Code examples will be in C# and F#

Pipeline Oriented Programming

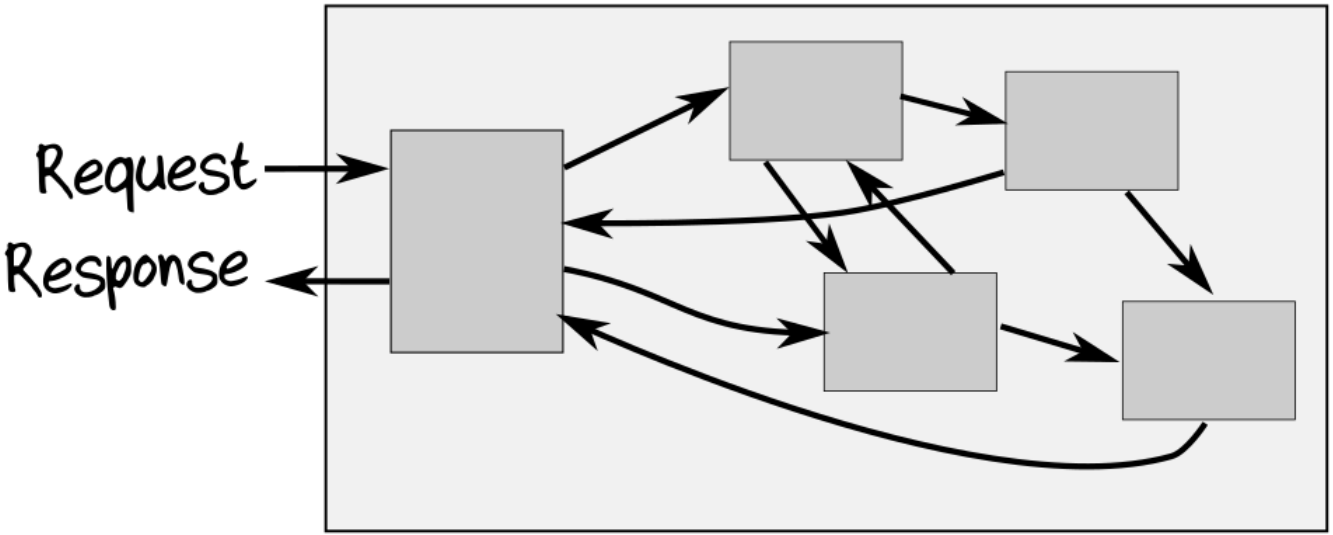
(DotNext 2021)

@ScottWlaschin
fsharpforfunandprofit.com/pipeline

Part I

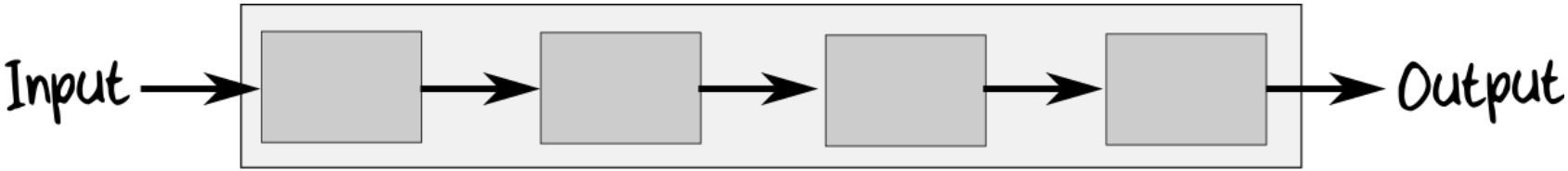
What is “Pipeline-oriented Programming"?

Object-oriented



Arrows go in all directions

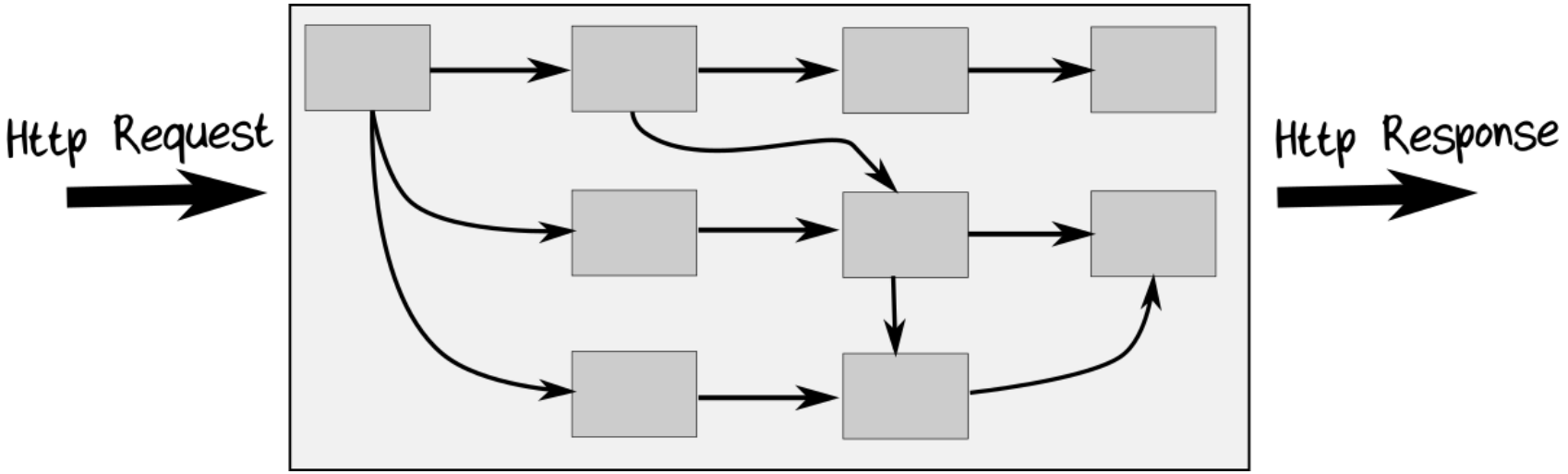
Pipeline-oriented



All arrows go left to right

(just like Unix-style pipes)

A pipeline-oriented web backend



One directional flow,
even with branching

A demo of this is coming later!

What are the benefits of a pipeline-oriented approach?

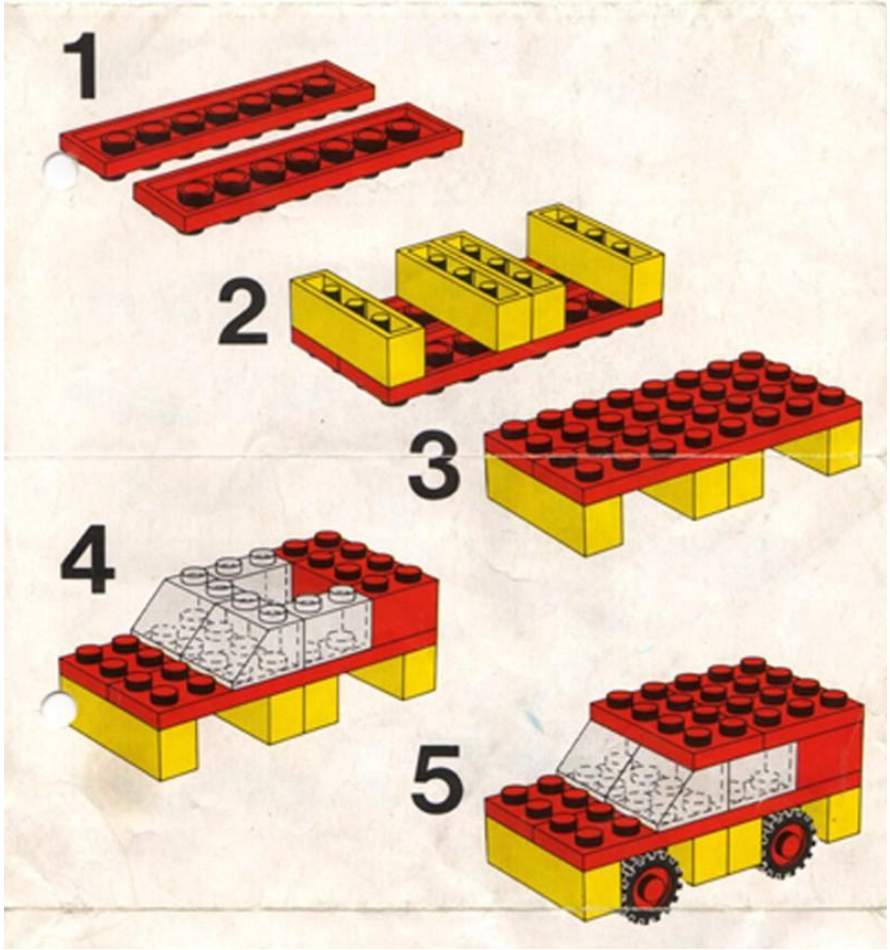
Benefits

- Pipelines encourage composability
- Pipelines follow good design principles
- Pipelines are easier to maintain
- Pipelines make testing easier
- Pipelines fit well with modern architectures
  - E.g. Onion/Clean/Hexagonal, etc

Benefit 1:

Pipelines encourage composability

Composable == “Like Lego”



Connect two pieces together and get another "piece" that can still be connected

You don't need to create a special adapter to make connections.

