



Nick Hodges

PROGRAMOWANIE W JEZYKU

# Delphi

**Poznaj najlepsze techniki pisania kodu w Delphi!**

Klasy generyczne, metody anonimowe i atrybuty – nowe funkcjonalności języka Delphi

Testy jednostkowe, poprawianie jakości kodu i wykorzystanie platformy izolacyjnej

Wstrzykiwanie zależności a tworzenie czytelnego, luźno sprzężonego i łatwego do testowania kodu

Tytuł oryginału: Coding in Delphi

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-0797-1

©2012 – 2014 Nick Hodges

Polish edition copyright © 2016 by Helion S.A.  
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:  
<ftp://ftp.helion.pl/przyklady/prodel.zip>

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
<http://helion.pl/user/opinie/prodel>  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

# Spis treści

<b>Przedmowa</b> .....	<b>9</b>
<b>Wprowadzenie</b> .....	<b>11</b>
<b>Podziękowania</b> .....	<b>13</b>
<b>Platformy wykorzystane w książce</b> .....	<b>15</b>
<b>Rozdział 1. Wyjątki i ich obsługa</b> .....	<b>17</b>
1.1. Wprowadzenie .....	17
1.2. Strukturalna obsługa wyjątków .....	17
1.3. Jak nie korzystać z wyjątków .....	18
Nie „połykaj” wyjątków .....	18
Nie przechwytuj wyjątków bezkrytycznie .....	18
Nie nadużywaj wyjątków .....	19
Nie używaj wyjątków jako podstawowego sposobu sygnalizacyjnego .....	19
1.4. Jak prawidłowo korzystać z wyjątków .....	20
Używaj wyjątków tak, aby kod ich obsługi nie zakłócał realizacji programu .....	20
Twórcy aplikacji powinni tylko przechwytywać wyjątki .....	20
Przechwytuj tylko wybrane wyjątki .....	20
Wyjątki mogą zgłaszać twórcy komponentów i bibliotek .....	22
Zgłaszaj własne niestandardowe wyjątki .....	22
Niech inni programiści widzą komunikaty o wyjątkach .....	22
Śmiało podawaj dokładne komunikaty o wyjątkach .....	23
W bibliotece twórz dwie wersje każdej metody .....	23
1.5. Wnioski .....	24
<b>Rozdział 2. Interfejsy</b> .....	<b>25</b>
2.1. Wprowadzenie .....	25
2.2. Rozprzęganie kodu .....	25
2.3. Czym jest interfejs? .....	26
2.4. Interfejsy są wszędzie .....	26
2.5. Prosty przykład .....	28
2.6. Implementacja interfejsu .....	29
2.7. Kilka dodatkowych uwag .....	30
2.8. Dziedziczenie interfejsów .....	30
2.9. Inne uwagi warte zapamiętania .....	31
2.10. Klasa <code>TInterfacedObject</code> .....	31
2.11. Jak poprawnie używać interfejsów .....	33

