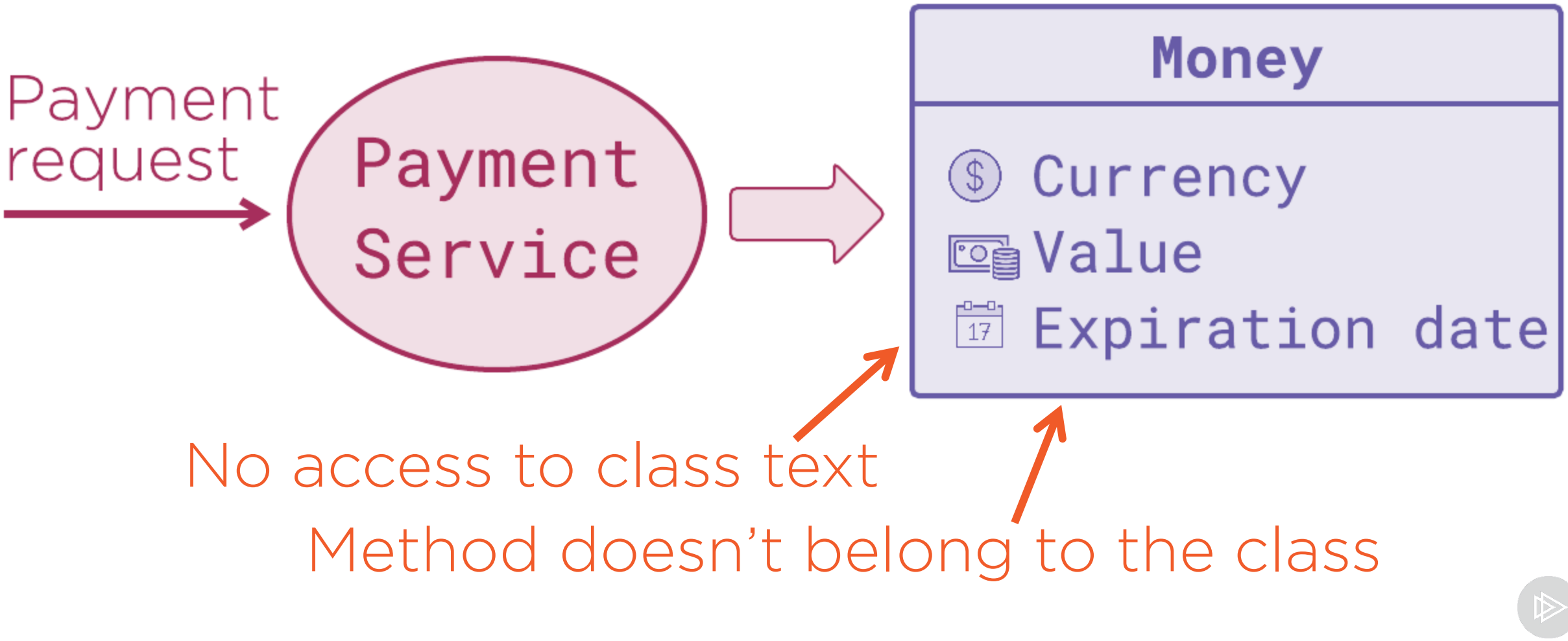
Introducing Metaprogramming



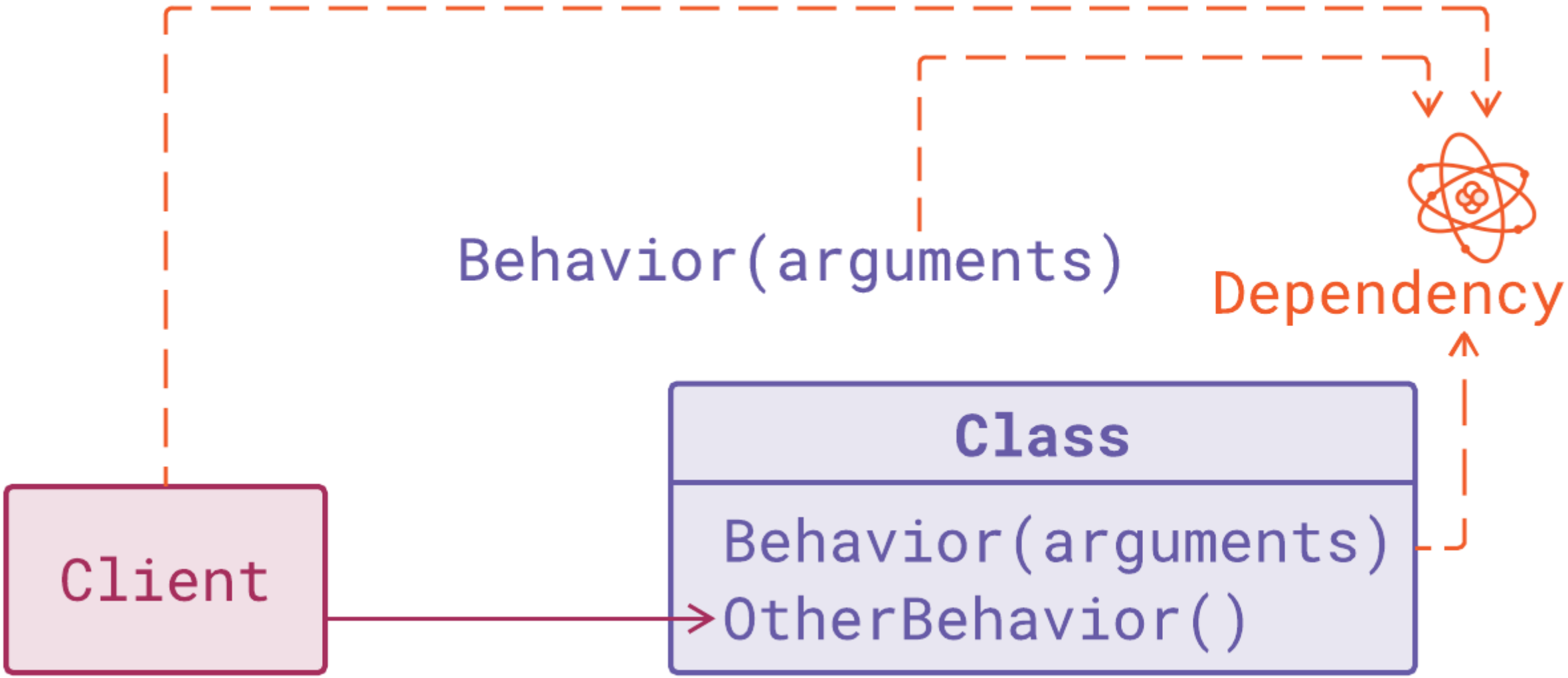
Introducing Metaprogramming



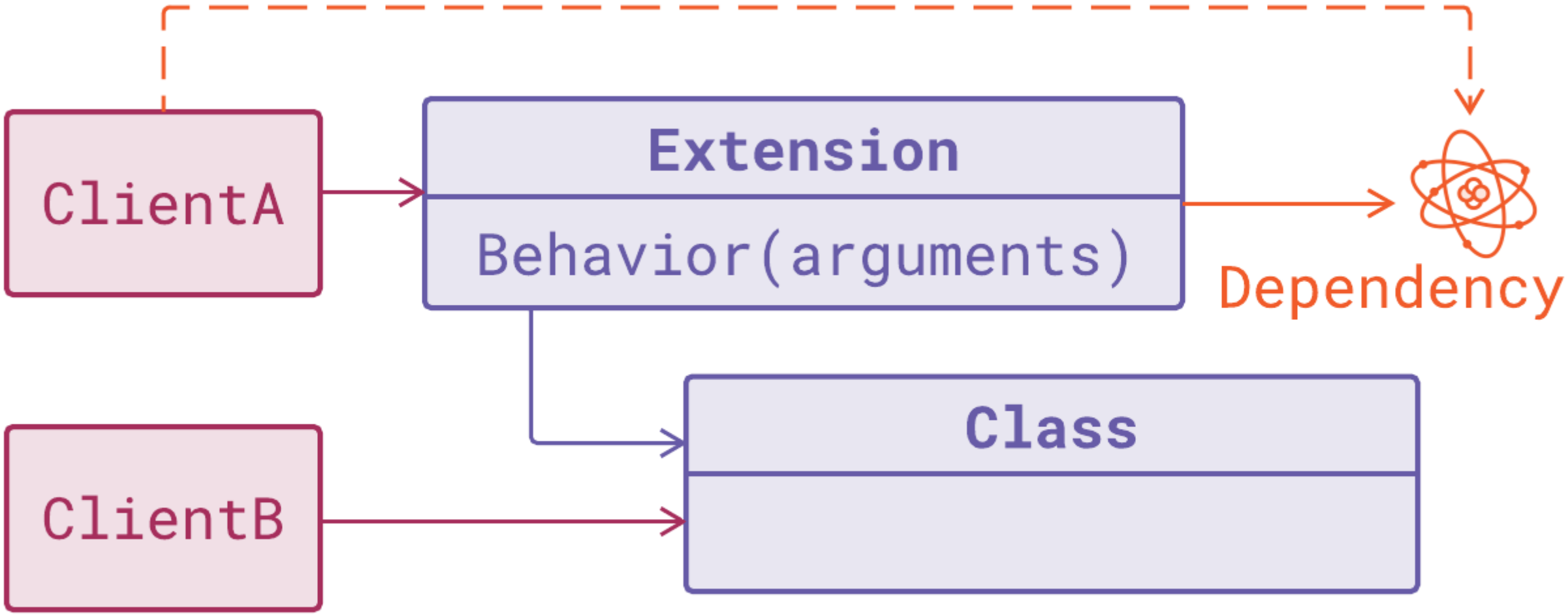
Introducing Metaprogramming



Introducing Metaprogramming



Introducing Metaprogramming



Metaprogramming in C#

Attaching methods to a class

