

JĘZYKI DOMENOWE I FLUENT INTERFACES

„Programy są przeznaczone do czytania przez ludzi, a tylko przypadkowo mogą być wykonywane przez komputer”

Donald Knuth

Języki domenowe

Języki domenowe (ang. Domain Specific Languages)
– języki programowania do rozwiązywania konkretnego problemu

Ograniczony zakres

Rozwiązują konkretny problem domenowy w bardzo dobry sposób

Języki domenowe



Który fragment jest bardziej czytelny?

```
var results =  
collection  
    .Where(i=>i.Age > 18)  
    .OrderBy(i=>i.FirstName)  
    .ThenBy(i=>i.LastName)  
    .ToList()
```

```
var results = new List<Person>()  
  
foreach(var item in collection) {  
    if(item.Age > 18)  
        results.Add(item);  
}  
  
results.Sort(ComparePersons);
```

Języki domenowe - zewnętrzne

Języki domenowe zewnętrzne są niezależnymi językami posiadającymi własną składnię i semantykę.

Przykładami są: CSS, SQL

Języki domenowe - wewnętrzne

Języki domenowe wewnętrzne (osadzone) są to specjalnie przygotowane API w określonym języku ogólnego przeznaczenia wykorzystujące jego składnię i semantykę. Często nazywane Fluent API

Fluent API

„ Fluent Interface jest metodą dla konstruowania API aplikacji, w taki sposób, aby czytelność kodu była jak najbardziej zbliżona do języka naturalnego. ”

Martin Fowler

Fluent API w języku C#

- Metody rozszerzające
- Łączenie metod (method chaining)
- Fluent interfaces

Metody rozszerzające, a DSL

```
public void Metoda() {  
    TimeSpan jednaGodzina = new TimeSpan(1, 0, 0);  
}
```

Metody rozszerzające, a DSL

```
static class Extensions {  
    public static TimeSpan Godzina(this int value) {  
        return new TimeSpan(value, 0, 0);  
    }  
}
```

```
public void Metoda() {  
    TimeSpan jednaGodzina = 1.Godzina();  
}
```

Łączenie metod (method chaining)

Pozwala na wielokrotne wywoływanie różnych metod bez konieczności wykorzystania zmiennych do przechowywania wyników pośrednich.

Łączenie metod

```
class Bus {  
    List<Passenger> _passengers;  
    List<Beacon> _beacons;  
  
    public void AddPassenger(Passenger p) {  
        _passengers.Add(p);  
    }  
  
    public void SetBeacon(Beacon b) {  
        _beacons.Add(b);  
    }  
    ...  
}
```

Łączenie metod

```
var bus = new Bus();  
bus.AddPassenger(new Passenger());  
bus.AddPassenger(new Passenger());  
bus.SetBeacon(new Beacon());  
bus.SetBeacon(new Beacon());  
bus.CloseDoor();  
bus.TurnOnBeacons();
```

Łączenie metod

```
class Bus {  
    List<Passenger> _passengers;  
    List<Beacon> _beacons;  
  
    public Bus AddPassenger(Passenger p) {  
        _passengers.Add(p);  
        return this;  
    }  
  
    public Bus SetBeacon(Beacon b) {  
        _beacons.Add(b);  
        return this;  
    }  
}
```

Łączenie metod

```
var bus =  
    new Bus()  
        .AddPassenger(new Passenger())  
        .AddPassenger(new Passenger())  
        .SetBeacon(new Beacon())  
        .SetBeacon(new Beacon())  
        .CloseDoor()  
        .TurnOnBeacons();
```


Łączenie metod

```
var results =  
collection  
    .where(i=>i.Age > 18)  
    .orderBy(i=>i.FirstName)  
    .thenBy(i=>i.LastName)
```

Łączenie metod w jQuery

```
$(function () {  
    $('#lista').  
        .find('> li')  
        .filter(':first').addClass('specjalny').end()  
        .find('ul')  
        .css('border', '1px solid red')  
        .find('li:last').addClass('specjalny').end()  
        .end()  
    .end()  
    .find('li')  
        .append('każdy li');  
});
```

