



Wstrzykiwanie Zależności w Praktyce

Bogdan Polak - OEC Connection



github.com/bogdanpolak



linkedin.com/in/bogdanpolak

O mnie

OEConnection, Team Leader, **Kraków/Warszawa**, C# & React & Azure & SCRUM

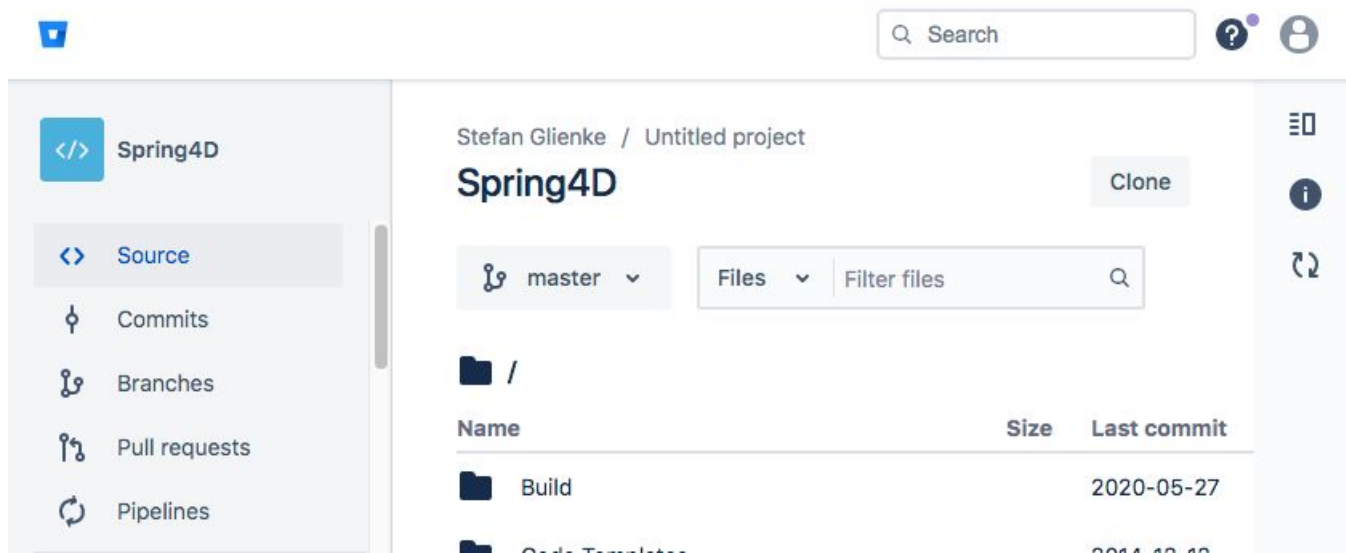
Craneware, Senior Software Engineer, **Edinburgh**, Delphi & C#

BSC Polska, Trainer/Consultant/Community Activist, **Warszawa**, Delphi, C++,
TestComplete, StarTeam,

