



# Wstrzykiwanie Zależności w Praktyce

Bogdan Polak - OEC Connection



[github.com/bogdanpolak](https://github.com/bogdanpolak)



[linkedin.com/in/bogdanpolak](https://linkedin.com/in/bogdanpolak)

---

# O mnie

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

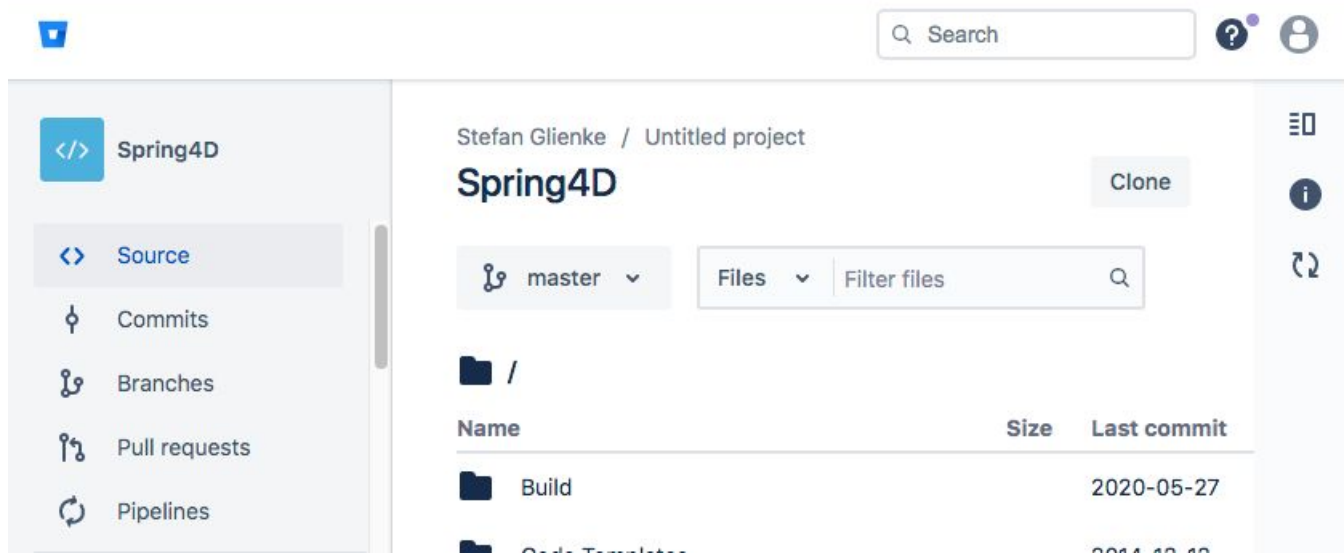
**Craneware**, Senior Software Engineer, **Edinburgh**, Delphi & C#