## 4 Programowanie w języku Delphi

2.12. Dlaczego należy używać interfejsów? .....	34
Kodowanie abstrakcyjne .....	34
Implementacje podłączane .....	35
Komunikacja między modułami .....	35
Testowalny kod .....	36
Wzorce kodu .....	36
<b>Rozdział 3. Typy generyczne .....</b>	<b>37</b>
3.1. Z pomocą przybywa typ generyczny .....	38
3.2. Ograniczenia .....	39
Ograniczenie constructor .....	40
Ograniczenie class .....	40
Ograniczenie record .....	41
Ograniczenie interface .....	42
Przekazywanie rekordów jako typów parametrycznych .....	42
3.3. Interfejsy generyczne .....	43
3.4. Metody generyczne .....	43
3.5. Kolekcje generyczne .....	44
3.6. Myślenie generyczne .....	45
3.7. Wystudiowany, prosty przykład .....	45
3.8. Praktyczny przykład .....	46
Typ TWylicz .....	46
Problemy z typami generycznymi .....	48
3.9. Wnioski .....	49
3.10. Wywiad z typem generycznym .....	49
<b>Rozdział 4. Metody anonimowe .....</b>	<b>51</b>
4.1. Wprowadzenie .....	51
4.2. Definicja .....	51
4.3. Po co to wszystko? .....	52
Prosty przykład .....	53
Metody anonimowe jako zmienne .....	55
4.4. Domknięcia .....	55
4.5. Deklaracje standardowe .....	56
4.6. Praktyczny przykład .....	57
4.7. Inny, jeszcze ciekawszy przykład .....	58
Metody anonimowe są bardzo elastyczne .....	59
4.8. Metody anonimowe w bibliotece RTL .....	61
Klasa TThread i metody anonimowe .....	61
4.9. Predykaty .....	61
I co z tego? .....	61
4.10. Wnioski .....	63
4.11. Wywiad z metodą anonimową .....	63
<b>Rozdział 5. Kolekcje .....</b>	<b>65</b>
5.1. Wprowadzenie .....	65
5.2. Ogólne uwagi dotyczące kolekcji .....	65
5.3. Kolekcje w języku Delphi .....	66
Kolekcja TList<T> .....	66
Kolekcja TStack<T> .....	68
Kolekcja TQueue<T> .....	69
Kolekcja TDictionary<TKey, TValue> .....	70
Kolekcje obiektów .....	72

5.4. Kolekcje w platformie Delphi Spring Framework .....	72
Ogólne informacje .....	72
Dwa nieopisane dotychczas typy kolekcji .....	73
5.5. Dlaczego należy używać kolekcji Spring4D? .....	73
5.6. Wnioski .....	74
<b>Rozdział 6. Enumeratory w Delphi .....</b>	<b>75</b>
6.1. Wprowadzenie .....	75
6.2. Interfejs IEnumerator<T> .....	77
6.3. Specjalistyczne enumeratory .....	78
6.4. Klasa TEnumerable<T> w module Generics.Collections .....	80
6.5. Wnioski .....	80
<b>Rozdział 7. Interfejs IEnumerable .....</b>	<b>81</b>
7.1. Interfejs IEnumerable<T> .....	82
7.2. Predykaty .....	83
7.3. Wywiad z interfejsem IEnumerable<T> (IEoT) .....	87
<b>Rozdział 8. Informacje RTTI .....</b>	<b>89</b>
8.1. Wprowadzenie .....	89
8.2. Typ TValue .....	90
8.3. RTTI i klasy .....	92
Klasa TRttiType .....	94
8.4. RTTI i instancje klas .....	96
Odczytywanie i nadawanie wartości .....	96
Wywoływanie metod .....	97
8.5. Uwagi ogólne .....	98
8.6. RTTI i inne typy danych .....	98
RTTI i typy porządkowe .....	99
RTTI i rekordy .....	99
RTTI i tabele .....	99
8.7. Inne metody RTTI .....	100
8.8. Dyrektywy kompilatora związane z RTTI .....	100
Silna konsolidacja typów .....	102
8.9. Wnioski .....	102
<b>Rozdział 9. Atrybuty .....</b>	<b>103</b>
9.1. Wprowadzenie .....	103
9.2. Czym są atrybuty? .....	103
9.3. Prosty przykład .....	107
9.4. Wnioski .....	109
9.5. Wywiad z atrybutem .....	109
<b>Rozdział 10. Klasa TVirtualInterface .....</b>	<b>111</b>
10.1. Trochę lepsza klasa TVirtualInterface .....	115
10.2. Naprawdę użyteczny przykład .....	116
10.3. Interfejs IProstaAtrapa .....	116
10.4. Klasa TProstaImitacja .....	118
10.5. Wnioski .....	120
<b>Rozdział 11. Wstęp do wstrzykiwania zależności .....</b>	<b>121</b>
11.1. Wprowadzenie .....	121
11.2. Czym jest zależność? .....	121
11.3. Prawo Demeter .....	122
11.4. Przykład projektu .....	122