Microgeo, Entrepreneur / Engineer, **Warszawa**, Delphi, Pascal, 2D Graphics

Instalacja Spring4D

<https://bitbucket.org/sglienke/spring4d>





Wymagane pakiety w ramach szkolenia

1. Spring4D

- a. aktualny branch master <https://bitbucket.org/sglienke/spring4d>

2. DUnitX

3. Indy

4. TestInsight

- a. instalacja: <https://bitbucket.org/sglienke/testinsight/wiki/Home>

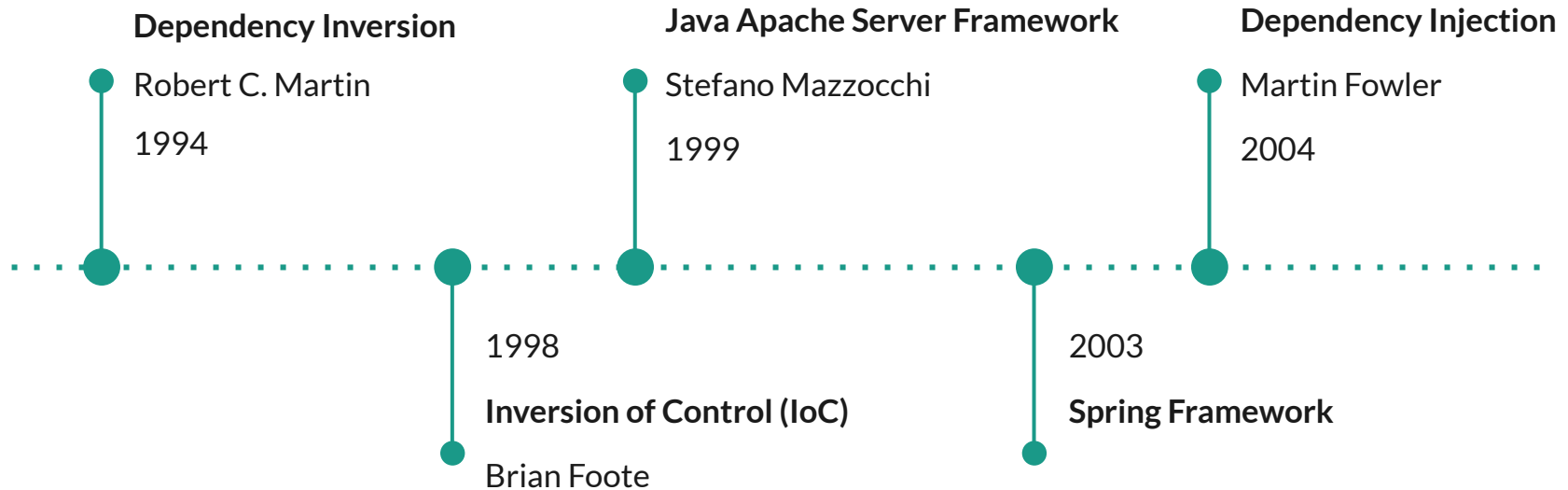
Wstrzykiwanie zależności

Trochę historii

Definicja

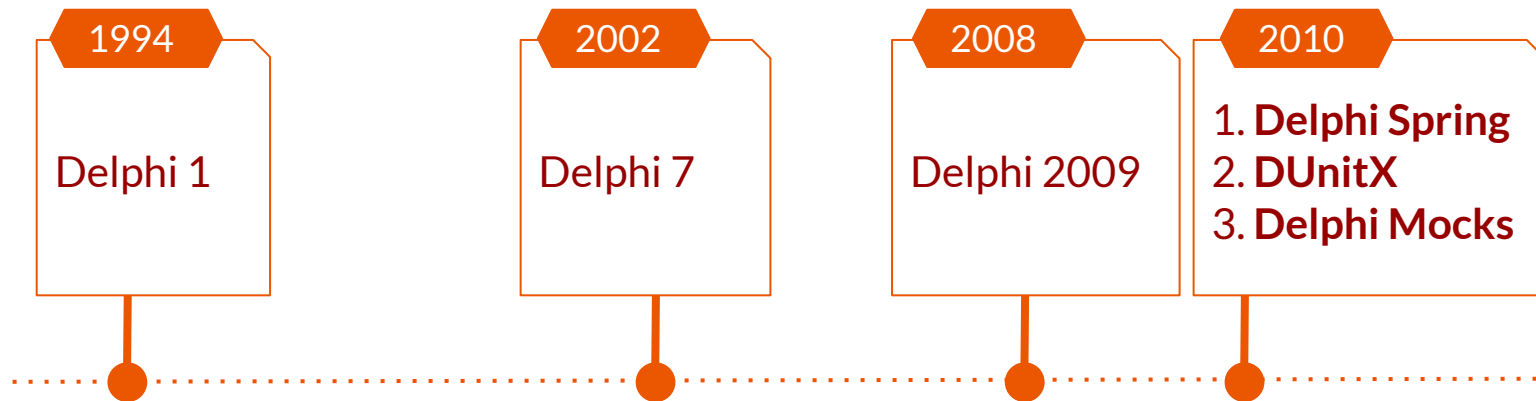


Historia Dependency Injection





Historia Delphi

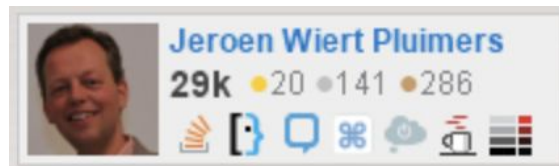


Rok 2011

Posted by **jpluimers** on 2011/09/27

Now that there is Spring and Mocks for Delphi, it is time to post a few links:

- **Nick Hodges indicating he is going to write more about Delphi Spring and Mocks**
- **Nick Hodges' first article (in a 5 series part) on Delphi Spring framework, Dependency Injection and Unit testing**
- **Vincent Parret on Introducing Delphi Mocks**
- **Nick Hodges' Delphi Live session on the Delphi Spring Framework** (I hope he posts samples and slides soon)



DELPHI SORCERY

Pulling rabbits and other stuff out of the hat with delphi ...



Stefan Glienke

19.7k ● 2 ● 45 ● 97

Delphi Sorcery ⇒ Spring4D



Dependency Injection - What is it about

Design Pattern. **Way to write loosely coupled code**

Don't create or make anything new up yourself, allow the caller to do that, and always ask for what you want, don't create what you want yourself.