```
class MyClass
{
}    class MyExtensions
     {
       public static void Attached(this MyClass obj) ...
     }
```

Consuming an attached method

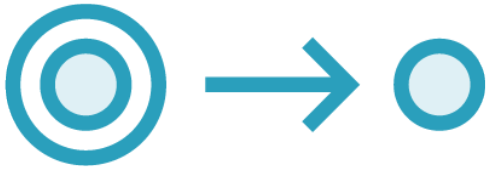
```
using MyExtensionsNamespace;

obj.Attached();
```

Enclose extensions
in a separate namespace

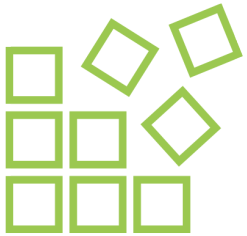
Consumer includes
type namespace **and**
extensions namespace

C#-specific metaprogramming

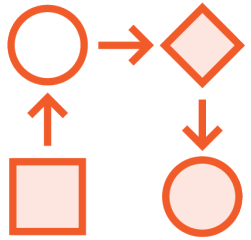
Intrinsic Cost of Code Reuse



Simple functions do not call for refactoring



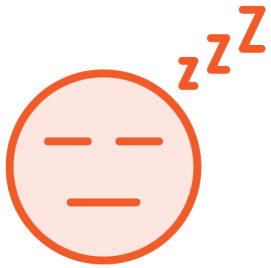
Simple logic rarely ends up in its own class