**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>

---

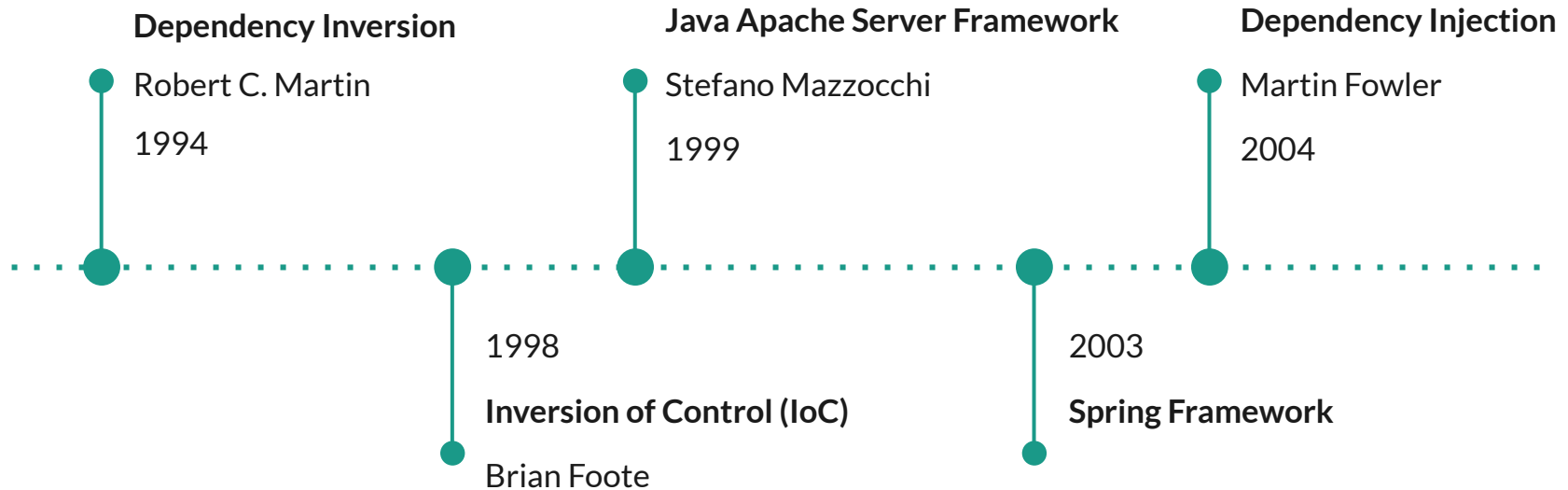
# Wstrzykiwanie zależności

Trochę historii

Definicja

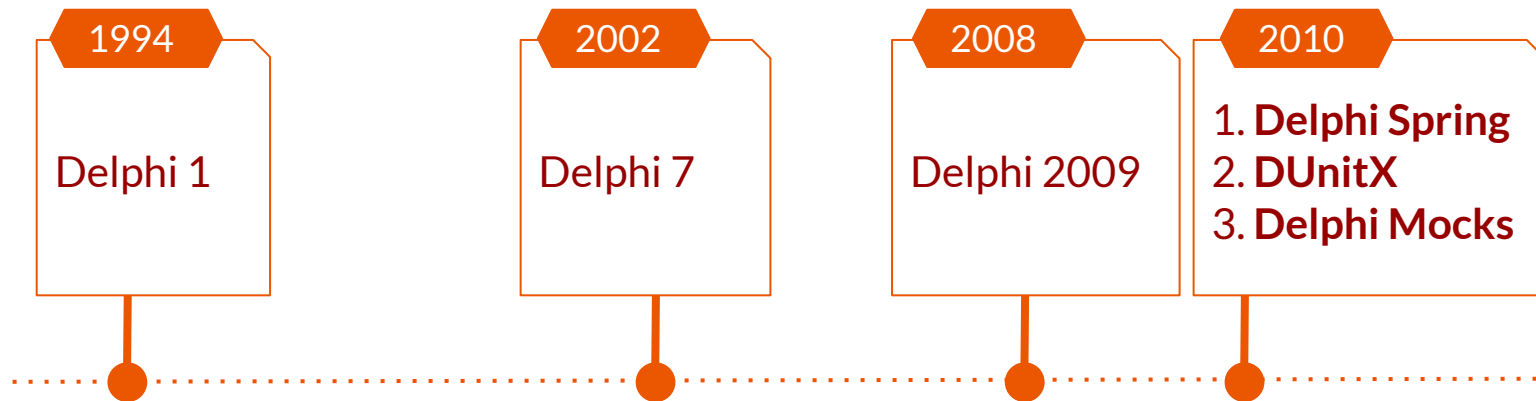


# Historia Dependency Injection





# Historia Delphi

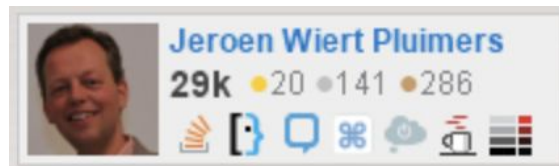


# Rok 2011

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

Spring4D



Stefan Glienke

19.7k ● 2 ● 45 ● 97

**DELPHI SORCERY**

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



## 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.



## Pojęcia

### Cohesion

Siła powiązań między elementami  
wewnątrz modułu

- Element  $\Rightarrow$  Metoda
- Moduł  $\Rightarrow$  Klasa

### Coupling

Stopień niezależności między modułami

# Dopasuj

Highly coupled

1

Loosely coupled

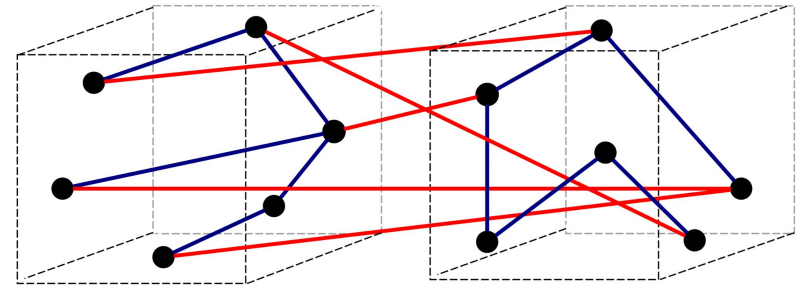
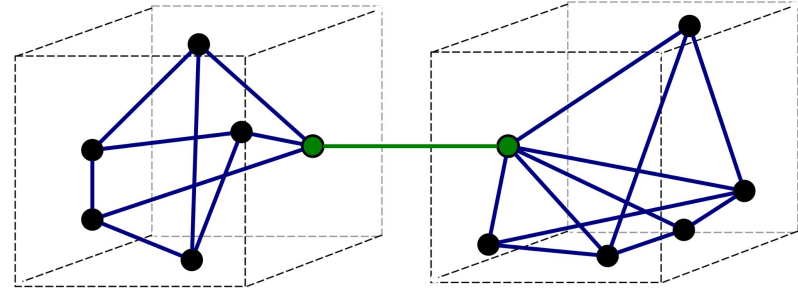
2

High cohesion

3

Low cohesion

4





## Czy jest mi potrzebny kontener DI?

- ☐ Dekompozycja skomplikowanych zadań?
- ☐ Reużywalność?
- ☐ Rozwój?
- ☐ Bezpieczeństwo?