That pushes the creation of things way back into what's called the **composition root of your application**.

And that causes your code to be very loosely coupled, because its **dependencies are on interfaces**, instead of implementations.



Czy jest mi potrzebny kontener DI?

Dekompozycja
skomplikowanych zadań?

Reużywalność?

Rozwój?

Bezpieczeństwo?

Dependency Injection w Spring4D



Register, Build & Resolve

```
container.RegisterType<IConsole, TStandardConsole>();  
container.RegisterType<ISomeService, TSomeService>;  
container.RegisterType<TApplicationRoot>().AsSingleton();  
// ... all types used in the resolve should be registered here  
  
container.Build();  
  
applicationRoot := container.Resolve<TApplicationRoot>();
```



Lokalny lub Globalny Kontener

```
uses
    Spring.Container;

begin
    container := TContainer.Create();
    try
        // Register, Build & Resolve
    finally
        container.Free;
    end;
end;
```

```
uses
    Spring.Container;

begin
    GlobalContainer.RegisterType<>();
end;
```

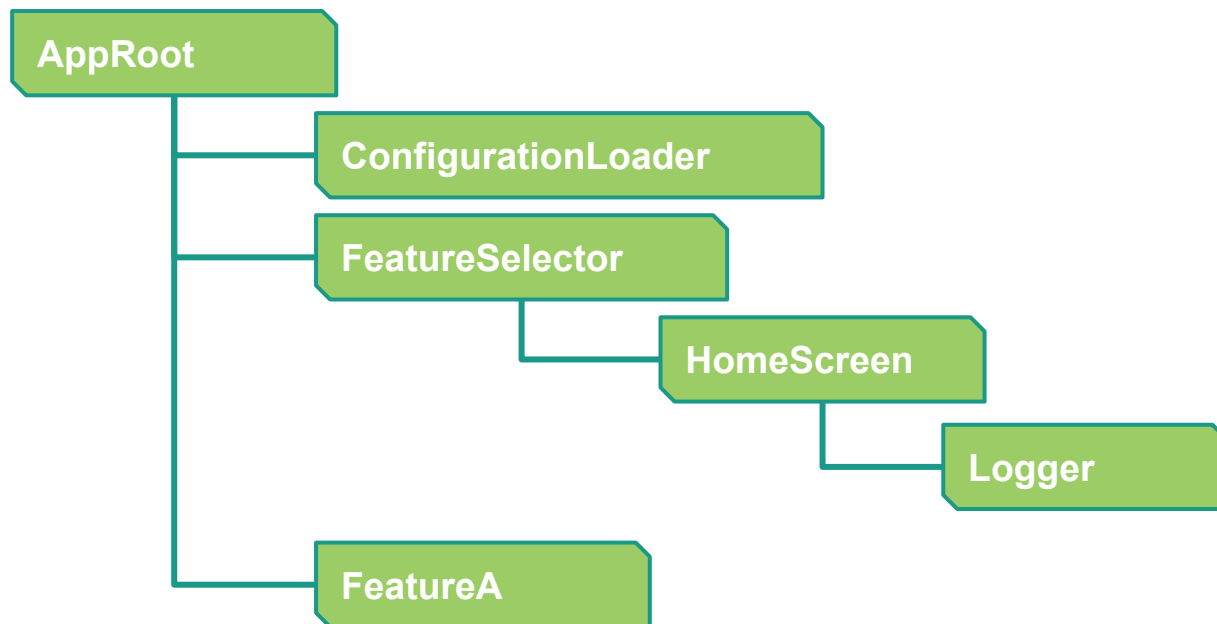
Wstrzykiwanie przez konstruktor

```
TSomeService = class(TInterfacedObject, ISomeService)
private
    fConsole: IConsole;
    fConfiguration: TStringList;
public
    constructor Create(
        const aConsole: IConsole;
        const aConfiguration: TStringList);
    procedure Execute();
end;
```



Wstrzyknięcie serwisu IConsole
Wstrzyknięcie obiektu TStringList

Przykładowe drzewo zależności





Wstrzykiwanie przez pole

```
uses Spring.Container.Common;
```

```
type
```

```
  TAnotherService = class(TInterfacedObject, IAnotherService)
```

```
  private
```

```
    fConsole: IConsole;
```

```
    [Inject]
```

```
    fLogger: ILogger;
```

```
  public
```

```
    constructor Create(const aConsole: IConsole);
```

```
  end;
```



Singleton

Czy w takim kodzie są wycieki pamięci?
`TRoot` jest klasą a nie interfejsem

```
var
    root: TRoot;
begin
    GlobalContainer.RegisterType<TRoot>().AsSingleton();
    GlobalContainer.Build();
    root := GlobalContainer.Resolve<TRoot>();
end;
```



Leniwe serwisy

```
type
  THomeController = class
  private
    fService: Lazy<IExampleService>;
  public
    constructor Create(const service:
Lazy<IExampleService>);
    function GetService: IExampleService;
  end;
```

Resolve jest opóźniony do czasu pierwszego użycia.