Code duplication masks correlation

All contrary to rules of good design

Often a subject to copy/paste pattern



We could have learned more about the domain

Understanding Extension Methods

Instance-level method

```
money.Pay(expense)
```

True call with implicit **`this`** reference

Method has access to private fields

Calling methods feels natural

Extension method

```
money.Pay(expense)
```

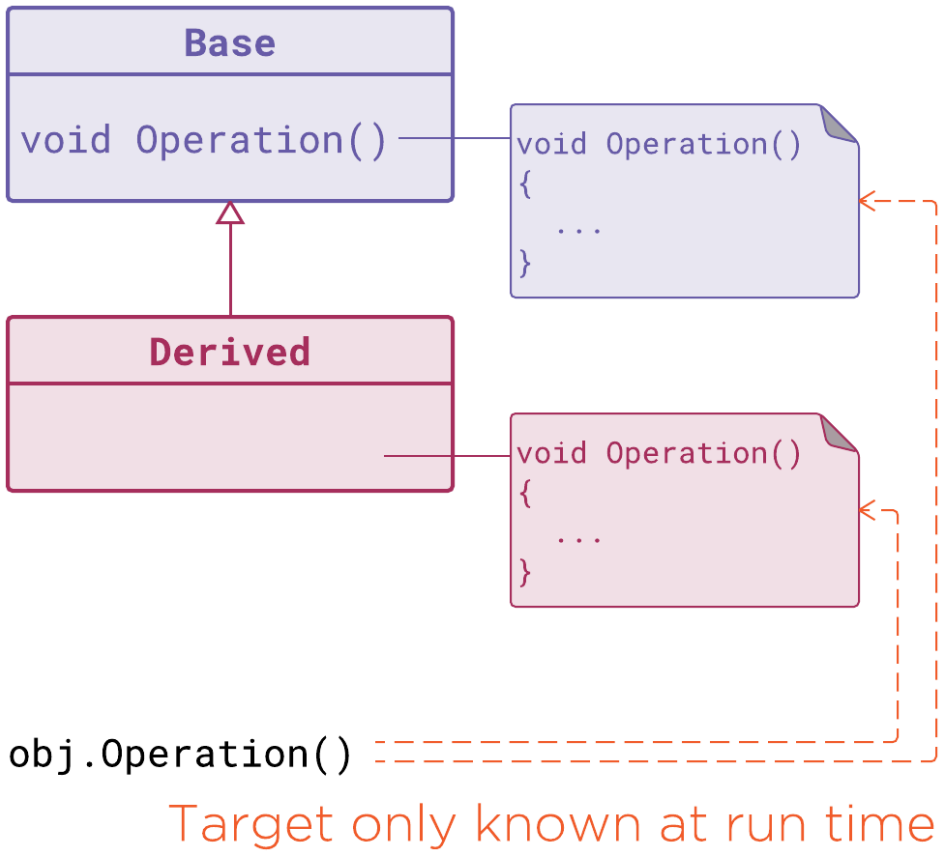
An illusion of passing **`money`** as **`this`**