Kontekst

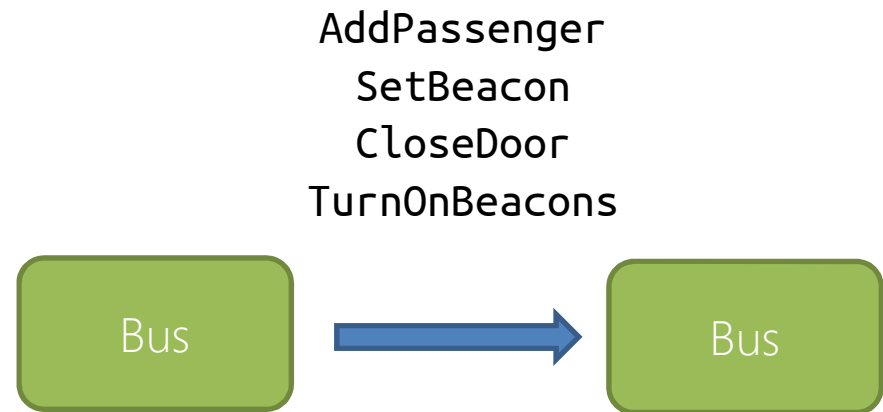
Kontekst jest to obiekt, który ułatwia łączenie metod oraz przechowuje informacje z poprzednich wywołań

Proste wywołanie

Proste wywołanie, jest to wywołanie, które zwraca kontekst tego samego typu co obiekt na którym zostało wykonane

Proste wywołanie

```
var bus =  
    new Bus()  
        .AddPassenger(new Passenger())  
        .AddPassenger(new Passenger())  
        .SetBeacon(new Beacon())  
        .SetBeacon(new Beacon())  
        .CloseDoor()  
        .TurnOnBeacons();
```



Proste wywołanie

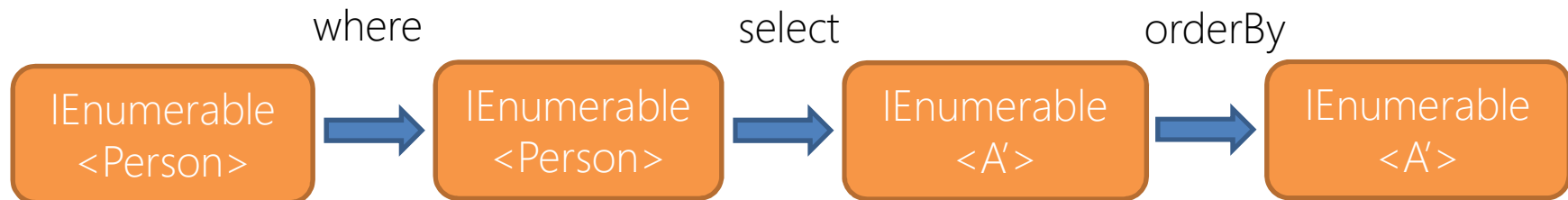
```
var results =  
collection  
    .where(i=>i.Age > 18)  
    .orderBy(i=>i.FirstName)  
    .thenBy(i=>i.LastName)
```

Złożone wywołanie

Złożone wywołanie, jest to wywołanie, które zwraca kontekst innego typu co obiekt na którym zostało wykonane