---

# Dependency Injection w Spring4D



## Register, Build & Resolve

Register

*Composer*

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

Resolve

```
applicationRoot := container.Resolve<TApplicationRoot>();
```



# Lokalny lub Globalny Kontener

uses

```
Spring.Container;
```

begin

```
container := TContainer.Create();
```

try

```
// Register
```

```
// Build
```

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*



## 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;
```

## Fabryka - kontrakt

type

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

*Interfejs mający  
rozszerzone informacje RTTI*

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



## Implementacja Serwisu tworzonego przez fabrykę

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



## Fabryka - rejestracja

```
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;
```

## Delegate To

```
type
  TWeatherApiOptions =
    record
      Token: string;
      BaseUrl: string;
    end;
```

```
type
  TTemperatureScale = (
    tsCelsius,
    tsFahrenheit
  );
```

```
aContainer
  .RegisterType<IWeatherAPI, TWeatherAPI>();
aContainer
  .RegisterType<TWeatherApiOptions>()
  .DelegateTo(function: TWeatherApiOptions
    begin
      Result.Token := '--super-secure-token--';
      Result.BaseUrl := 'https://api.org';
    end);
aContainer
  .RegisterType<TTemperatureService>
  .Implements<ITemperatureService>
  .DelegateTo(function: TTemperatureService
    begin
      Result := TTemperatureService.Create(
        aTemperatureScale,
        aContainer.Resolve<IWeatherAPI>());
    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

Pozwala wykryć brakujące  
rejestracje wymaganych serwisów

uses

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

begin

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

end;

## Fabryka za pomocą - reference to function

```
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;
```

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



# Strategia

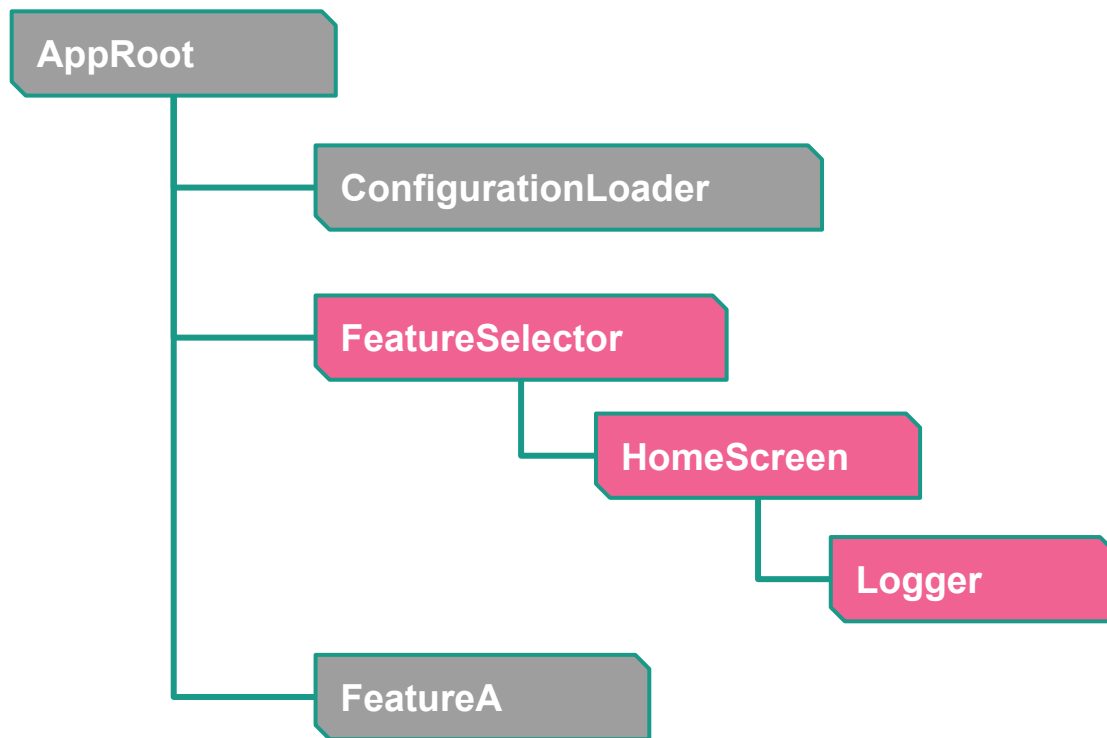
```
type
  TServiceInfo = (siServiceA, siServiceB);

  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>>();
```