Method can only access public members

Extension methods retain natural feel

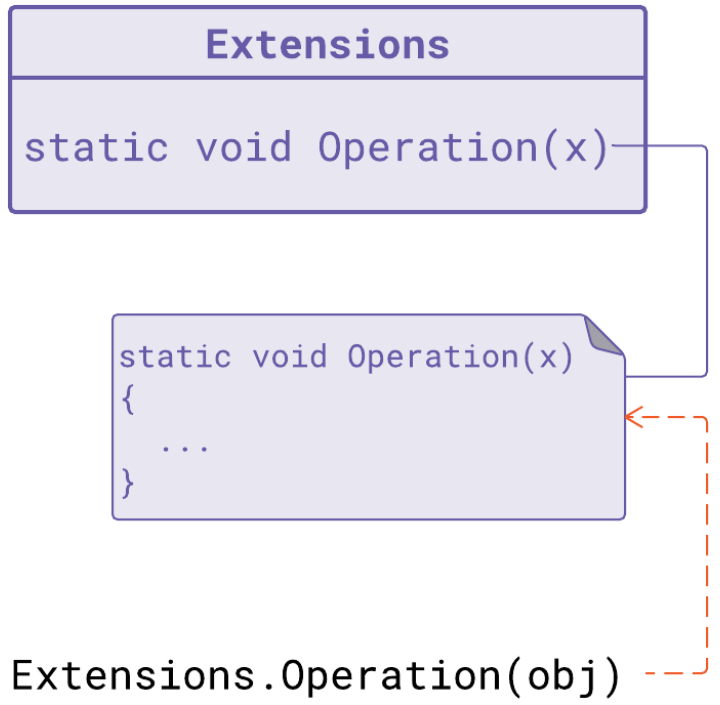
Understanding Extension Methods

Instance-level method



Target only known at run time

Static or extension method



Static call to the only implementation

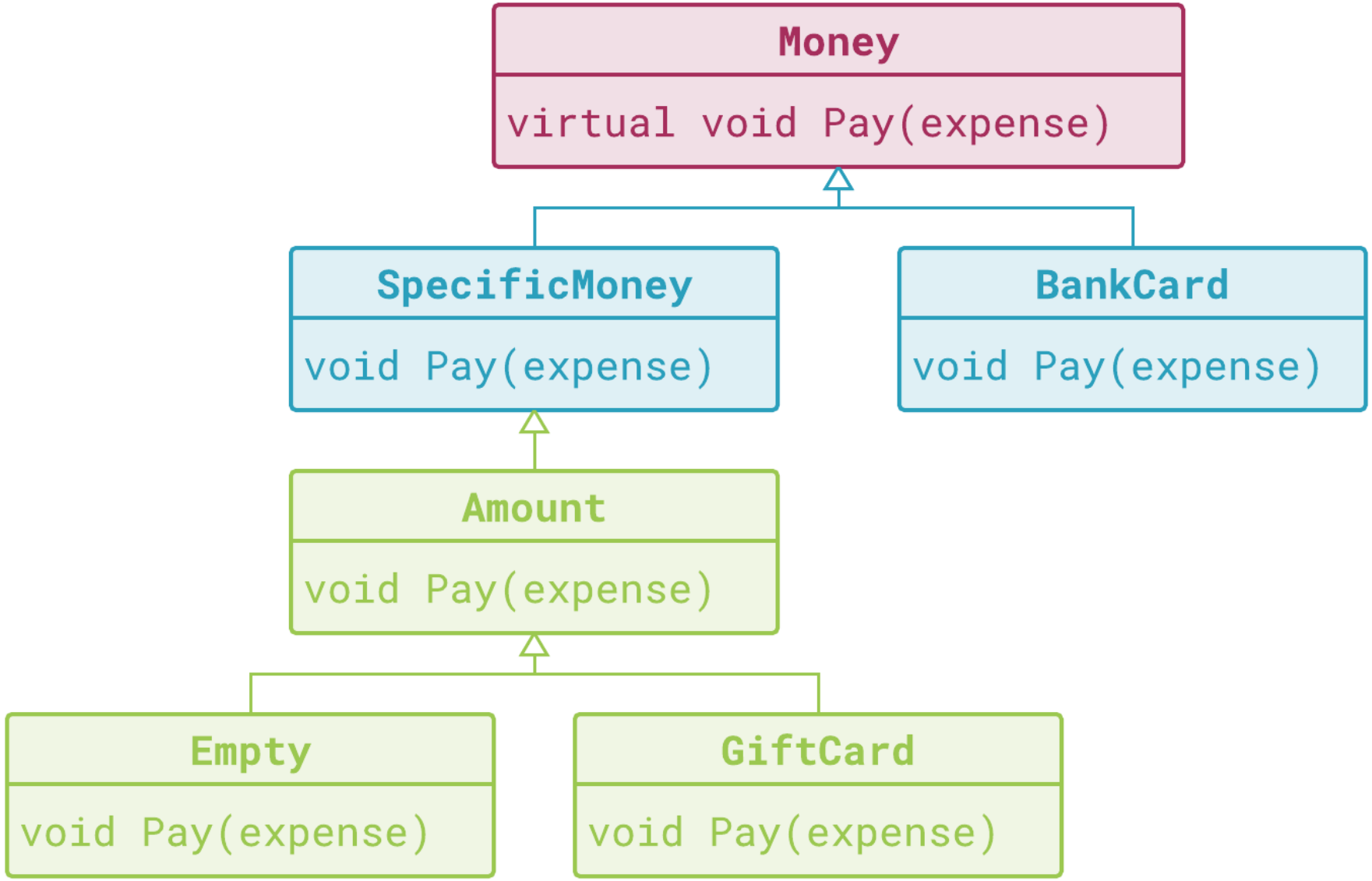
Functional Extensions on Related Types

```
public static (Amount paid, Money remaining) Pay(this Money money, Amount expense)
{
  Timestamp now = Timestamp.Now;
  switch (money)
  {
    case Amount amt when amt.Currency != expense.Currency:
      return (Amount.Zero(expense.Currency), money);
    case Amount amt when amt.Value <= expense.Value: return (amt, Amount.Zero(amt.Currency));
    case GiftCard gift when gift.Currency != expense.Currency:
      return (Amount.Zero(expense.Currency), gift);
    case GiftCard gift when gift.ValidBefore.CompareTo(now) < 0:
      return (Amount.Zero(expense.Currency), Amount.Zero(gift.Currency));
    case GiftCard gift when gift.Value <= expense.Value:
      return (new Amount(gift.Currency, gift.Value), Amount.Zero(gift.Currency));
    case Empty _: return (Amount.Zero(expense.Currency), money);
    case Amount amt: return (expense, amt.Subtract(expense));
    case BankCard card when card.ValidBefore.CompareTo(now) < 0:
      return (Amount.Zero(expense.Currency), Amount.Zero(expense.Currency));
    case BankCard _: return (expense, money);
    default: throw new ArgumentException($"Unsupported money type {money.GetType().Name}.");
  }
}
```