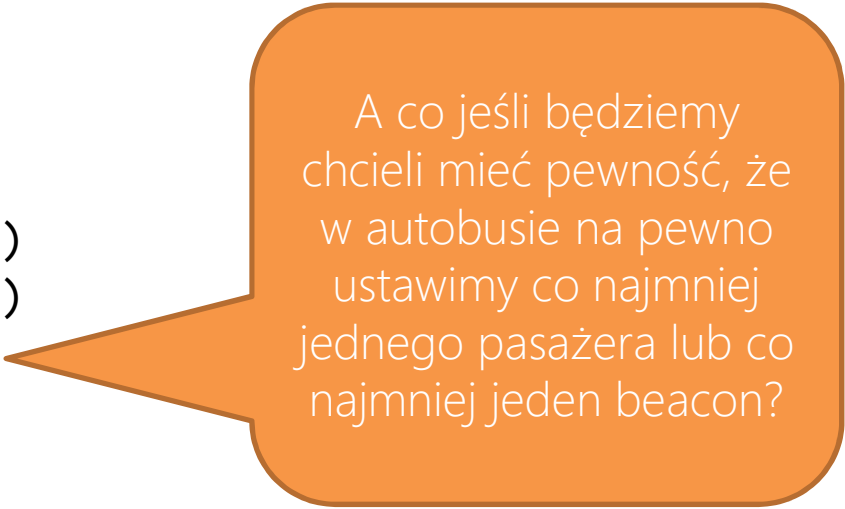
Złożone wywołanie

```
var results =  
collection  
    .where(i=>i.Age > 18)  
    .select(i=> new {Imie = i.FirstName, Nazwisko = i.LastName})  
    .orderBy(p=>p.Imie);
```



Autobus ponownie

```
var bus =  
    new Bus()  
        .AddPassenger(new Passenger())  
        .AddPassenger(new Passenger())  
        .SetBeacon(new Beacon())  
        .SetBeacon(new Beacon())  
        .CloseDoor()  
        .TurnOnBeacons();
```



A co jeśli będziemy chcieli mieć pewność, że w autobusie na pewno ustawimy co najmniej jednego pasażera lub co najmniej jeden beacon?

Autobus ponownie

```
var bus =  
    new Bus(  
        new List<Passenger>() {  
            new Passenger(),  
            new Passenger(),  
        },  
        new List<Beacon>() {  
            new Beacon(),  
            new Beacon()  
        })  
        .CloseDoor()  
        .TurnOnBeacons();
```



Kod poprawny, ale czy ładny?

Autobus ponownie – nazwane parametry

```
var bus =  
    new Bus(  
        passengers: new List<Passenger>() {  
            new Passenger(),  
            new Passenger(),  
        },  
        beacons: new List<Beacon>() {  
            new Beacon(),  
            new Beacon()  
        })  
        .CloseDoor()  
        .TurnOnBeacons();
```



Nazwane parametry

Autobus ponownie - fluent

```
class Bus {  
    ...  
    protected Bus() { ... }  
    public static Bus WithPassenger(Passenger p) {  
        var bus = new Bus();  
        bus.AddPassenger(p);  
        return bus;  
    }  
    public static Bus WithBeacon(Beacon b) {  
        var bus = new Bus();  
        bus.SetBeacon(b);  
        return bus;  
    }  
}
```

Autobus ponownie - fluent

```
var bus = new Bus();
```



Błąd. Konstruktor jest protected.

```
var bus = Bus.WithBeacon(new Beacon())  
    .SetBeacon(new Beacon())  
    .AddPassenger(new Passenger());
```

Builder

```
class BusBuilder {  
  
    public BusBuilder(Bus bus) { _bus = bus; }  
  
    public BusBuilder SetBeacon(Beacon b)  
        { _bus.SetBeacon(b); return this;}  
  
    public BusBuilder AddPassenger(Passenger p)  
        { _bus.AddPassenger(p); return this; }  
  
    public Bus Get() {  
        return _bus;  
    }  
}
```

Builder

```
class Bus {  
    ...  
    protected Bus() { ... }  
  
    public static BusBuilder Create() {  
        var bus = new Bus();  
        return new BusBuilder(bus);  
    }  
}
```

Builder

```
Bus.Create()  
    .AddPassenger(new Passenger())  
    .SetBeacon(new Beacon())  
    .AddPassenger(new Passenger())  
    .SetBeacon(new Beacon())  
    .Get();
```


Builder

```
class BusBuilder {
```

```
    ...  
    public Bus Get => _bus;  
}
```