## Przykładowe drzewo zależności





## Zasady DI

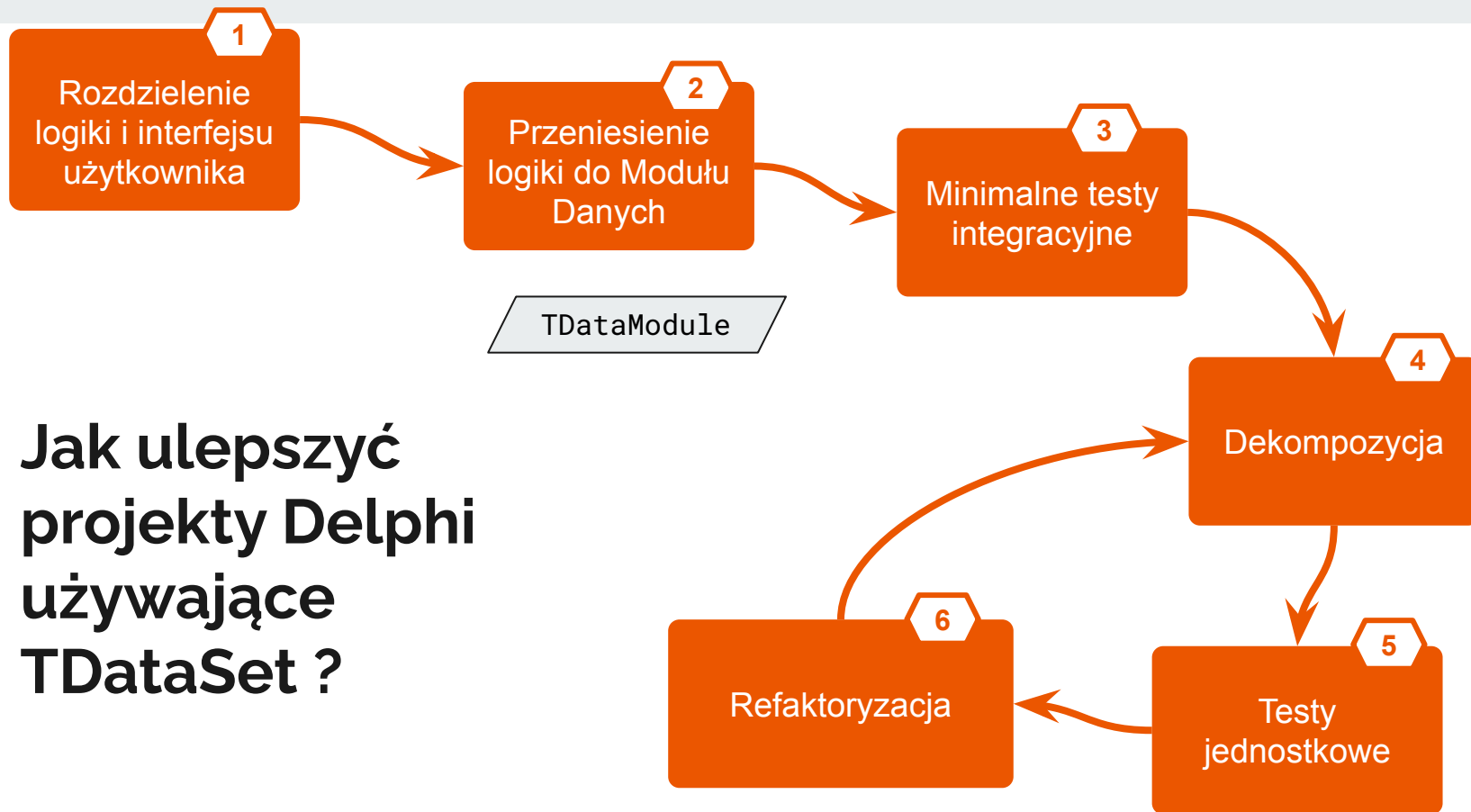
1. Używaj wstrzykiwania do tworzenia serwisów, a nie danych
2. Nazywaj serwisy precyzyjnie (pojedyncza odpowiedzialność)
3. Użyj leniwego serwisu aby nie budować całego drzewa
4. Użyj fabryki jeśli potrzebujesz “później” przekazać parametry
5. Unikaj combo serwisów (zarządzanie danymi + złożona logika)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

---

# VCL i Refactoring

## 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 **stopniowo i powoli**, 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

# Jak pisać zrozumiały i prosty kod Delphi?



Nazywanie klas, metod i zmiennych



Kod bez “efektów ubocznych”



Uczciwy kod → Metody szczerą ze sygnaturą



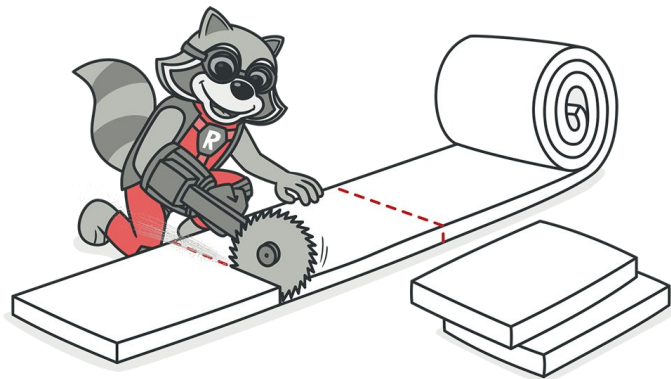
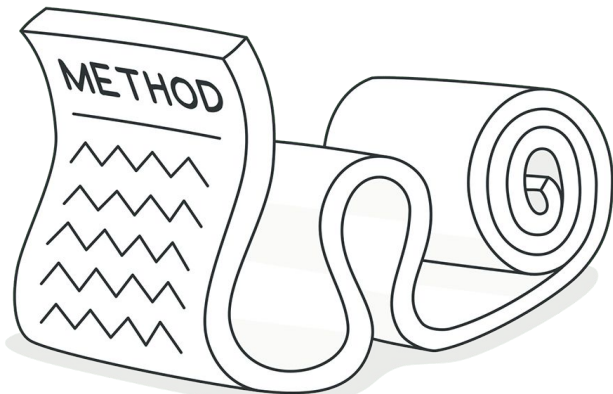
## Nazwy serwisów

|          |           |           |
|----------|-----------|-----------|
| Updater  | Mapper    | Matcher   |
| Store    | Converter | Loader    |
| Resolver | Cache     | Optimizer |