Functional Extensions on Related Types

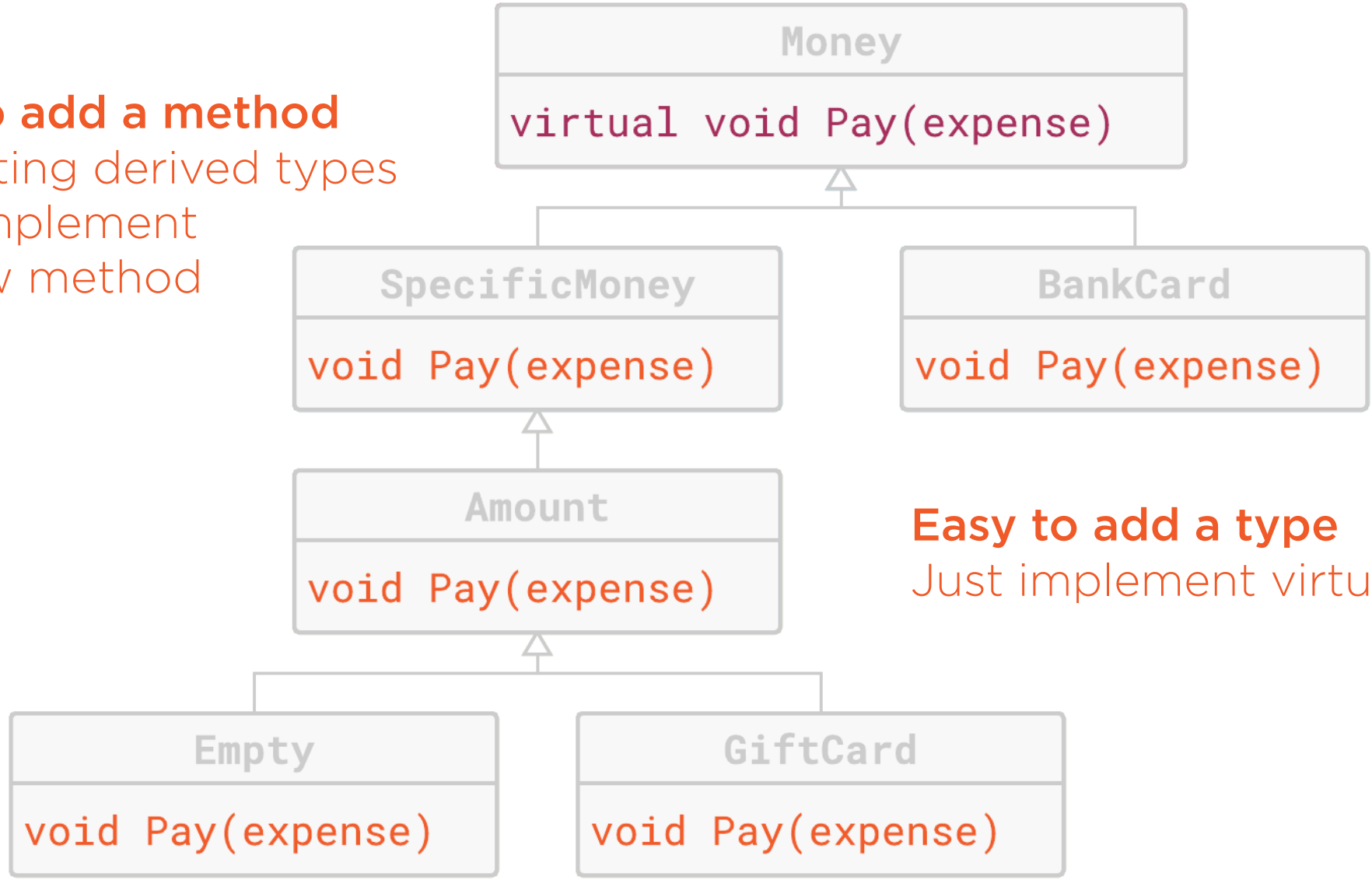
```
public static (Amount paid, Money remaining) Pay(this Money money, Amount expense)
{
  Timestamp now = Timestamp.Now;
  switch (money)
  {
    case Amount amt when amt.Currency != expense.Currency:
      return (Amount.Zero(expense.Currency), money);
    case Amount amt when amt.Value <= expense.Value: return (amt, Amount.Zero(amt.Currency));
    case GiftCard gift when gift.Currency != expense.Currency:
      return (Amount.Zero(expense.Currency), gift);
    case GiftCard gift when gift.ValidBefore.CompareTo(now) < 0:
      return (Amount.Zero(expense.Currency), Amount.Zero(gift.Currency));
    case GiftCard gift when gift.Value <= expense.Value:
      return (new Amount(gift.Currency, gift.Value), Amount.Zero(gift.Currency));
    case Empty _: return (Amount.Zero(expense.Currency), money);
    case Amount amt: return (expense, amt.Subtract(expense));
    case BankCard card when card.ValidBefore.CompareTo(now) < 0:
      return (Amount.Zero(expense.Currency), Amount.Zero(expense.Currency));
    case BankCard _: return (expense, money);
    default: throw new ArgumentException($"Unsupported money type {money.GetType().Name}.");
  }
}
```