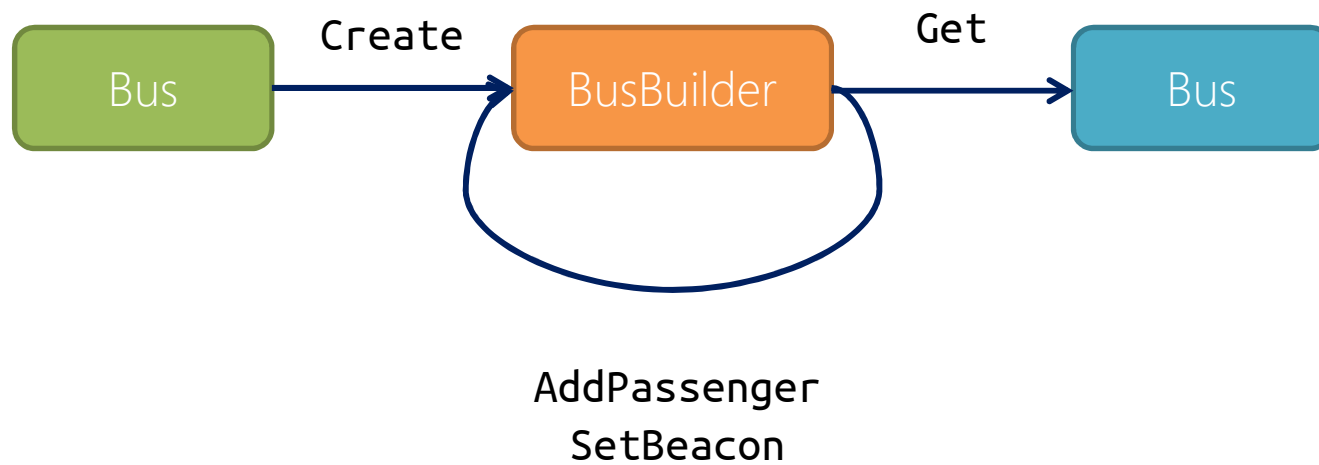
```
class Bus {
```

```
    ...  
    public static BusBuilder Create =>  
        new BusBuilder(new Bus());  
}
```

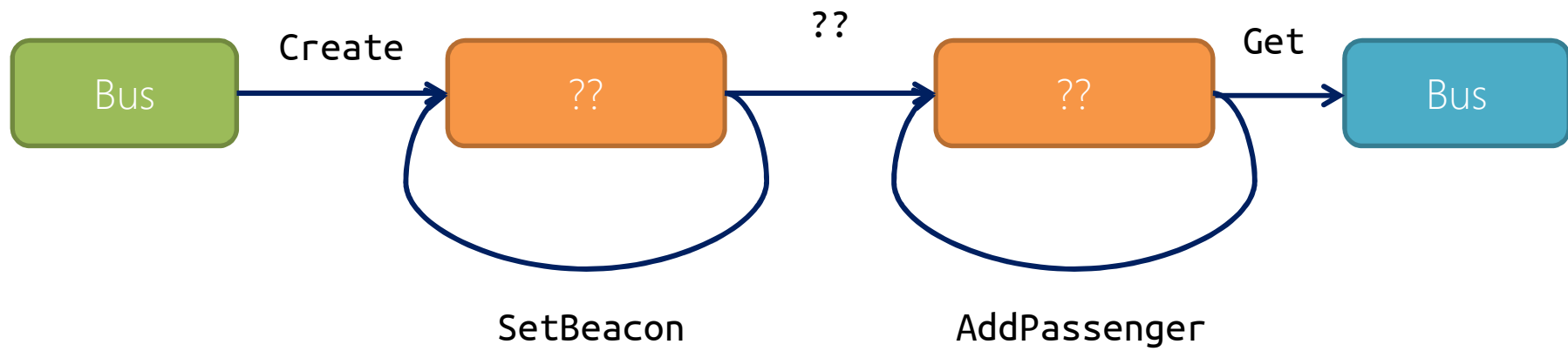
Builder

```
Bus.Create  
    .AddPassenger(new Passenger())  
    .SetBeacon(new Beacon())  
    .AddPassenger(new Passenger())  
    .SetBeacon(new Beacon())  
    .Get;
```