~~Manager~~

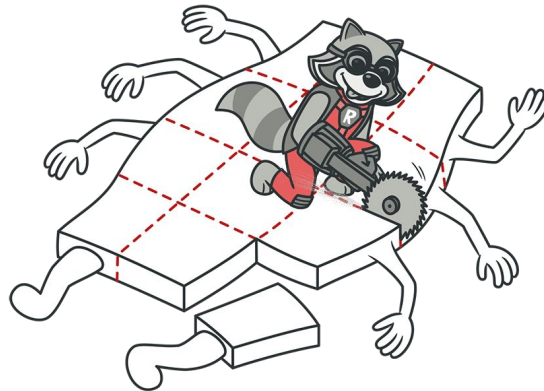
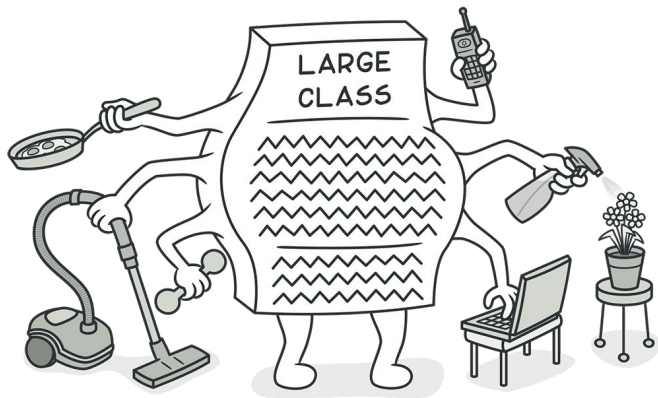
Unikaj zbyt ogólnych nazw

## Zapasek: Długa metoda



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

## Zapaszek: Duża klasa



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



Reguła kształtowania kodu

**Composition**

over inheritance **Composition**

over

compacted  
structure

**Composition**

over

short code

# Wzorce na ratunek

- Strategia
- Dekorator
- Fabryka
- Budowniczy
- 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

**TBD**



# 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



**TBD**



# 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>



# Pizza Restaurant App

```
procedure Order(const pizzaName: string;
                const tableNum: Integer);
begin
    pizza := TPizza.Create();
    pizza.Prepare(pizzaName);      // eg. 'Hawaiian'
    Bake(pizza);
    Serve(pizza, tableNum);
end;
```

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



## Simple Factory / Factory Method

```
procedure Order(const pizzaName: string;  
                const tableNum: Integer;  
                const pizzaFactory: TFunc<string, IPizza>);  
begin  
    pizza := pizzaFactory(pizzaName);  
    Bake(pizza);  
    Serve(pizza, tableNum);  
end;
```

```
pizza := _pizzaFactory.build(pizzaName);
```



## 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();
```

**Serwis  
domenowy**

**TBD**

**Repozytorium**

**TBD**



---

# Testy jednostkowe z DelphiX

# Minimal DUnitX Project

```
program MinimalDUnitX;
{$APPTYPE CONSOLE}
uses
  System.SysUtils,
  DUnitX.Loggers.Console,
  DUnitX.TestFramework;

// TestFeatureOne class declaration and implementation

begin
  TUnitX.RegisterTestFixture(TestFeatureOne);
  TUnitX.CheckCommandLine;
  with TUnitX.CreateRunner() do begin
    UseRTTI := True;
    FailsOnNoAsserts := False;
    AddLogger(TUnitXConsoleLogger.Create());
    Execute;
  end;
end.
```

# Minimal DUnitX Project

```
type
  [TestFixture]
  TestFeatureOne = class
    [Test]
    procedure Test1;
  end;

procedure TestFeatureOne.Test1;
begin
  Assert.Fail();
end;
```

# DUnitX Assertions

```
Assert.AreEqual(  
    'test-{b}',  
    StringReplace('{a}-{b}', '{a}', 'test', [])  
);
```

expected = wartość oczekiwana

actual = wartość wyliczona

```
Assert.AreEqual(44562, EncodeDate(2022, 1, 1), 0.0001);
```

```
Assert.Contains<byte>([1, 2, 3, 5, 8, 13, 21], 8);
```

```
Assert.WillRaise(  
    procedure  
    begin  
        raise Exception.Create('Error Message');  
    end, Exception);
```

# Assertions Dobre Praktyki

Nie używać  
IsTrue / IsFalse

```
Assert.IsTrue(product.Price = 23.49);
```

```
Assert.AreEqual(23.49, product.Price);
```

```
Assert.IsTrue(product.IsDeleted);
```

```
Assert.AreEqual(true, product.IsDeleted);
```

3x A

Arrange = Przygotowanie  
Act = Uruchomienie logiki  
Assert = Weryfikacja

Twórz proste i czytelne testy

3x A

```
// ----- Arrange -----  
sut := TBirthdayCalculator.Create;  
dataset := TDatasetBuilder.BuildFriends([  
    [1, 'Adam', EncodeDate(2001, 4, 5)],  
    [2, 'Tomasz', EncodeDate(1989, 5, 29)]]);  
sut.UseFriendsFrom(dataset);  
  
// ----- Act -----  
friendId := sut.FindNearest(EncodeDate(2022,5,1));  
  
// ----- Assert -----  
Assert.AreEqual(1, friendId);
```

“Twórz proste i czytelne testy”, czyli:

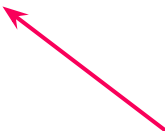
- Unikaj złożonej sekcji Arrange oraz Assert
- Refaktoryzuj testy i pracuj nad czytelnością
- Wykorzystuj Fabryki, proste Build-ery oraz wzorzec Builder

## Łącz asercje

```
products := FindProductsReleasedAt(EncodeDate(2021,06,11));
```

```
product.ToVariants().ShouldBeEquivalentTo([  
    [1, 'iMac 24', Day(11).June(2021), 1249.99]  
]);
```

Array of variants and array of  
arrays of variants are useful in tests