Działa podobnie jak fabryka, ale używając fabryki można przekazać dodatkowe parametry.



Activator Extension

uses

```
Spring.Container,  
Spring.Container.ActivatorExtension;
```

Pozwala wykryć brakujące
rejestracje wymaganych serwisów

begin

```
GlobalContainer.AddExtension<TActivatorContainerExtension>();  
GlobalContainer.RegisterType<TRoot>().AsSingleton();  
GlobalContainer.Build();  
root := GlobalContainer.Resolve<TRoot>();
```


end;

Fabryka - kontrakt

type

```
IDbContext = interface  
    [ '{F4653C0C-2C05-4348-A744-3288E520F586}' ]  
    procedure Execute;  
end;
```

Interfejs o rozszerzonych
informacjach RTTI



```
IDbContextFactory = interface(IInvokable)  
    [ '{F632D1FB-9C34-48FD-BD72-6BBC436D1B47}' ]  
    function Create(const aConnectionString: string): IDbContext;  
end;
```



Fabryka - implementacja

```
type
  TDbContext = class(TInterfacedObject, IDbContext)
  private
    fOwner: TComponent;
    fConnectionString: string;
  public
    constructor Create(const aConnectionString: string);
    destructor Destroy; override;
    procedure Execute;
  end;
```



Fabryka - rejestracja i użycie

```
var
  dbContextFactory: IDbContextFactory;
  context: IDbContext;
begin
  GlobalContainer.RegisterType<IDbContext, TDbContext>();
  GlobalContainer.RegisterType<IDbContextFactory>.AsFactory();
  GlobalContainer.Build();

  dbContextFactory := GlobalContainer.Resolve<IDbContextFactory>();
  context := dbContextFactory.Create(aConnectionString);
end;
```

Fabryka za pomocą - reference to function

```
GlobalContainer.RegisterType<TConnectionFactory>.AsFactory();
```

```
type
    {$M+}
    TConnectionFactory = reference to function(const aToken: string)
        : IDbConnection;
    {$M-}

    TMainService = class(TInterfacedObject, IMainService)
    private
        fConnectionFactory: TConnectionFactory;
    public
        constructor Create(const aConnectionFactory:
TConnectionFactory);
        end;
```




Strategia

type

```
TServiceInfo = (siServiceA, siServiceB);
```

```
IService = interface
```

```
  [ '{09D2AC06-85AE-4E27-B614-49B9195AD0F5}' ]
```

```
  function GetType: TServiceInfo;
```

```
  procedure Execute();
```

```
end;
```

```
TServiceA = class(TInterfacedObject, IService)
```

```
TServiceB = class(TInterfacedObject, IService)
```

```
GlobalContainer.RegisterType<IService, TServiceA>('A');
```

```
GlobalContainer.RegisterType<IService, TServiceB>('B');
```

```
services := GlobalContainer.Resolve<TArray<IService>>();
```



Zasady DI

1. Używaj wstrzykiwania do tworzenia serwisów, a nie danych
2. Nazywaj serwisy precyzyjnie (pojedyncza odpowiedzialność)
3. Użyj leniwego serwisu aby nie budować całego drzewa
4. Użyj fabryki jeśli potrzebujesz “później” przekazać parametry
5. Unikaj combo serwisów (zarządzanie danymi + złożona logika)
6. Staraj się unikać mutacji stanu w serwisie
7. Unikaj singletonów
8. ServiceLocator jest uważany za niepoprawny