Defining a Virtual Method



Defining a Virtual Method

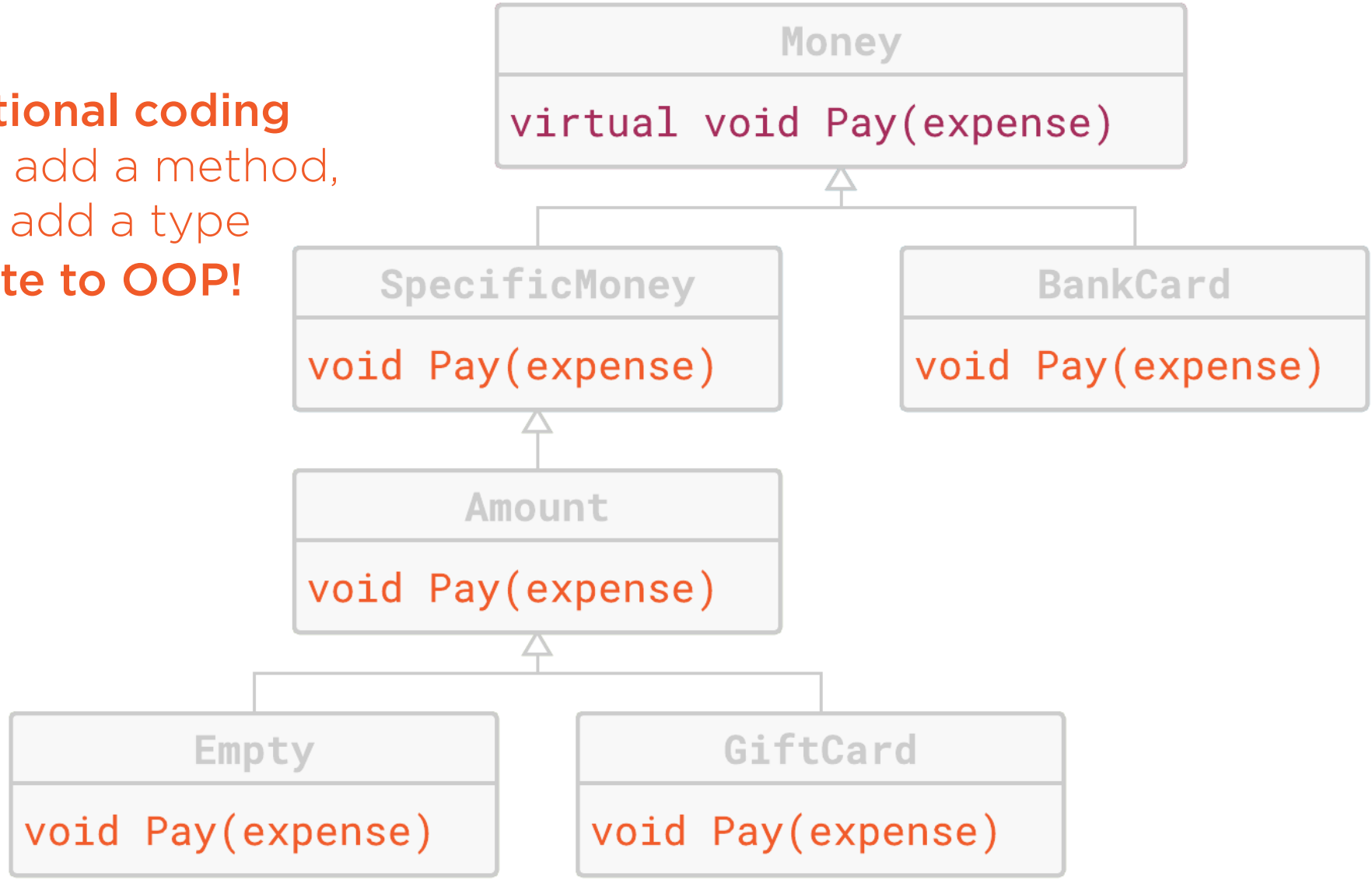
Hard to add a method
All existing derived types must implement the new method



Easy to add a type
Just implement virtual methods

Defining a Virtual Method

In functional coding
Easy to add a method,
hard to add a type
Opposite to OOP!



Summary

Metaprogramming using extension methods

- Explicit passing of the `this` reference
- Modifying the existing type
- "Programming the program"

Supporting functional coding style

- Add one function to a fixed set of plain types
- Keep behavior separate from data
- Enclose extensions in separate namespaces in C#

Next module:
Function Composition with Object Model