Builder



Builder



Fluent interfaces

Wykorzystanie interfejsów do stworzenia Fluent API

Fluent interfaces

```
interface IBusBuilder_Beacons
{
    IBusBuilder_Beacons SetBeacon(Beacon b);
    IBusBuilder_Passengers AddPassenger(Passenger p);
}

interface IBusBuilder_Passengers
{
    IBusBuilder_Passengers AddPassenger(Passenger p);
    Bus Get { get; }
}
```

Fluent interfaces

```
class BusBuilder : IBusBuilder_Passengers, IBusBuilder_Beacons
{
    private Bus _bus;

    public BusBuilder(Bus bus) { _bus = bus; }

    public IBusBuilder_Beacons SetBeacon(Beacon b)
    { _bus.SetBeacon(b); return this; }

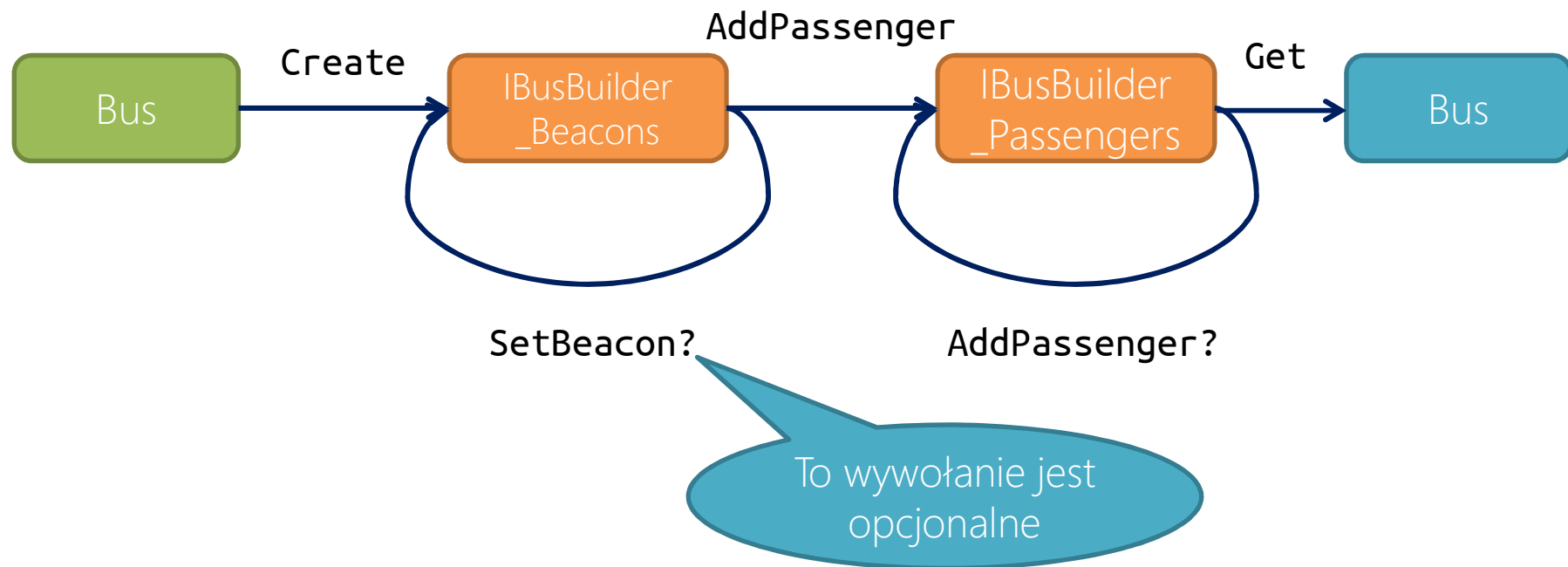
    public IBusBuilder_Passengers AddPassenger(Passenger p)
    { _bus.AddPassenger(p); return this; }

    public Bus Get => _bus;
}
```