## 6 Programowanie w języku Delphi

11.5. Prawo Demeter w języku Delphi .....	123
11.6. Przykładowy kod krok po kroku .....	124
11.7. Kontener wstrzykiwania zależności .....	131
11.8. Wnioski .....	134
<b>Rozdział 12. Więcej o wstrzykiwaniu zależności .....</b>	<b>135</b>
12.1. Zależności opcjonalne .....	136
12.2. Wstrzykiwanie metod przypisujących .....	136
12.3. Wstrzykiwanie metod .....	137
12.4. Kontener Spring Container i klasa ServiceLocator .....	138
Rejestrowanie interfejsów .....	139
Zarządzanie czasem życia obiektów .....	140
Niestandardowe tworzenie obiektów .....	141
Określenie domyślnego obiektu .....	142
Rejestrowanie tych samych klas dla dwóch interfejsów .....	142
Wstrzykiwanie pól i właściwości podczas rejestracji .....	142
Wykorzystanie rejestracji do wstrzykiwania konstruktorów i metod .....	144
12.5. Rejestrowanie elementów za pomocą atrybutów .....	145
12.6. Klasa ServiceLocator jako antywzorzec .....	147
12.7. Podsumowanie wstrzykiwania zależności .....	151
<b>Rozdział 13. Testy jednostkowe .....</b>	<b>153</b>
13.1. Co to są testy jednostkowe? .....	153
Co to jest „jednostka”? .....	154
Czy rzeczywiście wykonuję testy jednostkowe? .....	154
Czym jest platforma izolacyjna? .....	154
13.2. Po co wykonywać testy jednostkowe? .....	157
Dzięki testom jednostkowym wyszukasz błędy .....	157
Dzięki testom jednostkowym unikniesz błędów .....	157
Dzięki testom jednostkowym oszczędzisz czas .....	157
Testy jednostkowe zapewniają spokój .....	157
Testy jednostkowe dokumentują właściwe użycie klas .....	158
13.3. Testy jednostkowe w Delphi .....	158
Platforma DUnit .....	158
Platforma DUnitX .....	158
13.4. Ogólne zasady testów jednostkowych .....	159
Testuj jedną klasę w izolacji .....	159
Przestrzegaj zasady AAA .....	159
Najpierw twórz proste testy, „krótkie i na temat” .....	159
Twórz testy sprawdzające zakresy .....	159
Testuj granice .....	160
Jeżeli to możliwe, sprawdź całe spektrum możliwych wartości .....	160
Jeżeli to możliwe, sprawdź każdy przebieg kodu .....	160
Twórz testy wyszukujące błędy i poprawiaj je .....	160
Každy test musi być niezależny od innych testów .....	160
Stosuj w teście tylko jedną asercję .....	161
Nadawaj testom czytelne nazwy i nie przejmuj się, że są długie .....	161
Sprawdź, czy każdy wyjątek jest rzeczywiście zgłaszany .....	161
Unikaj stosowania metod CheckTrue i Assert.IsTrue .....	161
Regularnie wykonuj testy .....	161
Wykonuj testy przy każdorazowej kompilacji kodu .....	161
13.5. Programowanie uwzględniające testy .....	162
13.6. Prosty przykład .....	162

<b>Rozdział 14. Testy z użyciem platformy izolacyjnej .....</b>	<b>169</b>
14.1. Krótkie przypomnienie .....	169
14.2. Platformy izolacyjne w Delphi .....	170
14.3. Jak zacząć .....	170
Do akcji wkracza platforma izolacyjna .....	170
Prosta atrapa .....	170
Testowanie dziennika .....	172
Atrapy, które wykonują operacje .....	173
Zależności wykonujące oczekiwane operacje .....	176
Zależności zgłaszające wyjątki .....	178
W teście stosuj tylko jedną imitację .....	179
Oczekiwane parametry muszą być zgodne z rzeczywistymi .....	180
14.4. Wnioski .....	180
<b>Dodatek A. Materiały .....</b>	<b>181</b>
Wstrzykiwanie zależności .....	181
Testy jednostkowe .....	181
Systemy kontroli kodu źródłowego .....	182
Subversion .....	182
Git .....	182
Mercurial .....	182
Projekty .....	182
Inne ciekawe materiały .....	183
<b>Dodatek B. Moja przygoda z Delphi .....</b>	<b>185</b>
<b>Skorowidz .....</b>	<b>189</b>