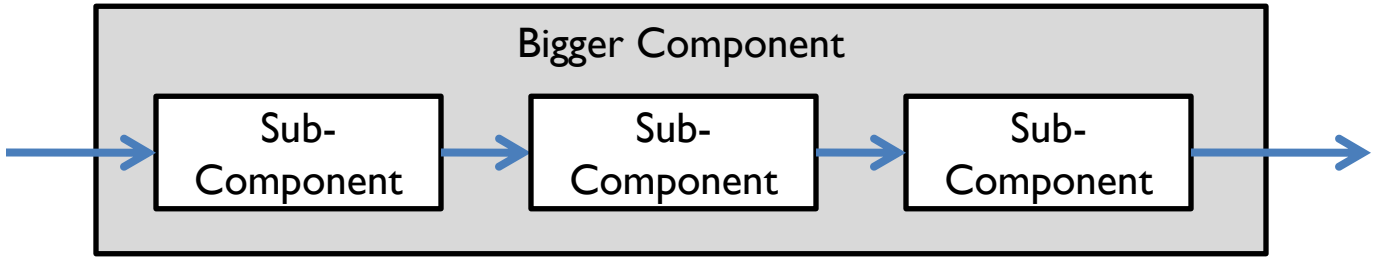
You can keep adding and adding.

Pipelines ❤ composition

Pipelines encourage you to design so that:

- Any pieces can be connected
- You don't need special adapters

If you do this, you get nice composable components



Sub-components composed into a new bigger unit

Can't tell it was built from smaller components!

Benefit 2:

Pipelines follow good design principles

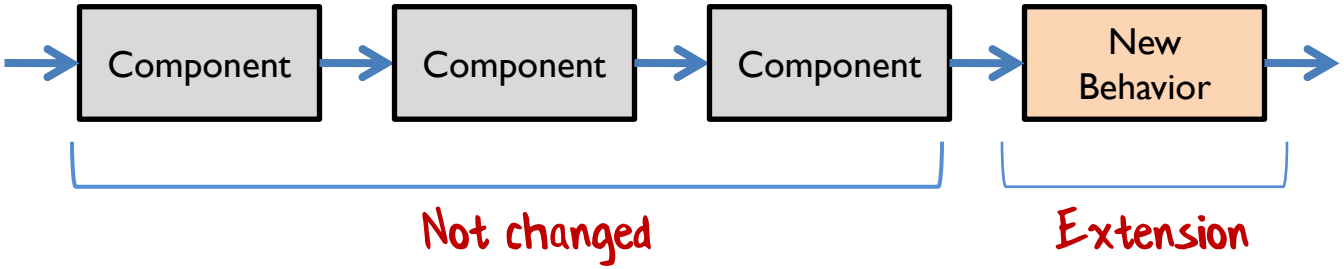
Many design patterns work naturally with a pipeline-oriented approach

Single responsibility principle

Can't get more single responsibility than this!

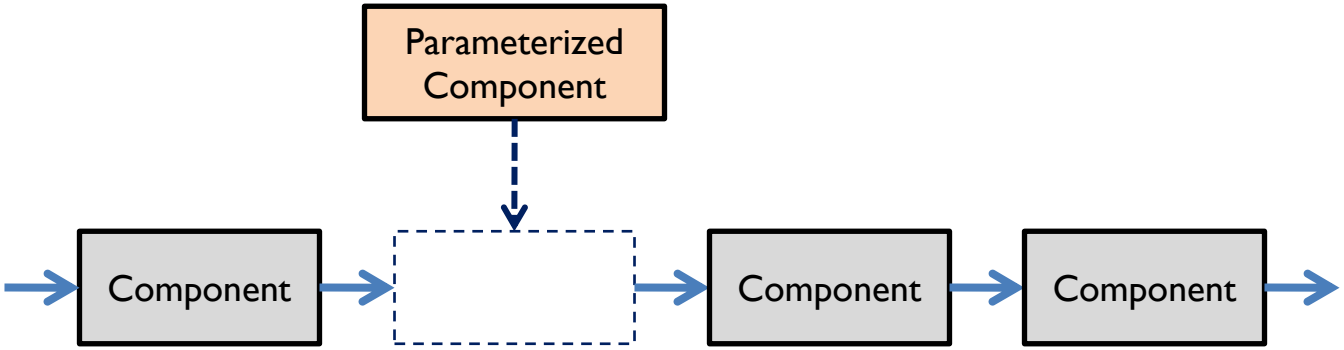
Open/Closed principle

You could be able to add new functionality ("open for extension")
without changing existing code ("closed for modification")

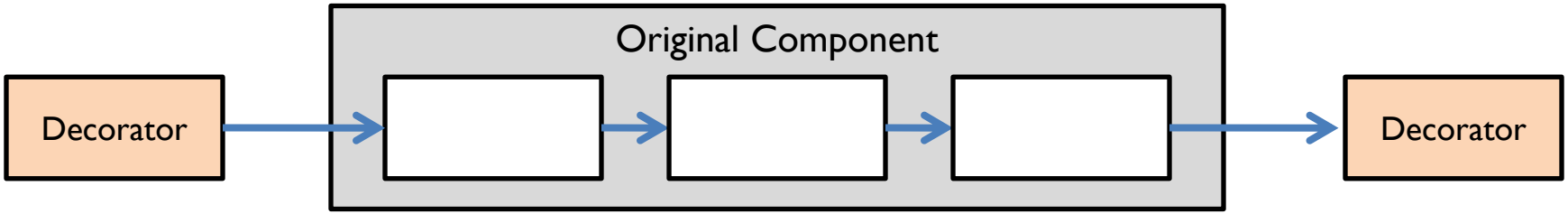


We will see some examples of this later

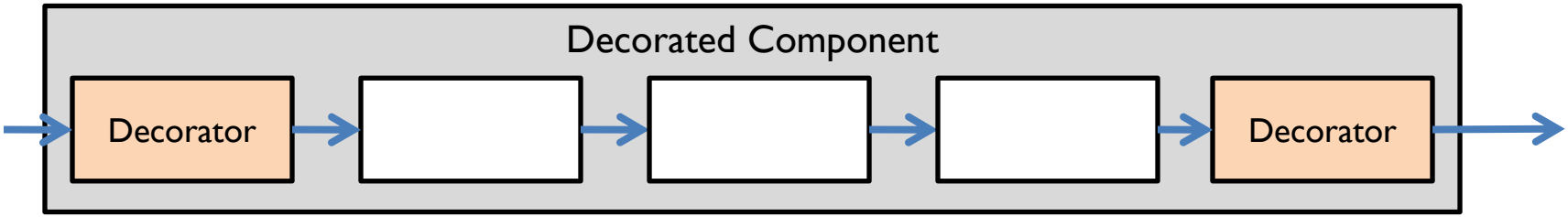
Strategy Pattern



Decorator Pattern



Decorator Pattern

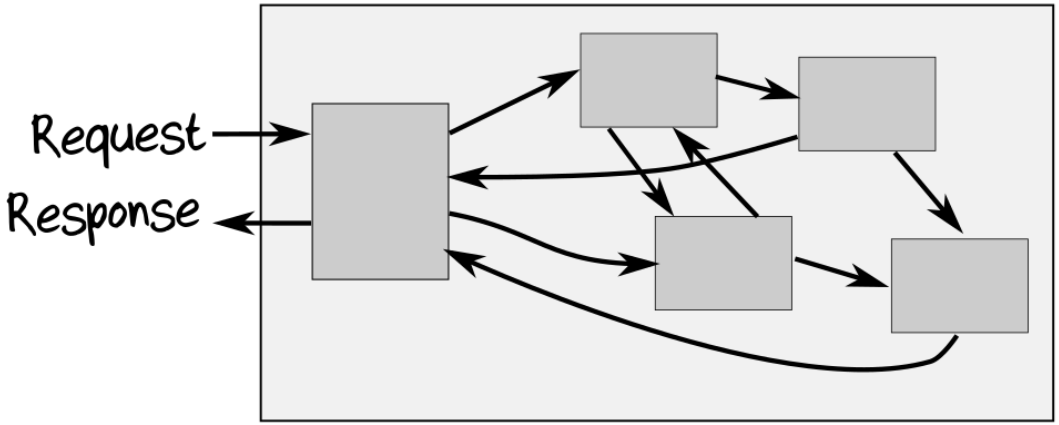


"decorated" component works just like the original one

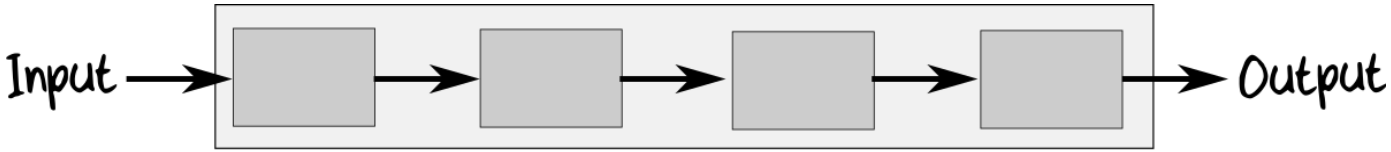
We will see an example of this soon

Benefit 3:

Pipelines are easier to maintain

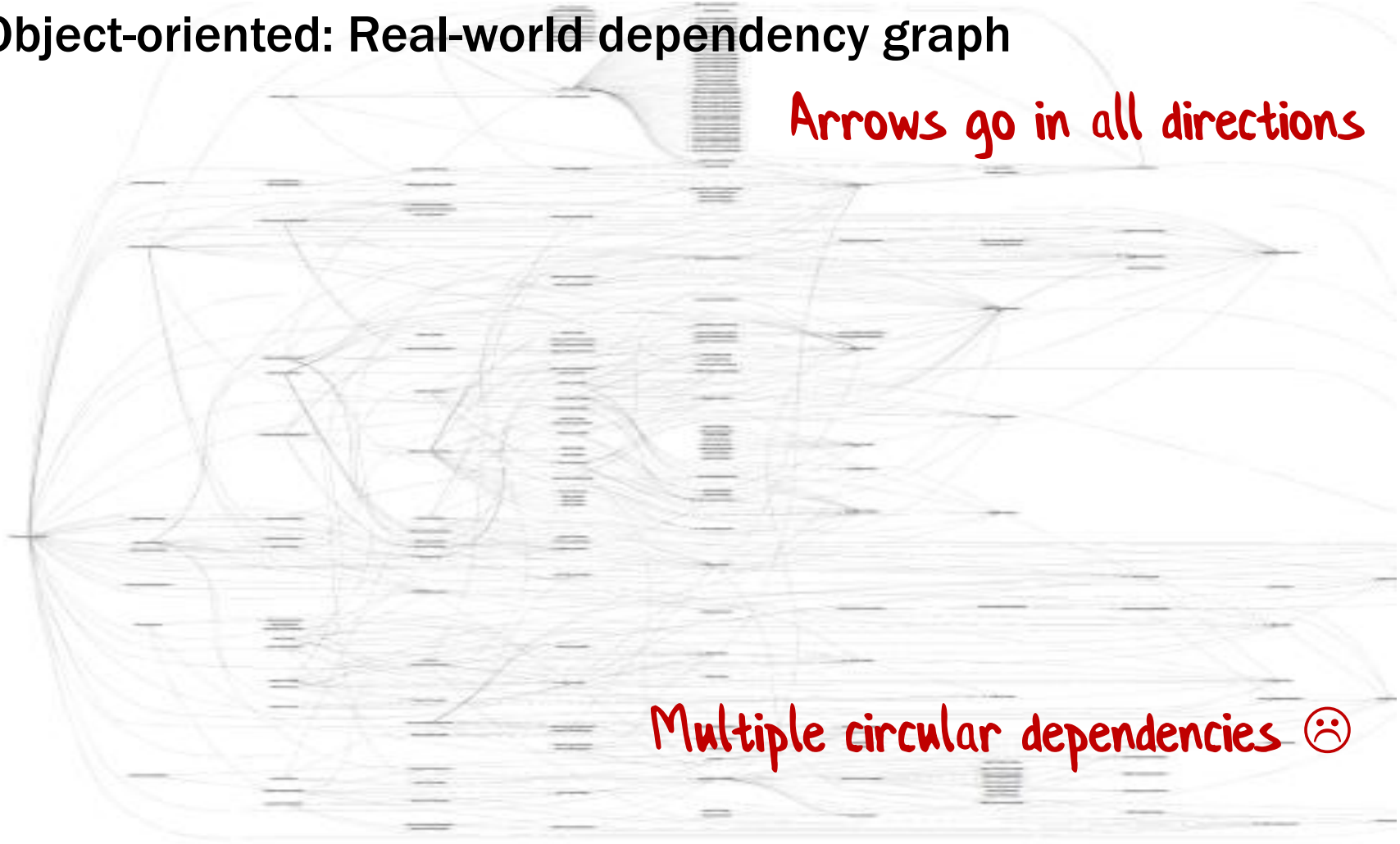


This is harder to maintain



This is easier to maintain

Object-oriented: Real-world dependency graph



Pipeline-oriented: Real-world dependency graph

Same functionality as the OO example

All arrows go left to right ☺

More benefits:

- Pipelines make testing easier
- Pipelines fit well with modern architectures

These will be discussed later on

Part 2

Pipeline-oriented Programming in practice

```
var count = 0;
foreach (var i in list) {
    var j = i + 2;
    if (j > 3)
    {
        count++;
    }
}
return count;
```

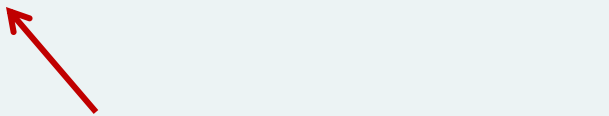
Here is a traditional for- loop in C#

```
return list
    .Select(x => x + 2)
    .Where(x => x > 3)
    .Count();
```

Pipeline-oriented

Here is the same code using LINQ in C#

```
return list
    .Select(x => x + 2)
    .Where(x => x > 3)
    .Count();
```

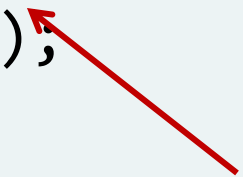


Benefit: Composability

The LINQ components have been designed to fit together in many different ways

You could write your code this way too!

```
return list
    .Select(x => x + 2)
    .Where(x => x > 3)
    .Count();
```



Benefit: Single responsibility

Each LINQ component does one thing only.

Easier to understand and test

```
return list
    .Select(x => x + 2)
    .Where(x => x > 3)
    .Where(x => x < 10)
    .Count();
```

Benefit: open for extension

It is easy to add new steps in the pipeline without touching anything else

```
list
|> List.map (fun x -> x + 2)
|> List.filter (fun x -> x > 3)
|> List.length
```

Here is the same code using F#

```
list
|> List.map (fun x -> x + 2)
|> List.filter (fun x -> x > 3)
|> List.length
```

F# pipe operator

Here is the same code using F#

Immutability and pipelines

If you have immutable data
you will probably need a pipeline

```
// Person is immutable
var p = new Person("Scott","s@example.com",21);
var p2 = p.WithName("Tom");
var p3 = p2.WithEmail("tom@example.com");
var p4 = p3.WithAge(42);
```

Ugly!

When each call returns a new object, it gets repetitive

```
// Person is immutable
var p = new Person("Scott","s@example.com",21);

p
.WithName("Tom")
.WithEmail("tom@example.com")
.WithAge(42);
```

Pipeline

Pipelines make it look nicer

Pipes vs. Extension methods

Pipes are more general

```
int Add1(int input) { return input + 1; }
int Square(int input) { return input * input; }
int Double(int input) { return input * 2; }
```

```
int Add1(int input) { return input + 1; }
int Square(int input) { return input * input; }
int Double(int input) { return input * 2; }

int NestedCalls()
{
    return Double(Square(Add1(5)));
}
```

How can I make this look nicer?

Deeply nested calls

How about a pipeline?

```
return 5
    .Add1
    .Square
    .Double;
```

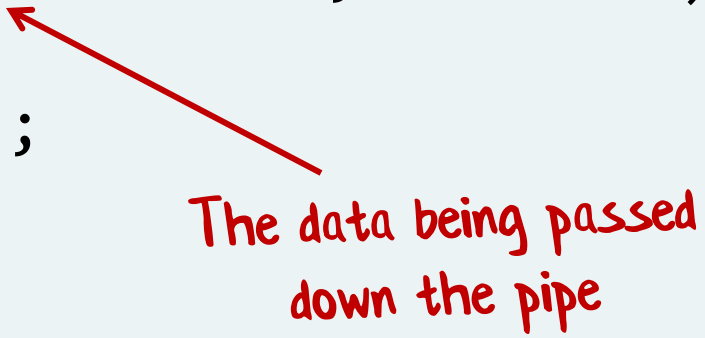
But this doesn't work, because these are not extension methods.

We need a helper function to turn any function into an extension method!

```
public static TOut Pipe<TIn, TOut>
    (this TIn input, Func<TIn, TOut> fn)
{
    return fn(input);
}
```

Introducing "Pipe"

```
public static TOut Pipe<TIn, TOut>
    (this TIn input, Func<TIn, TOut> fn)
{
    return fn(input);
}
```



```
public static TOut Pipe<TIn, TOut>
    (this TIn input, Func<TIn, TOut> fn)
{
    return fn(input);
}
```

Then just call it

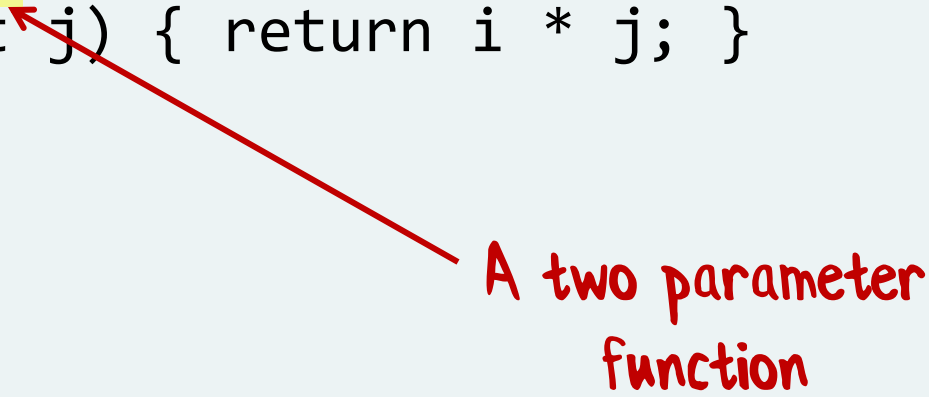
Pass in the method you want to convert to an extension method

```
return 5
    .Pipe(Add1)
    .Pipe(Square)
    .Pipe(Double);
```

And now we can create a pipeline out of any* existing functions

*only works for functions with one input ☹

```
int Add(int i, int j) { return i + j; }
int Times(int i, int j) { return i * j; }
```



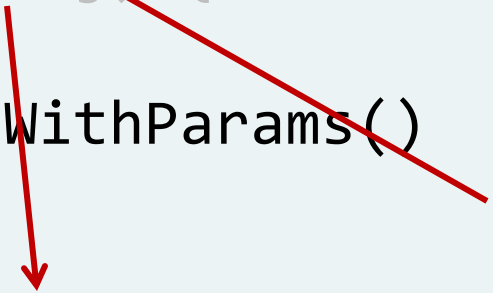
```
public static TOut Pipe<TIn, TParam, TOut>
  (this TIn input, Func<TIn, TParam, TOut> fn, TParam p1)
{
    return fn(input, p1);
}
```

You call the function with the parameter as well

We can create another Pipe method which allows a parameter

```
int Add(int i, int j) { return i + j; }
int Times(int i, int j) { return i * j; }

public int PipelineWithParams()
{
    return 5
        .Pipe(Add, 1)
        .Pipe(Times, 2);
}
```



We can now create a pipeline out of any existing functions with parameters

```
public int PipelineWithParams()
{
    return 5
        .Pipe(Add, 1)
        .Pipe(Times, 2);
}
```

Why bother?