Fluent interfaces

```
class Bus {  
    ...  
    public static IBusBuilder_Beacons Create  
        => new BusBuilder(new Bus());  
}
```


Fluent interfaces



Fluent interfaces

```
interface IBusBuilder_Beacons1
{
    IBusBuilder_Beacons2 SetBeacon(Beacon b);
}

interface IBusBuilder_Beacons2
{
    IBusBuilder_Beacons2 SetBeacon(Beacon b);
    IBusBuilder_Passengers AddPassenger(Passenger p);
}
```

Fluent interfaces

```
class BusBuilder
    : IBusBuilder_Passengers, IBusBuilder_Beacons1, IBusBuilder_Beacons2
{
    private Bus _bus;

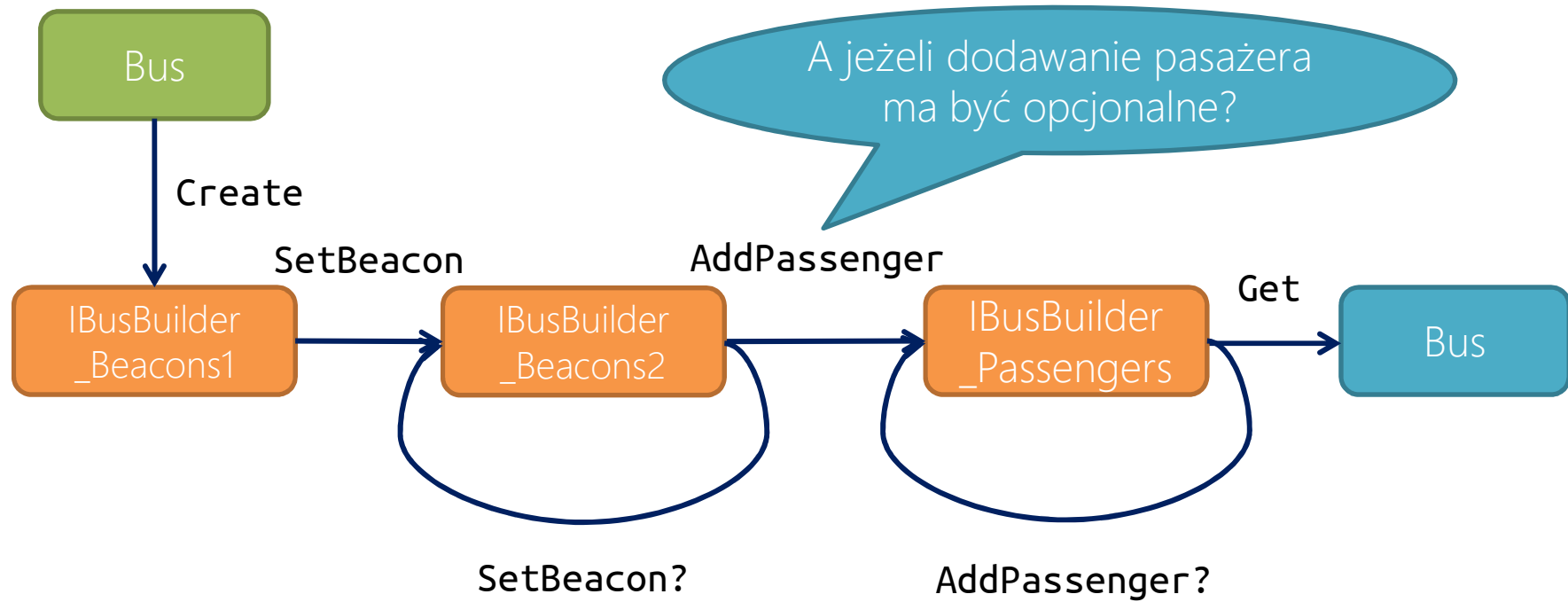
    public BusBuilder(Bus bus) { _bus = bus; }

    public IBusBuilder_Beacons2 SetBeacon(Beacon b)
    { _bus.SetBeacon(b); return this; }

    public IBusBuilder_Passengers AddPassenger(Passenger p)
    { _bus.AddPassenger(p); return this; }

    public Bus Get => _bus;
}
```