Komunikacja między serwisami a interfejsem użytkownika



Komunikacja z UI

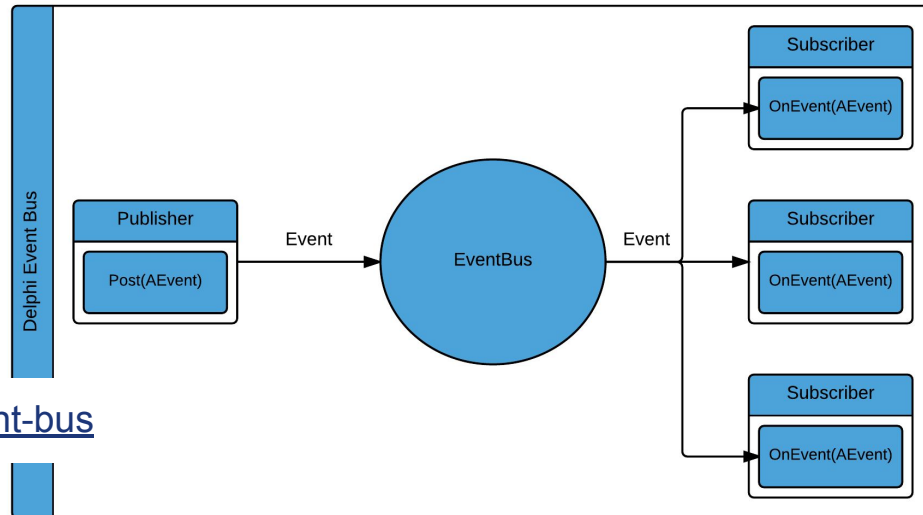
Zasada:

Formatka może znać serwis, ale serwis nie może odwołać się do formatki

- Systematyczne odpytywanie serwisu o stan
- Sprawdzanie flagi w serwisie
- Interfejs na formatce
- Event handler w formatce
- Obserwator
- Event Bus



<https://github.com/spinettaro/delphi-event-bus>



Przykład komunikacji w DEB

uses

EventBus;

```
GlobalEventBus.Post(command);
```

type

TForm1 = class(TForm)

// ... VCL controls and event handles

public

[Subscribe]

procedure OnState1Update(aCommand: IState1ChangedCommand);

end;

IState1ChangedCommand = interface

['{DCFE64D2-9BA8-4949-9BB1-F5CD672E51A2}']

procedure SetState(const aState: TDataState1);

function GetState: TDataState1;

end;



Nazwy serwisów

Updater	Mapper	Matcher
Store	Converter	Loader
Resolver	Cache	Optimizer

~~Manager~~

Unikaj zbyt ogólnych nazw

Czy zmieniać zastane projekty?



Kluczowa decyzja



Jak ulepszyć projekty Delphi używające TDataSet





Jak rozpocząć?

1. Zrozumiały kod - “*dla czytelnika*”
2. Wzorzec DI
3. Dekompozycja kodu
4. Testy jednostkowe - DUnit

Zmiany trzeba wprowadzać powoli i stopniowo, co jakiś czas wyciągając wnioski oraz ulepszać praktyki.

Warto zarezerwować stały czas i wprowadzać ulepszenie przy okazji codziennych zadań

Nie warto pytać przełożonych o zgodę

Zrozumiały kod

1



Jak pisać zrozumiały i prosty kod Delphi?

 Nazywanie klas, metod i zmiennych

 Kod bez “efektów ubocznych”

 Uczciwy kod → Metody szczerą ze sygnaturą

Nazywanie klas, metod i zmiennych



Jak nazywać? Doprować szczerść do bólu

Arlo Belshee - Deep Roots

Get to Obvious Nonsense - SokJabłkowy

<https://www.digdeeproofs.com/articles/get-to-obvious-nonsense/>

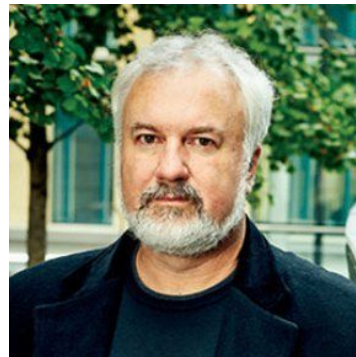
**Kod bez efektów
ubocznych**

Kod bez efektów ubocznych

Functional Code is Honest Code:

<https://michaelfeathers.silvrback.com/functional-code-is-honest-code>

Michael Feathers



<https://youtu.be/iEN2nTuNqDE>

Mattias Petter Johansson (mpj)



Michael Feathers



„Największym problemem w przestarzałym kodzie jest jego **zrozumiałość**”.

„**Niezmiennność** ... wywołujesz funkcję i nie masz pojęcia, że modyfikuje ona jakąś zmienną globalną lub łańcuch pod-metod modyfikuje stan w bazie danych”.

„Innymi słowy, **sygnatura funkcji** jest w pewnym sensie kłamstwem”.

“The biggest issue in legacy code is **understandability**.”

“**Invariably**... You call a function and you have no idea that it modifies some global variable, or saves to a database through a chain of calls.”

“In other words, the **function signature** is a bit of a lie.”

Michael Feathers



„Używamy terminu **efekt uboczny** dla tych rzeczy”

„Programowanie **bez skutków ubocznych** polega na tym, aby uczciwie deklarować sygnatury metod. Jeśli chcesz wiedzieć, czy komunikacja IO jest możliwa to **spójrz na sygnaturę**. Ona powinno Ci o tym powiedzieć.