Because now we get all the benefits of a pipeline

```
public int PipelineWithParams()
{
    return 5
        .Pipe(Add, 1)
        .Pipe(Times, 2)
        .Pipe(Add, 42)
        .Pipe(Square);
}
```

Why bother?
Because now we get all the benefits of a pipeline, such as adding things to the pipeline easily

(diffs look nicer too!)

```
let add x y = x + y
let times x y = x * y

5
|> add 1
|> times 2
```

And here's what the same code looks like in F#

F# uses pipelines everywhere!

Pipes vs. Extension methods, Part 2

Extension methods cannot be parameters

```
int StrategyPattern(... list, ... injectedFn) {

    return list
        .Select(x => x + 2)
        .injectedFn
        .Where(x => x > 3)
        .Count();
}
```

My "strategy"

We cant use a function parameter as an extension method

```
int StrategyPattern(... list, ... injectedFn) {

    return list
        .Select(x => x + 2)
        .Pipe(injectedFn)
        .Where(x => x > 3)
        .Count();
}
```

But we can use a function parameter in a pipeline!

Part 3:

Pipelines in practice

Three demonstrations

- Pipeline-oriented Roman Numerals
- Pipeline-oriented FizzBuzz
- Pipeline-oriented web API

Roman Numerals

To Roman Numerals

Task: convert an integer to roman numerals

- 5 => "V"
- 12 => "XII"
- 107 => "CVII"

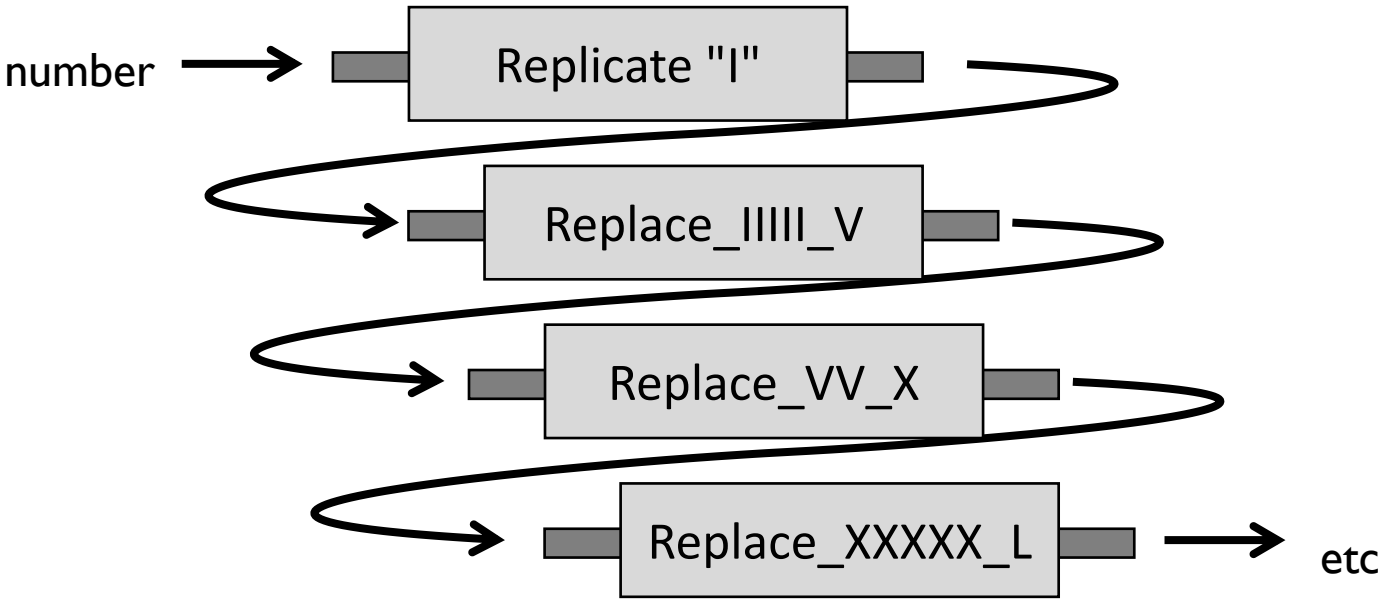
To Roman Numerals

Roman numbers evolved from this

To Roman Numerals

- Use the "tally" approach
  - Start with N copies of "I"
  - Replace five "I"s with a "V"
  - Replace two "V"s with a "X"
  - Replace five "X"s with a "L"
  - Replace two "L"s with a "C"
  - etc

To Roman Numerals



C# example

```
string ToRomanNumerals(int n)
{
    return new String('I', n)
        .Replace("IIIII", "V")
        .Replace("VV", "X")
        .Replace("XXXXX", "L")
        .Replace("LL", "C");
}
```

C# example

```
string ToRomanNumerals(int n)
{
    return new String('I', n)
        .Replace("IIIII", "V")
        .Replace("VV", "X")
        .Replace("XXXXX", "L")
        .Replace("LL", "C")
        .Replace("VIIII", "IX")
        .Replace("IIII", "IV")
        .Replace("LXXXX", "XC")
        .Replace("XXXX", "XL");
}
```

Extend functionality without touching existing code

F# example

```
let toRomanNumerals n =
    String.replicate n "I"
    |> replace "IIIII" "V"
    |> replace "VV" "X"
    |> replace "XXXXX" "L"
    |> replace "LL" "C"
    // special cases
    |> replace "VIIII" "IX"
    |> replace "IIII" "IV"
    |> replace "LXXXX" "XC"
    |> replace "XXXX" "XL"
```

FizzBuzz

FizzBuzz definition

- Write a program that prints the numbers from 1 to N
- But:
  - For multiples of three print "Fizz" instead
  - For multiples of five print "Buzz" instead
  - For multiples of both three and five print "FizzBuzz" instead.

C# example

```
for (var i = 1; i <= 30; i++)
{
    if (i % 15 == 0)
        Console.Write("FizzBuzz,");
    else if (i % 3 == 0)
        Console.Write("Fizz,");
    else if (i % 5 == 0)
        Console.Write("Buzz,");
    else
        Console.Write($"{i},");
}
```

```
for (var i = 1; i <= 30; i++)
{
    if (i % 15 == 0)
        Console.Write("FizzBuzz,");
    else if (i % 3 == 0)
        Console.Write("Fizz,");
    else if (i % 5 == 0)
        Console.Write("Buzz,");
    else
        Console.Write($"{i},");
}
```

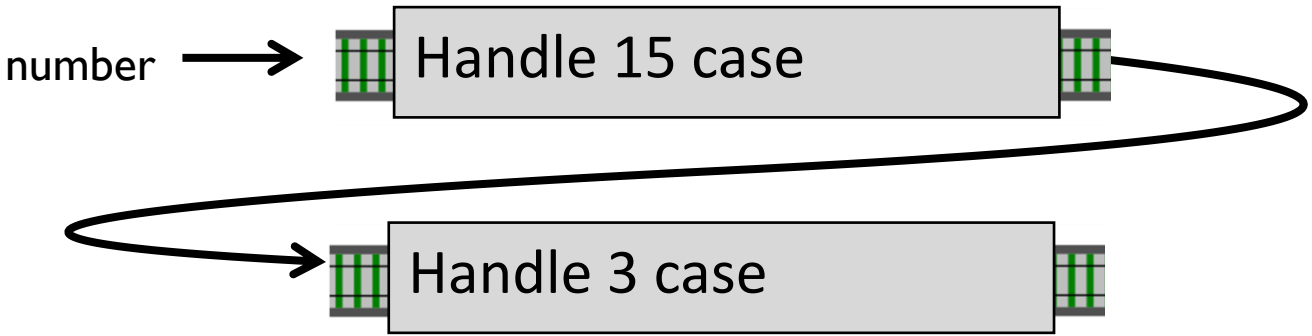
C# example

Too easy!

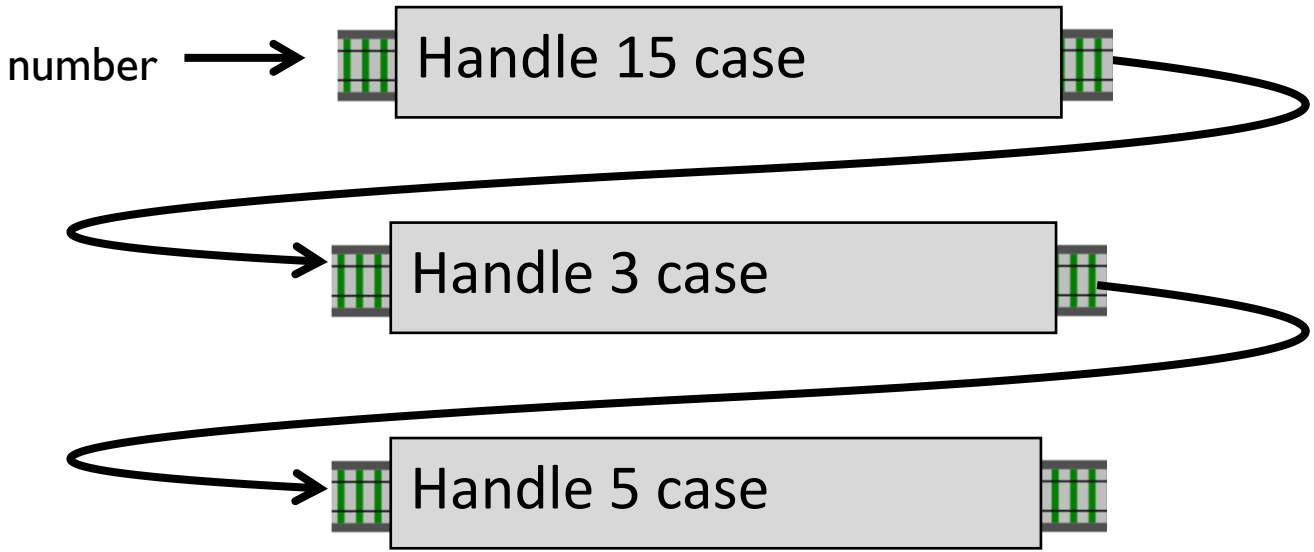
Also, not composable!