Fluent interfaces

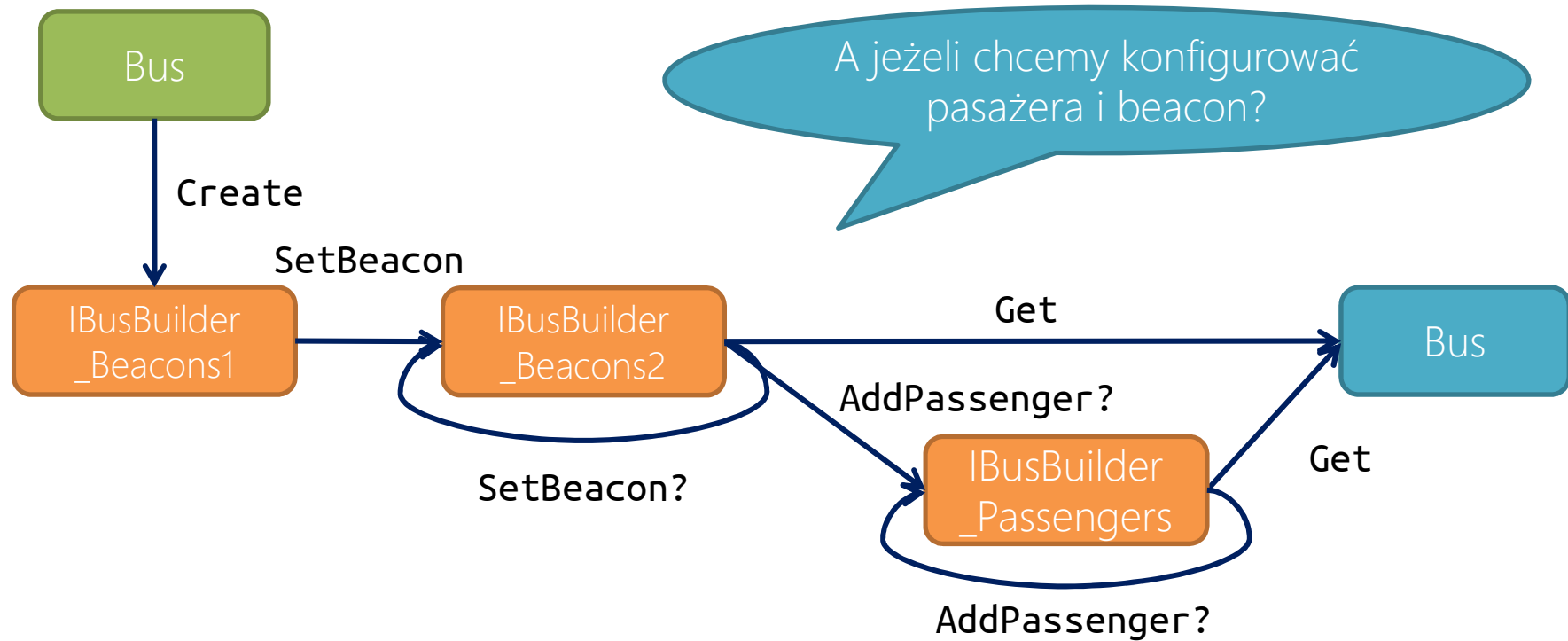


Fluent interfaces

```
interface IBusBuilder_Beacons1
{
    IBusBuilder_Beacons2 SetBeacon(Beacon b);
}

interface IBusBuilder_Beacons2
{
    IBusBuilder_Beacons2 SetBeacon(Beacon b);
    IBusBuilder_Passengers AddPassenger(Passenger p);
    Bus Get { get; }
}
```

Fluent interfaces



Fluent interfaces