```
products := FindProductsReleasedAt(EncodeDate(2021-06-11));
```

```
Assert.AreEqual(  
    '[{Id:1, Name:iMac 24, Released:2021-06-11, Price:1249.99}]',  
    ProductsToString(products));
```

## Class Helper for Assert

```
sut := TDelphiDatasetGenerator.Create();
sut.dataSet := GivenDataSet(templateMiniHistoricalEvents);

code := sut._GenerateFunction;

Assert.AreMemosEqual(
    'function GivenDataSet (aOwner: TComponent): TDataSet;'#13 +
    'var ds: TClientDataSet;'#13 +
    'begin'#13 +
    '    ds := TClientDataSet.Create(AOwner);'#13 +
    '    with ds do'#13 +
    '        begin'#13 +
    '            FieldDefs.Add(''EventID'', ftInteger);'#13 +
    '            FieldDefs.Add(''Event'', ftWideString, 50);'#13 +
    '            FieldDefs.Add(''Date'', ftDate);'#13 +
    '            CreateDataSet;'#13 +
    '        end;'#13 +
    '        ds.AppendRecord([1, ''Liberation of Poland'', EncodeDate(1989,6,4)]);'#13 +
    '        ds.AppendRecord([2, ''Battle of Vienna'', EncodeDate(1683,9,12)]);'#13 +
    '        ds.First;'#13 +
    '        Result := ds;'#13 +
    'end;'#13, code);
```





## Projektowanie kodu związanego z TDataset



<https://github.com/UweRaabe/DataSetEnumerator>

---

# Separacja i zastępowanie z Delphi Mocks

# Kod łatwy do odizolowania

Łatwy do połączenia / kompozycji

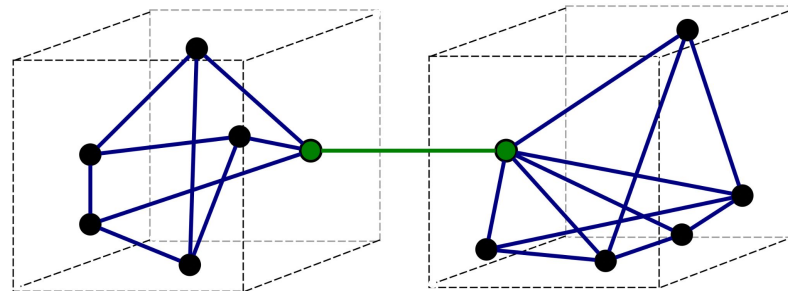
Łatwy do czytania

Reużywalny

Testowalny

Loosely coupled

High cohesion





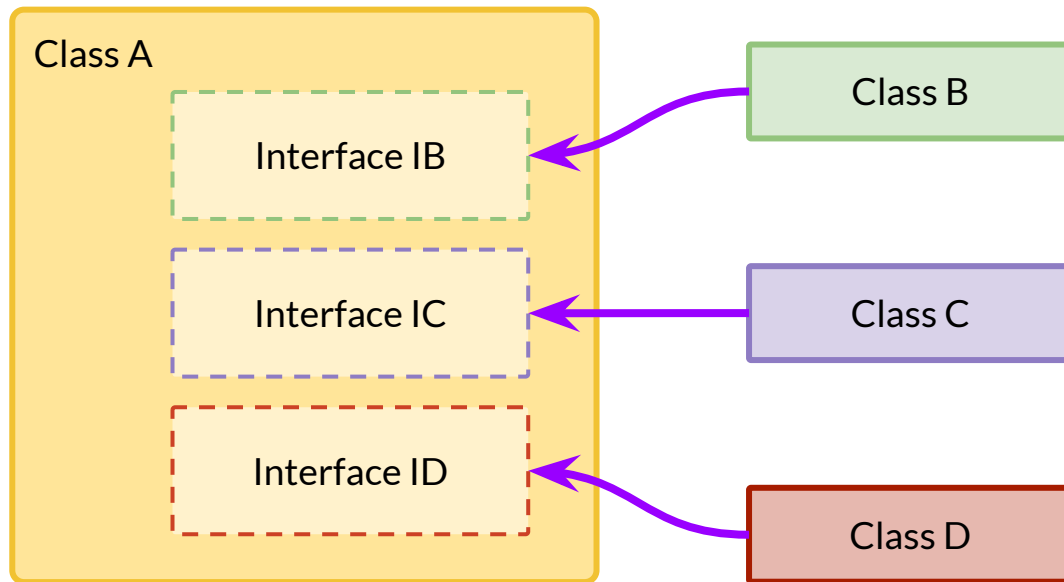
# Jak izolować klasy od innych klas?

Używając interfejsów

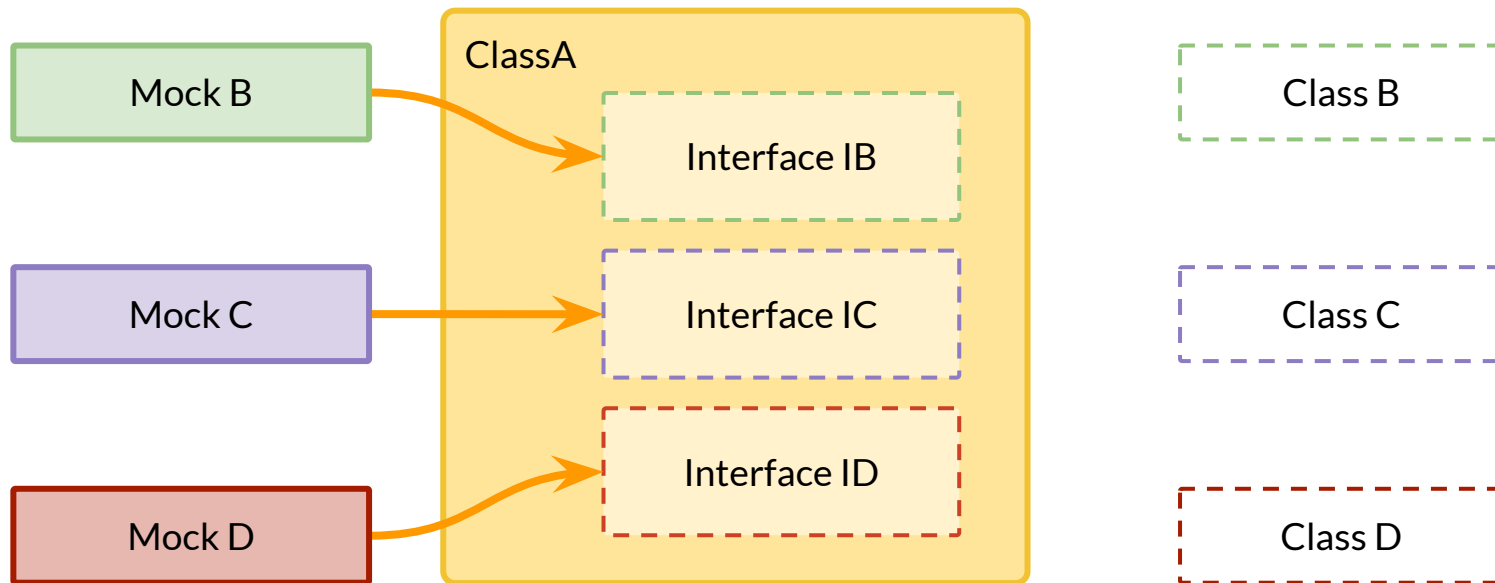
Używając dziedziczenia

Używając funkcji (zdarzenia VCL, funkcje anonimowe)

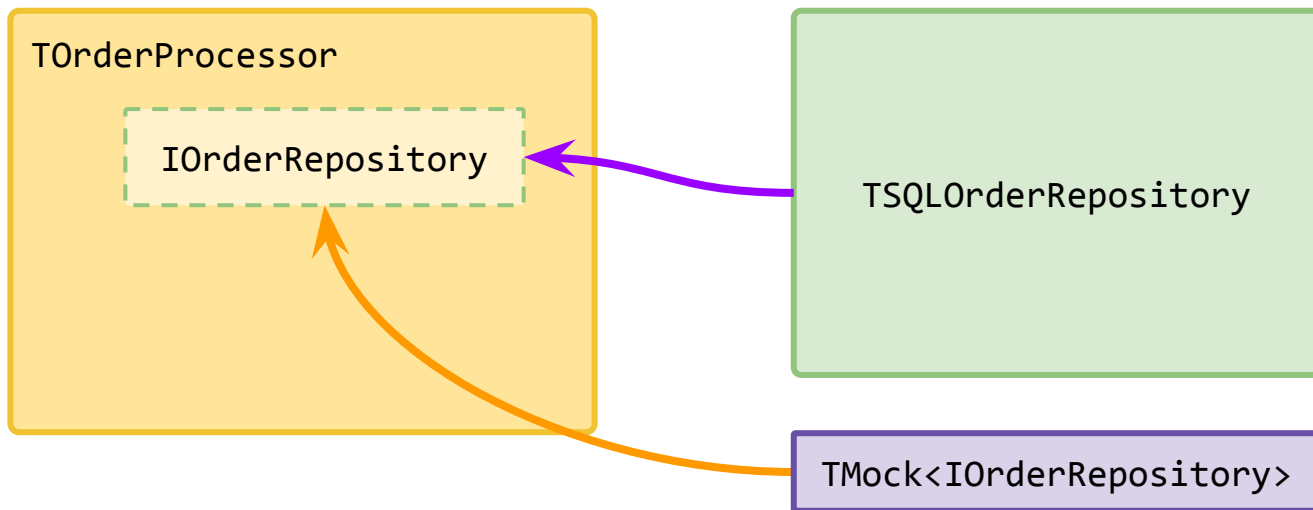
## Używając interfejsów



## Jak zastąpić trudne do testowania klasy?



## Delphi Mocks - W Akcji



## Przygotowanie interfejsu