“We use the term **side effect** for these things”

“Programming **without side effects** is really about making signatures honest in that way. If you want to know whether IO is possible, **look at the signature**. It should tell you.”

Efekty uboczne w praktyce

Zmiana stanu

*Przykład w C# kalkulator liczący pole powierzchni
koła*

```
public class CircleMath {  
    private int _decimalPlaces = 2;  
  
    public double Area(double radius) {  
        var area = Math.PI * Math.Pow(radius, 2);  
        return Math.Round(area, _decimalPlaces);  
    }  
  
    // ... trochę więcej kodu, który zmienia stan obiektu  
}
```

```
var circleMath = new CircleMath();  
Console.WriteLine("area(12) = " + circleMath.Area(12));
```

area(12) = 452.39

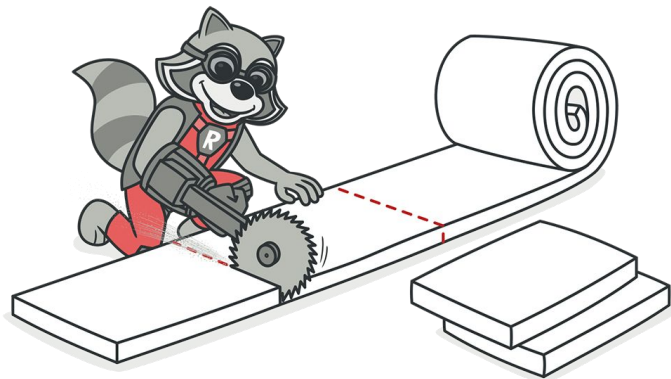
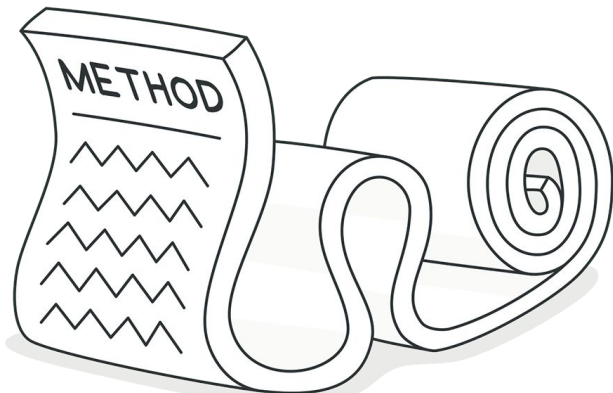
```
var circleMath = new CircleMath();  
circleMath.ChangeDecimalPlaces(8);  
Console.WriteLine("area(12) = " + circleMath.Area(12));
```

area(12) = 452.38934212

Dekompozycja

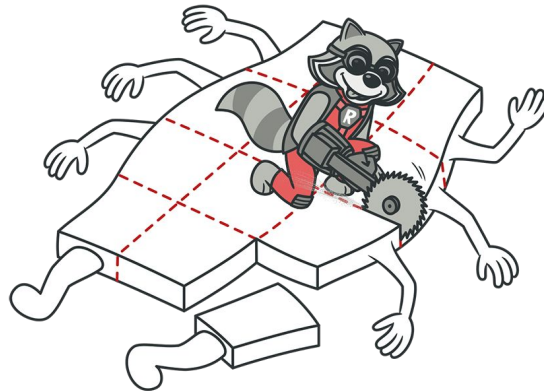
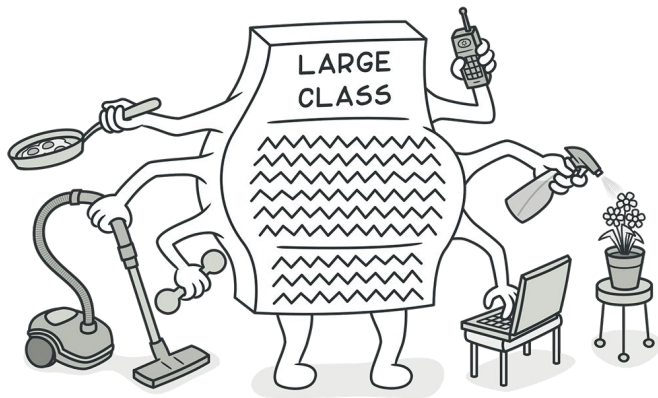
2