Pipeline implementation

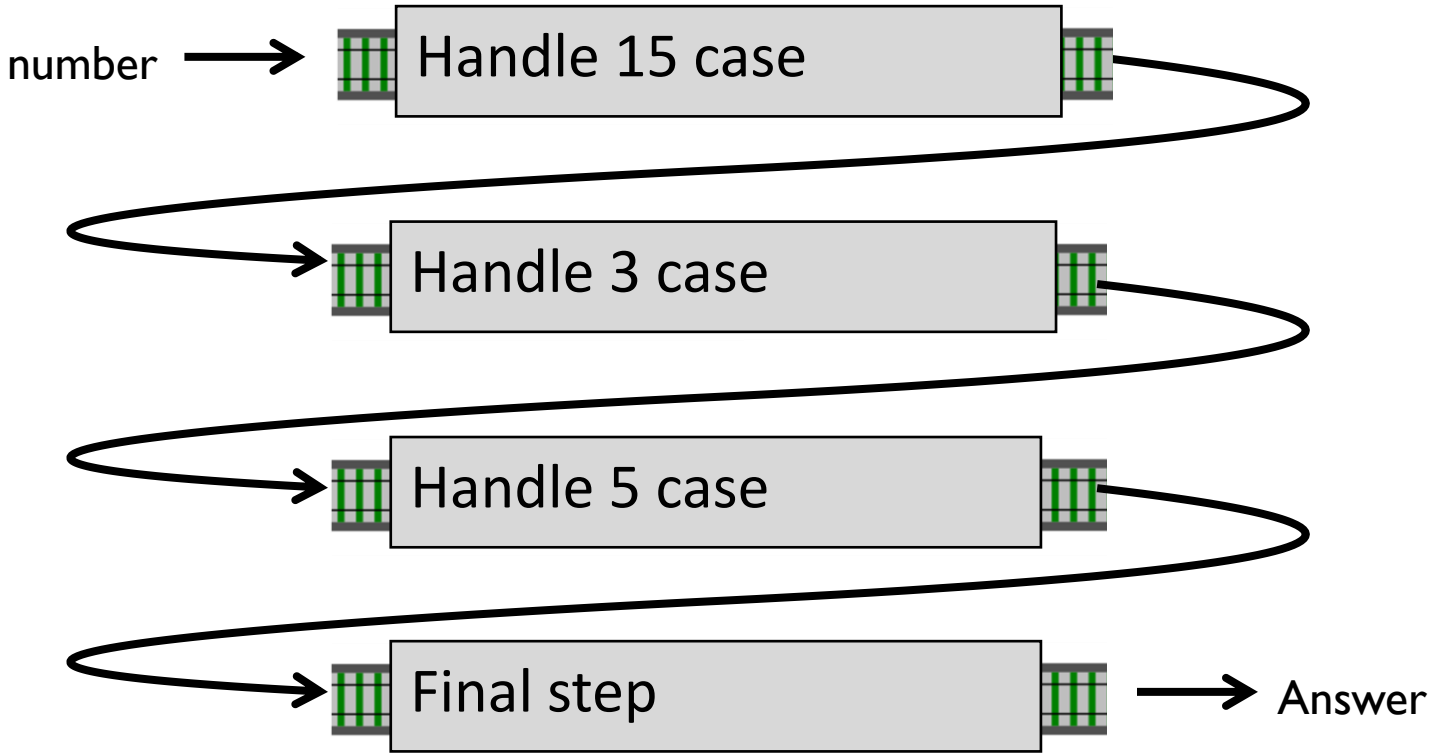
Pipeline implementation

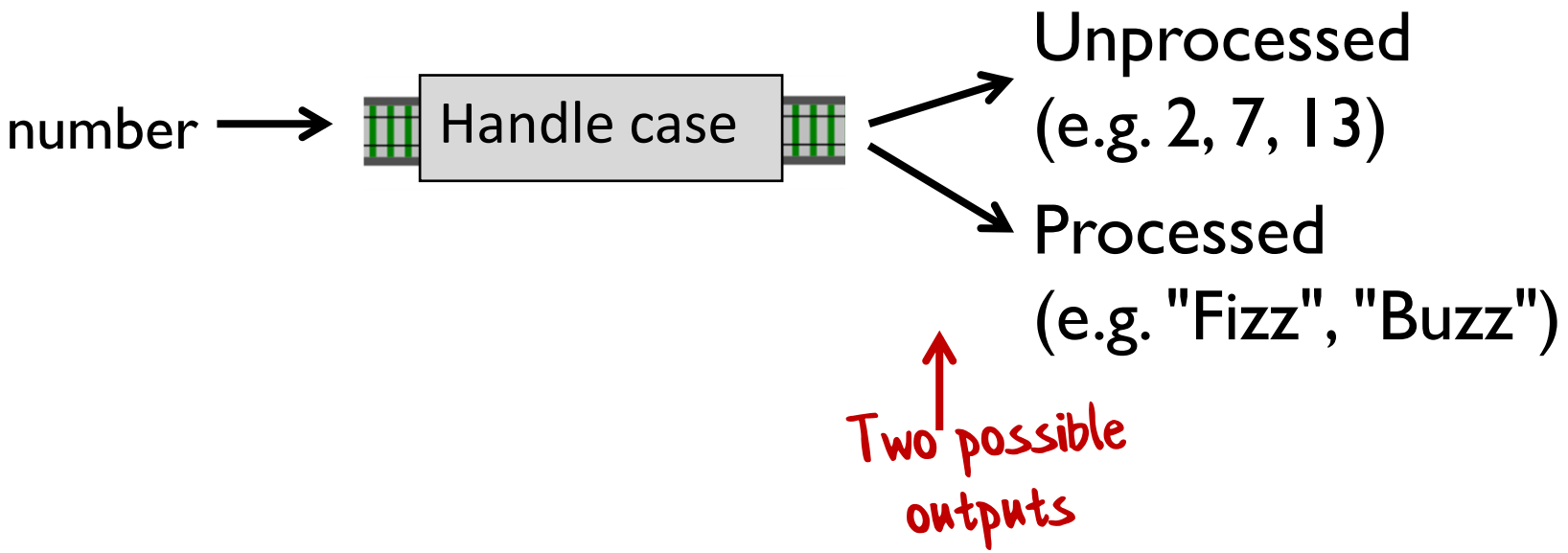


Pipeline implementation



Pipeline implementation






```
record FizzBuzzData(string Output, int Number);
```

```
record FizzBuzzData(string Output, int Number);

static FizzBuzzData Handle(
   this FizzBuzzData data,
   int divisor,        // e.g. 3, 5, etc
   string output)      // e.g. "Fizz", "Buzz", etc
{
    if (data.Output != "")
        return data; // already processed
    if (data.Number % divisor != 0)
        return data; // not applicable
    return new FizzBuzzData(output, data.Number);
}
```

```
record FizzBuzzData(string Output, int Number);

static FizzBuzzData Handle(
    this FizzBuzzData data,
    int divisor,        // e.g. 3, 5, etc
    string output)      // e.g. "Fizz", "Buzz", etc
{
     if (data.Output != "")
         return data; // already processed
     if (data.Number % divisor != 0)
         return data; // not applicable
     return new FizzBuzzData(output, data.Number);
}
```

Extension method

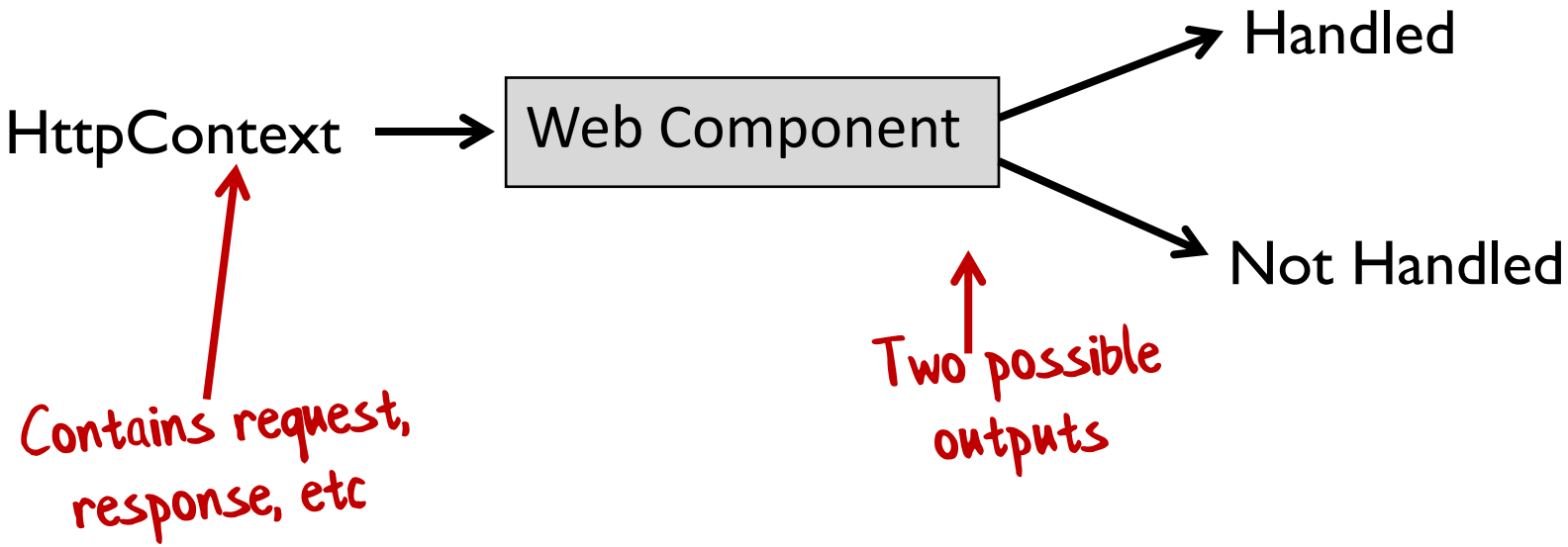
```
static string FizzBuzzPipeline(int i)
{
    return
        new FizzBuzzData("", i)
        .Handle(15, "FizzBuzz")
        .Handle(3, "Fizz")
        .Handle(5, "Buzz")
        .FinalStep();
}
static void FizzBuzz()
{
    var words = Enumerable.Range(1, 30)
        .Select(FizzBuzzPipeline);
    Console.WriteLine(string.Join(",", words));
}
```