```
type
  IProcessor = interface(IInvokable)
    ['{69162E72-8C1E-421B-B970-15230BBB3B2B}']
    function GetString(aIdx: Integer): string;
end;
```

Interface has to be declared with  
“Run-Time Type Information”

```
{ $M+ } / { $M- }  
{ $TYPEINFO ON } / { $TYPEINFO OFF }
```



## Create / Setup / Use mock

```
mock := TMock<IProcessor>.Create();  
mock.Setup  
    .WillReturn('item-01')  
    .When.GetString(1);
```

```
mock.Instance.GetString(1);    'item-01'  
mock.Instance.GetString(0);    ⚡ Exception
```



## Setup mock using WillReturnDefault

```
mock.Setup  
    .WillReturnDefault('GetString', 'Hello');
```

```
mock.Setup  
    .WillReturn('Hello')  
    .When.GetString(It.IsAny<Integer>);
```

```
mock.Instance.GetString(1);    'Hello'  
mock.Instance.GetString(0);    'Hello'
```



## Setup mock using WillExecute

```
mock.Setup.WillExecute('GetString',  
    function(const args: TArray<TValue>; const ReturnType: TRttiType): TValue  
    begin  
        // args[0] is the Self interface reference (here it's IProcessor)  
        case args[1].AsInteger of  
            1 .. 9: Result := Format('item-%s', [chr(ord('A') + args[1].AsInteger - 1)]);  
            else      Result := '--error--';  
        end;  
    end);
```

|                              |          |
|------------------------------|----------|
| mock.Instance.GetString(1);  | 'item-A' |
| mock.Instance.GetString(3);  | 'item-C' |
| mock.Instance.GetString(-1); | 'error'  |

## Will Execute with enums and objects

```
type
  TEnum = (evDefault, evOne, evTwo, evThree);

  IProcessor = interface(IInvokable)
    function GetEnum(aIdx: Integer): TEnum;
    function GetObject: TObject;
  end;
```

TValue.From<...>(...)

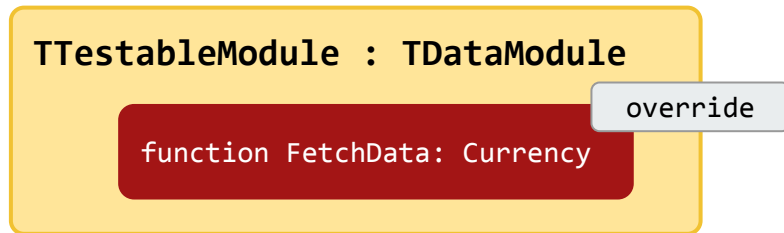
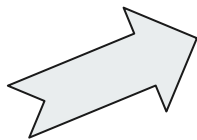
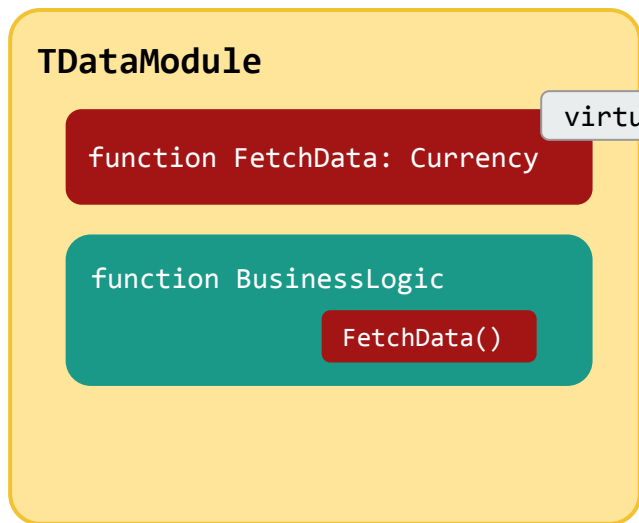
```
var mock := TMock<IProcessor>.Create();
mock.Setup
  .WillReturn(TValue.From<TEnum>(evTwo))
  .When.GetEnum(It0.IsAny<Integer>);
```

```
var mock := TMock<IProcessor>.Create();
var comp01:=TComponent.Create(fOwner);
mock.Setup
  .WillReturn(TValue.From<TComponent>(comp01))
  .When.GetObject;
```

---

# Zagadnienia fakultatywne

## Pull data module in the test harness





## Komunikacja z UI

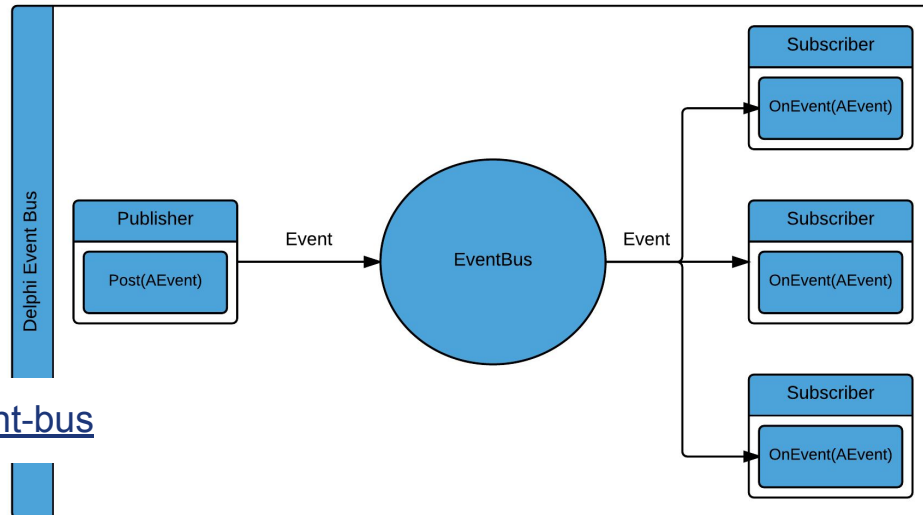
### Zasada:

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>






# Komunikaty w DEB

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

```
type
  ISomethingChangedCommand = interface
    ['{DCFE64D2-9BA8-4949-9BB1-F5CD672E51A2}']
    procedure SetState(const aState: TSomethingState);
    function GetState: TSomethingState;
  end;
```

```
uses
  EventBus;
type
  TForm1 = class(TForm)
    // VCL controls and event handles
  public
    [Subscribe]
    procedure OnSomethingUpdate(
      aCommand: ISomethingChangedCommand);
  end;
```



---

# Koniec