Zapasek: Długa metoda



<https://refactoring.guru/smells/long-method>

Zapaszek: Duża klasa



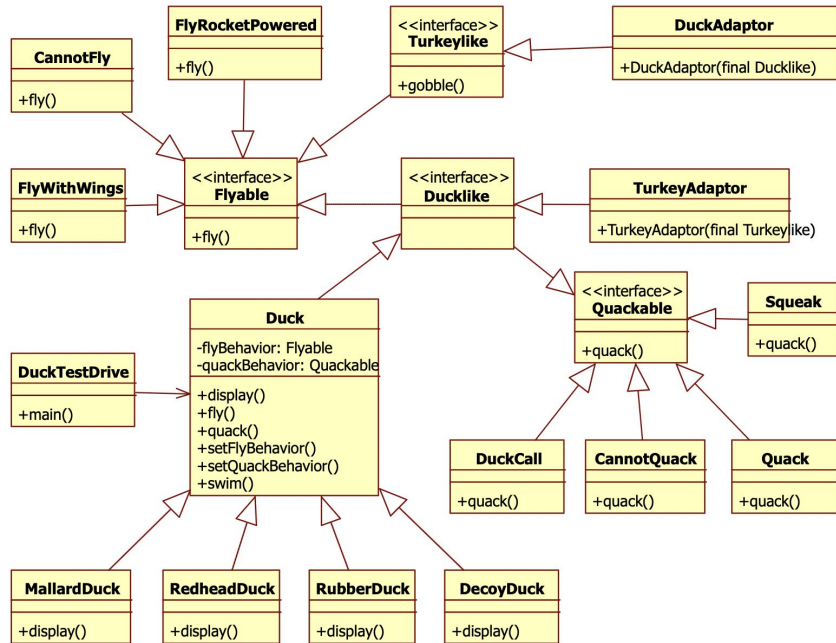
<https://refactoring.guru/smells/large-class>



Gdzie ciąć?
Jak ciąć?

Reguła

Composition
over inheritance





Pojęcia

Cohesion

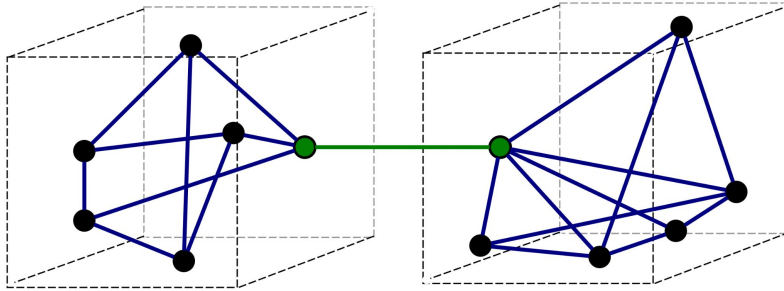
Siła powiązań między elementami
wewnątrz modułu

- Element \Rightarrow Metoda
- Moduł \Rightarrow Klasa

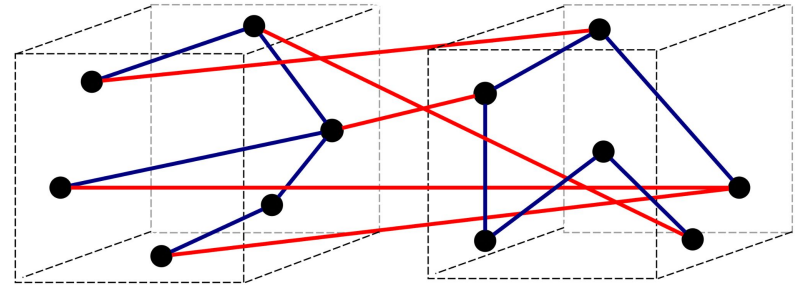
Coupling

Stopień niezależności między modułami

Dopasuj



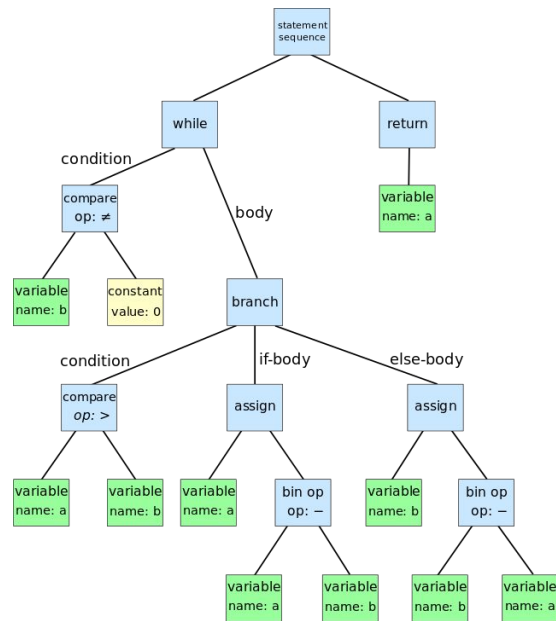
- Highly coupled
- Loosely coupled
- High cohesion
- Low cohesion



Jak mierzyć?