Demo:
FizzBuzz in F#

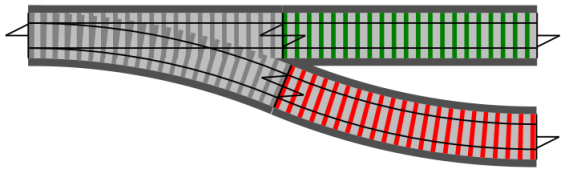
Extensions, Decorations and Parallelization

A pipeline-oriented web API

What does a component in a web pipeline look like?



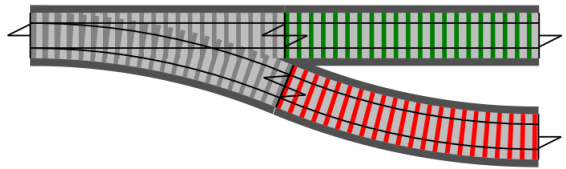
A bit like middleware

HttpContext →  (async) HttpContext

null

Contains request, response, etc

HttpContext →



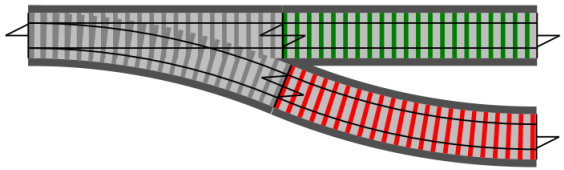
matches verb

doesn't match verb

```
GET
```

Define a component

HttpContext →

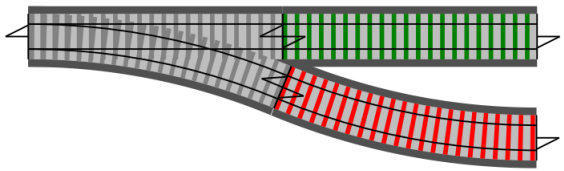


matches route

doesn't match route

```
route "/hello"
```

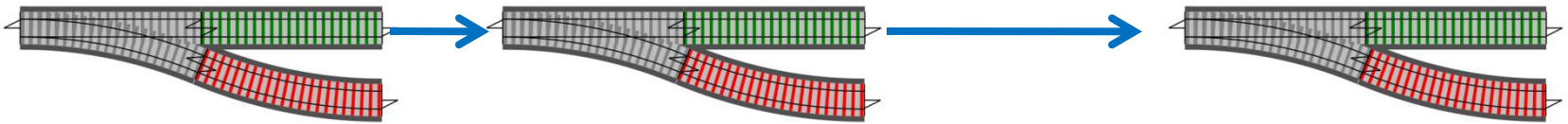
Define a component

HttpContext →  Always succeeds

```
setStatusCode 200
```

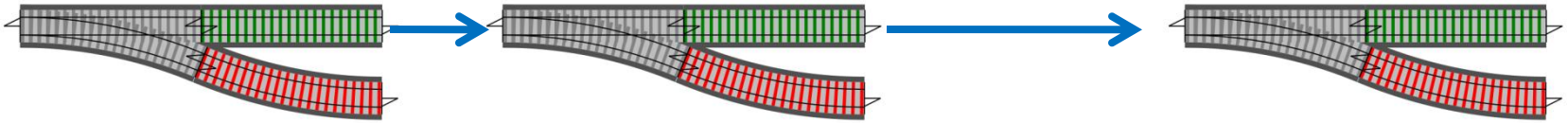
↑

Define a component

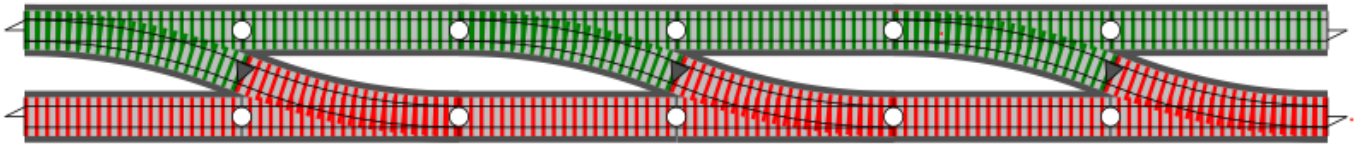


```
GET  >=>  route "/hello"  >=>  setStatusCode 200
```

A special pipe for web components

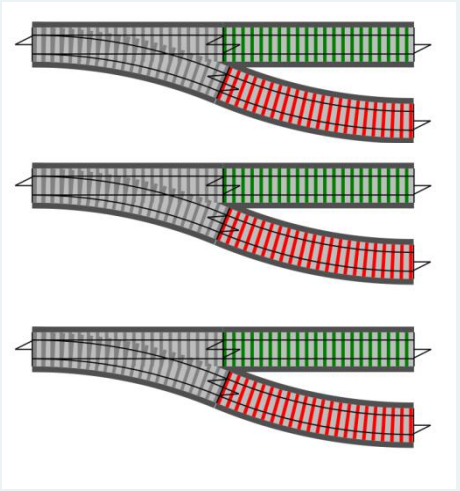


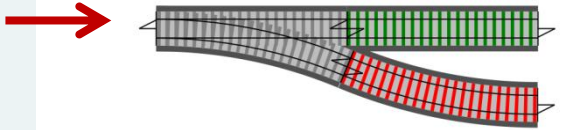
Are you wondering how do we compose these?



Easy!

```
choose [
]
```





Picks first component that succeeds

Define a new component


```
choose [
  GET >=> route "/hello" >=> OK "Hello"
  GET >=> route "/goodbye" >=> OK "Goodbye"
  ]
```

All the benefits of pipeline-oriented programming

```
choose [
  GET >=> route "/hello" >=> OK "Hello"
  GET >=> route "/goodbye" >=> OK "Goodbye"
  POST >=> route "/bad" >=> BAD_REQUEST
  ]
```

All the benefits of pipeline-oriented programming

```
choose [
  GET >=> route "/hello" >=> OK "Hello"
  GET >=> route "/goodbye" >=> OK "Goodbye"
  POST
    >=> route "/user"
    >=> mustBeLoggedIn UNAUTHORIZED
    >=> requiresRole "Admin"
    // etc
  ]
```

All the benefits of pipeline-oriented programming

```
choose [
  GET >=> route "/hello" >=> OK "Hello"
  GET >=> route "/goodbye" >=> OK "Goodbye"
  POST
    >=> route "/user"
    >=> mustBeLoggedIn UNAUTHORIZED
    >=> requiresRole "Admin"
    // etc
  ]
```

The components are composable, reusable, testable, etc.

Demo:
A pipeline oriented web app

For more on the web framework I'm using, search the internet for "F# Giraffe"

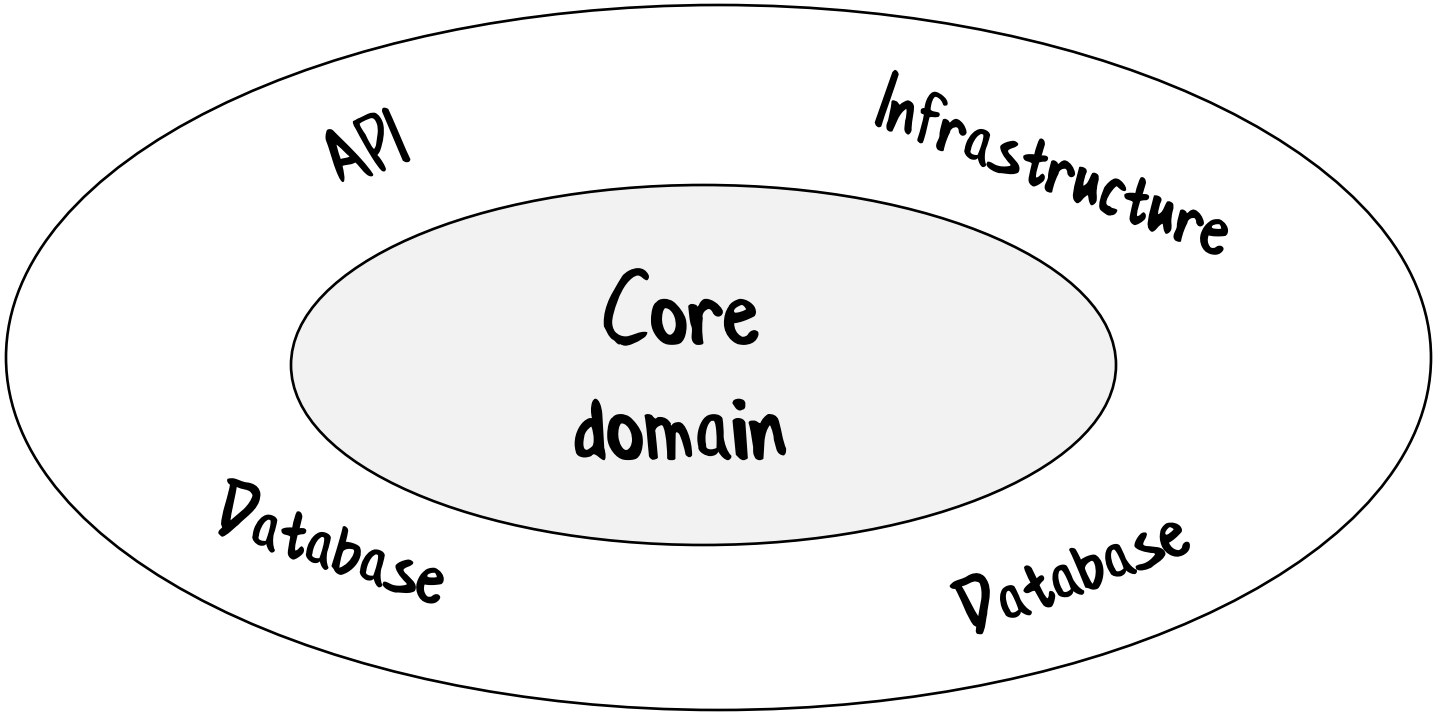
Part 4

Testing and architecture

Benefit 4:

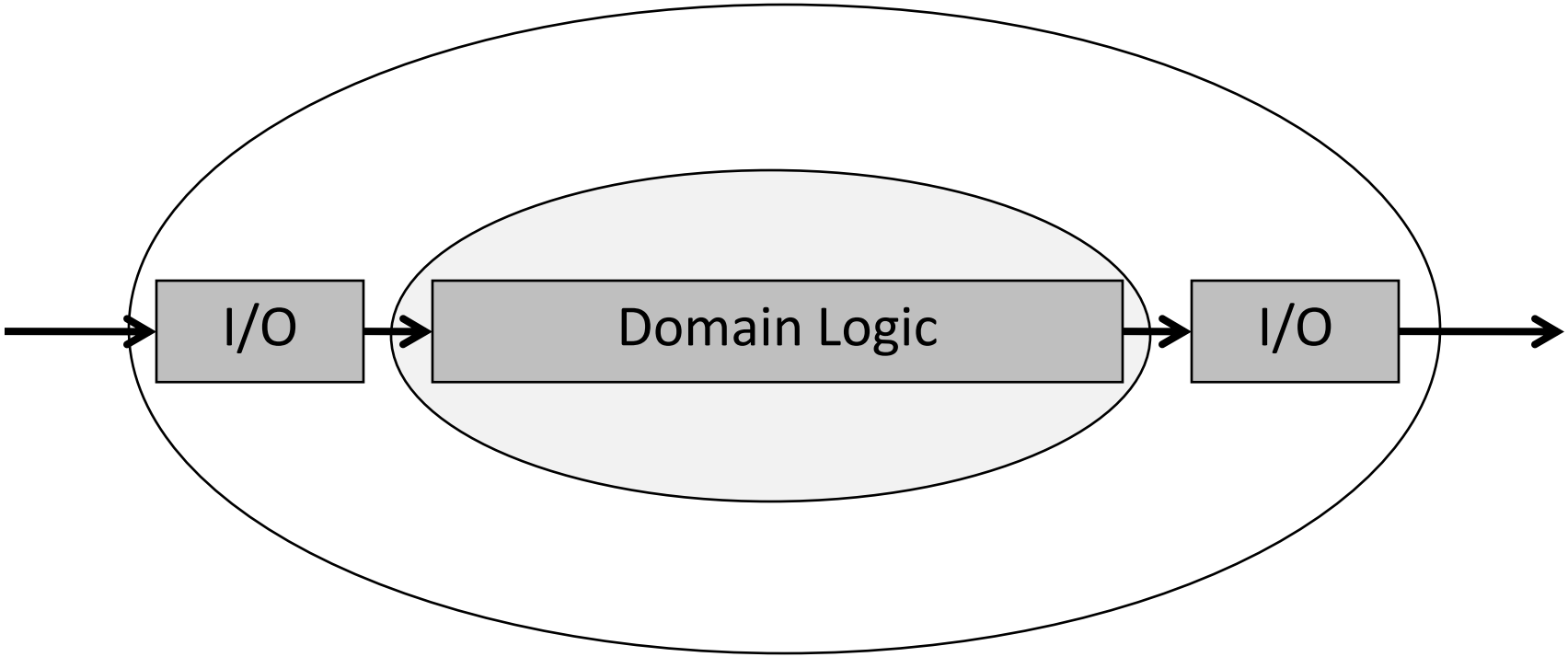
Pipelines encourage good architecture

The "onion" architecture

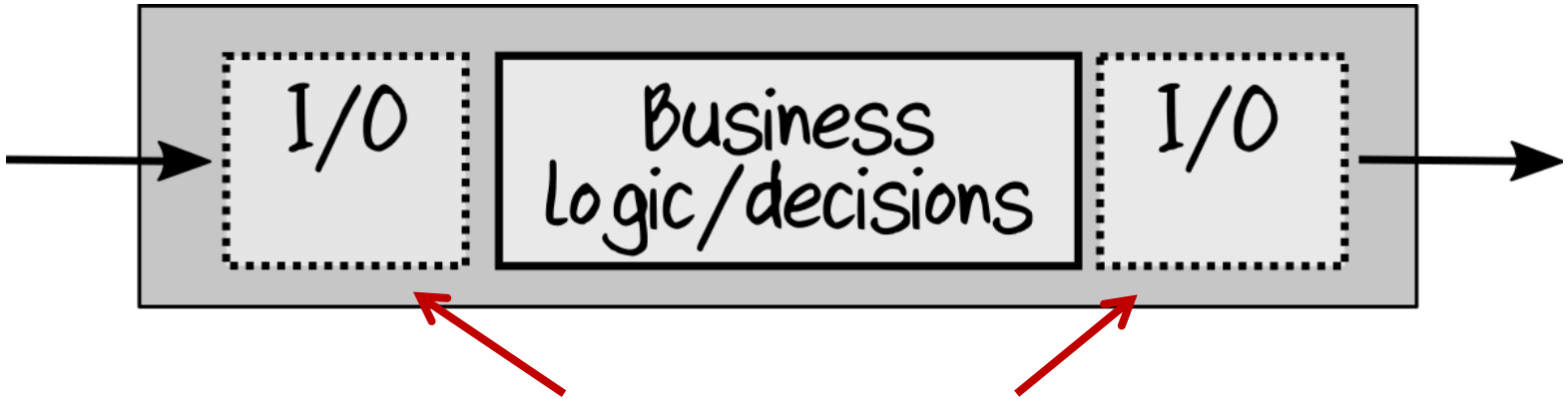


Core domain is pure, and all I/O is at the edges

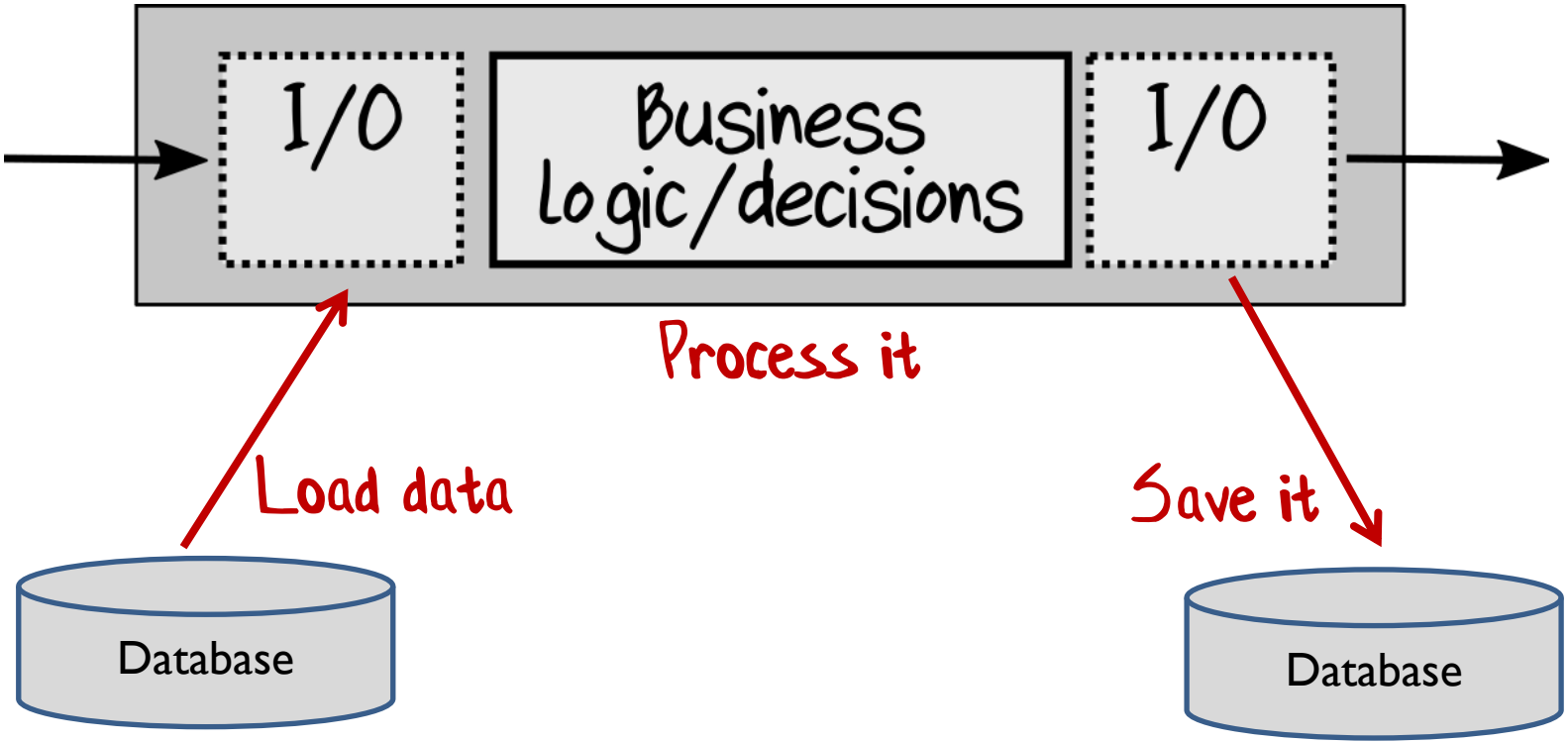
The "onion" architecture



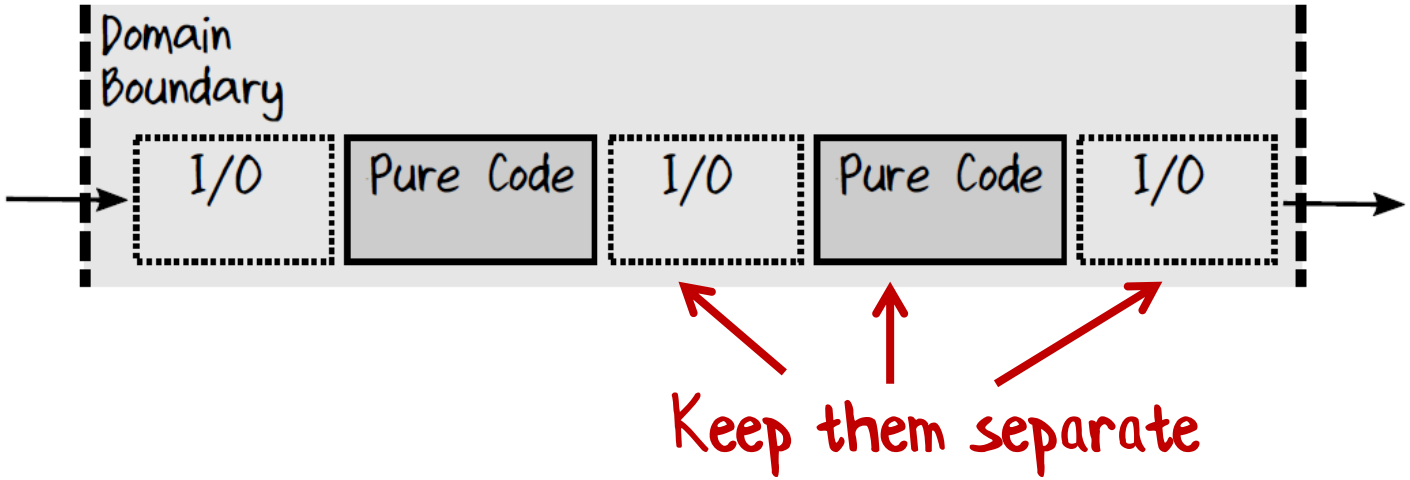
Core domain is pure, and all I/O is at the edges



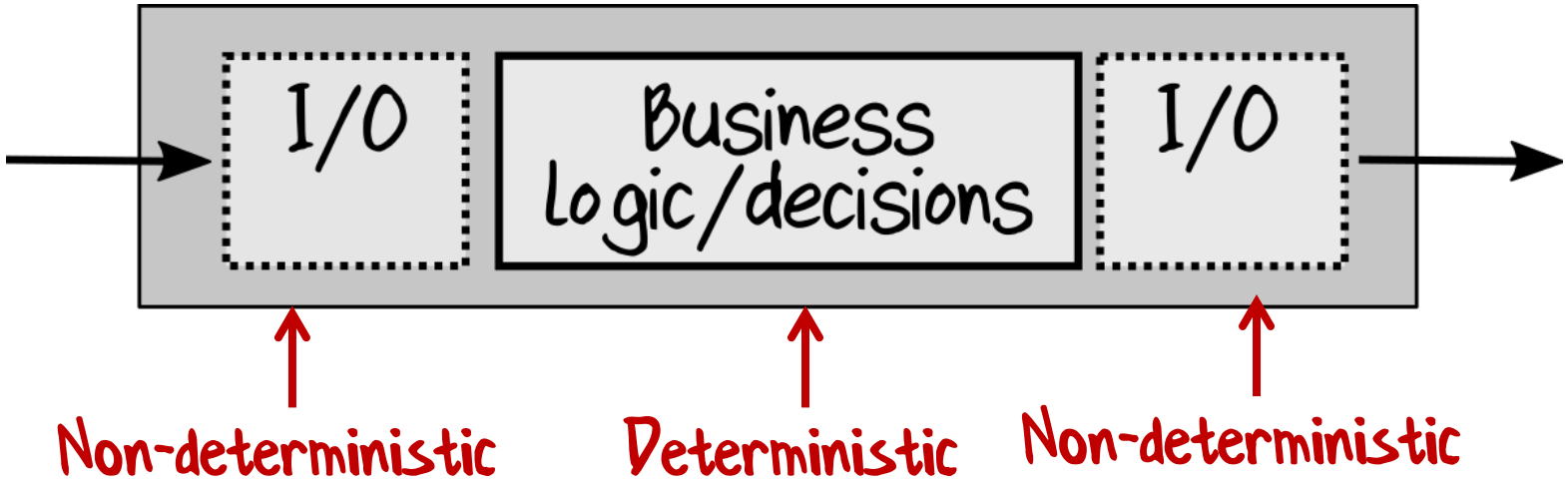
Easy to enforce this with a pipeline-oriented approach


I/O
Business logic/decisions
I/O
Process it
Load data
Save it
Database
Database

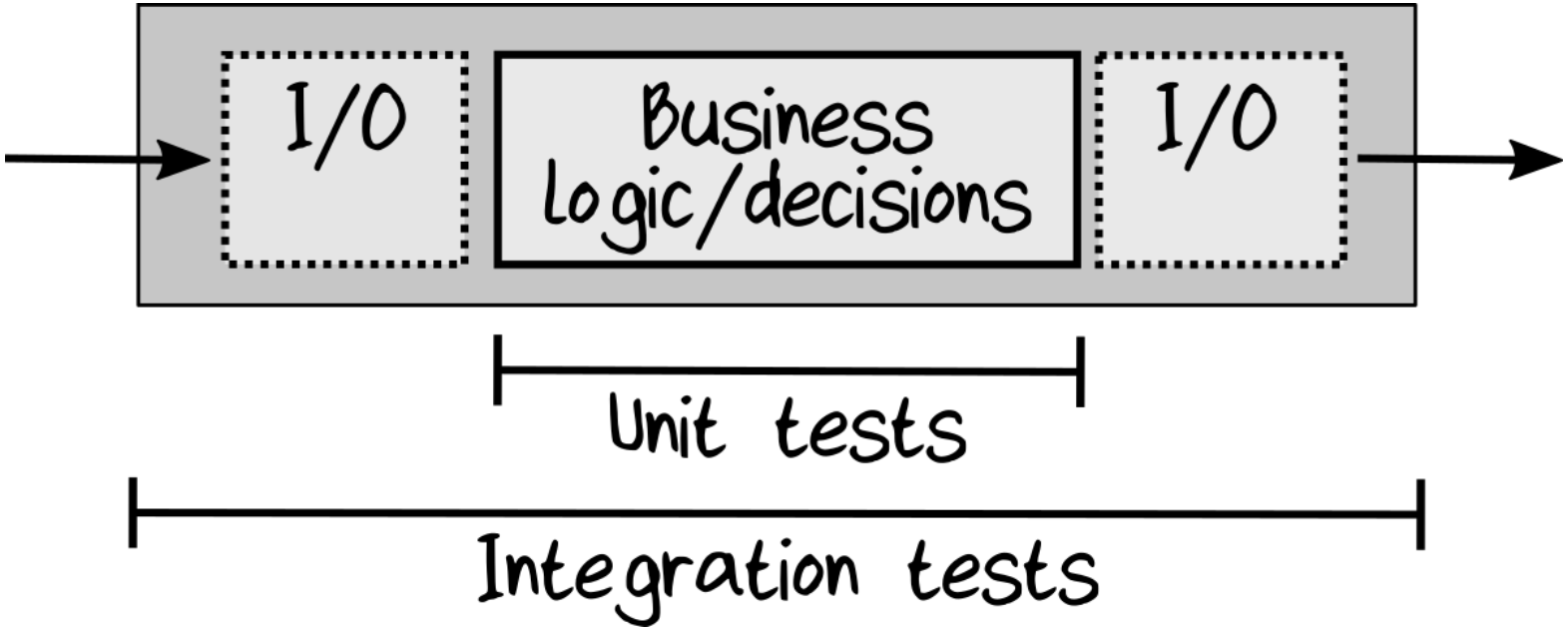
I/O in the middle of a pipeline



Benefit 5:
Easier to test


I/O
Business logic/decisions
I/O
Non-deterministic
Deterministic
Non-deterministic

This makes testing easy!



In Conclusion

Why bother? 

- Reusable components
- Understandable – data flows in one direction
- Extendable – add new parts without touching old code
- Testable – parts can be tested in isolation
- A different way of thinking –
  it's good for your brain to learn new things!

Does pipeline-oriented programming work for every situation? No!

But it should be part of your toolkit

Databases

Domain-driven design

Object-oriented programmng

Pipeline-oriented programming

Pipeline Oriented Programming

- Slides and video will be posted at
  - fsharpforfunandprofit.com/pipeline

Related talks

- "The Power of Composition"
  - fsharpforfunandprofit.com/composition
- "Railway Oriented Programming"
  - fsharpforfunandprofit.com/rop

Twitter: @ScottWlaschin

Thanks!