DelphiAST

Zrób to sam



Abstract Syntax Tree

<https://github.com/RomanYankovsky/DelphiAST>



Co mierzyć?

Liczbę używanych unit-ów

Liczbę używanych serwisów
(interfejsów)

Liczba metod

Rozmiar metod

Skomplikowanie unit-u



Mierzenie git-a

Najstarszy kod

Wiek komentarza

Wzorce na ratunek

Fabryka

Budowniczy

Dekorator

Strategia

Serwis domenowy

Repozytorium

Strategia

1. Klient ma dostęp do wszystkich strategii
2. Klient wybiera strategię - zawiera logikę wyboru
3. Klient zleca wykonanie zadania przy pomocy wybranej strategii



Dekorator

1. Klient zna pierwszy komponent
2. Pierwszy komponent zna kolejny komponent
3. Pierwszy komponent wykonuje zadania
4. Pierwszy komponent przekazuje sterowanie do drugiego komponentu



Ewolucja za pomocą wzorców

Architektura
“on the go”

Tworzenie pizzy



Ewolucja do wzorca Builder

- Factory Method
- Abstract Factory
- Builder

↔ Relations with Other Patterns

- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated).

<https://refactoring.guru/design-patterns/builder>

Simple Factory / Factory Method

```
pizza := aSimpleFactoryFunc("Hawaiian"); // TFunc<String,IPizza>
```

```
pizza := _pizzaFactory.build("Hawaiian");
```

Pizza Abstract Factory

```
pizza := TPizza.Create(  
    _pizzaAbstractFactory.buildCrusts("chicago-style"),  
    _pizzaAbstractFactory.buildCheese("mozzarella"),  
    _pizzaAbstractFactory.buildToppings("pineapple", "ham")  
);
```

Pizza Builder

```
pizzaBuilder := TPizzaBuilder.Create();  
pizza = pizzaBuilder  
    .WithCrust("chicago-style")  
    .WithCheese("mozzarella")  
    .WithTopping("pineapple")  
    .WithTopping("ham")  
    .Build();
```

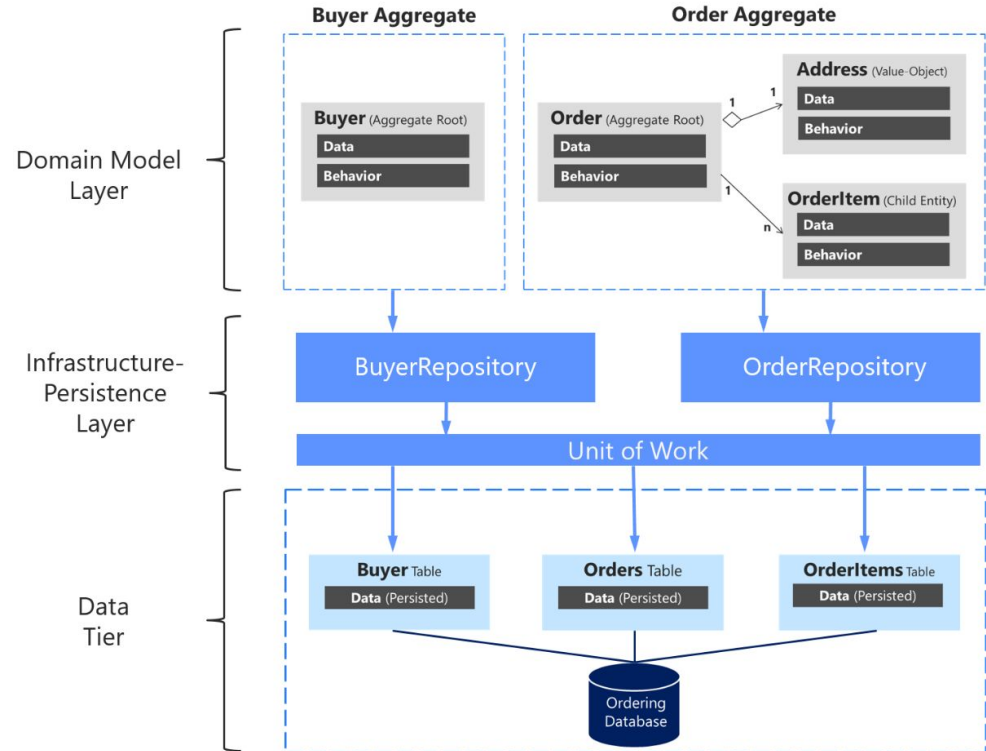

Odległy cel



Repozytoria

Design the infrastructure persistence layer

<https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design>



An orange speech bubble with a pointed tail pointing towards the top-left corner of the image. The bubble has rounded corners and contains the word "Koniec" in white text.

Koniec