# Klasa TVirtualInterface

W tym miejscu powinieneś już być przekonany, że należy programować interfejsy, a nie implementacje. (Czyż nie obiecałem Ci w pierwszym rozdziale, że będę to zdanie powtarzał aż do znudzenia?) Interfejsy umożliwiają tworzenie luźno sprzężonego kodu. Jeżeli wciąż nie wierzysz, że rozprzężenie kodu jest wyjątkowo ważną kwestią, radzę Ci przerwać lekturę w tym miejscu, wziąć do ręki mały młotek i stukać się nim w czoło tak długo, aż zmienisz zdanie.

Jeżeli więc już to zrobiłeś, to wiesz, że interfejs przed użyciem trzeba zaimplementować. Zanim cokolwiek zrobisz, musisz dopisać do niego trochę kodu. Zazwyczaj wykorzystuje się w tym celu klasę implementacyjną:

```
1 type
2   IPrzetwarzanieDanych = interface
3     procedure PrzetwarzajDane;
4   end;
5
6 TPrzetwarzanieDanych = class(TInterfacedObject, IPrzetwarzanieDanych)
7   procedure PrzetwarzajDane;
8 end;
```

A gdyby tak zaimplementować interfejs, nie korzystając z określonej klasy? Gdyby można było w jakiś sposób zaimplementować interfejs za pomocą jednego modułu kodu? Gdyby można było decydować podczas wykonywania programu o sposobie implementacji interfejsu? Czy zadawałbym te pytania i pisał o tych rzeczach, gdyby nie były możliwe do wykonania?

Oczywiście jest to możliwe. W wersji Delphi XE2 została wprowadzona bardzo ciekawa klasa **TVirtualInterface**, służąca do tworzenia klas pochodnych i dynamicznego implementowania interfejsów. Jeżeli zastanowisz się przez chwilę, stwierdzisz, że jest to bardzo ciekawa funkcjonalność. Została ona wykorzystana między innymi w niezwyklej platformie testowej Delphi Mocks Framework (o której opowiem w następnym rozdziale), umożliwiającej implementację imitacji klas dowolnego interfejsu, który się w powyższej klasie wskaże.

Jak dowiedziałeś się w rozdziale 2., zazwyczaj podczas implementacji interfejsu trzeba tworzyć określoną klasę. Zatem podczas wykonywania kodu będzie to implementacja statyczna. Istnieją sposoby, zazwyczaj jest to wstrzykiwanie zależności, umożliwiające wybór implementacji, ale nawet wtedy zbiór klas jest ograniczony.

Klasa **TVirtualInterface** umożliwia dynamiczne, tj. w trakcie wykonywania kodu, określenie sposobu implementacji interfejsu. Zacznę od pokazania kilku prostych, a potem bardziej praktycznych przykładów zastosowania tej klasy.

Poniżej przedstawiona jest publiczna część klasy **TVirtualInterface**:

```
1 { TVirtualInterface: Creates an implementation of an interface at runtime.
2   All methods in the Interface are marshaled through a generic stub function
3   that raises the OnInvoke event.}
```

```

4 TVirtualInterface = class(TInterfacedObject, IInterface)
5 ...
6 public
7   function QueryInterface(const IID: TGUID; out Obj): HRESULT; virtual; stdcall;
8   { Create an instance of TVirtualInterface that implements the methods of
9     an interface. PIID is the PTypeInfo for the Interface that is to be
10    implemented. The Interface must have TypeInfo ($M+). Either inherit from
11    IInvokable, or enable TypeInfo for the interface. Because this is an
12    TInterfacedObject, it is reference counted and it should not be Freed
13    directly.
14  }
15  constructor Create(PIID: PTypeInfo); overload;
16  constructor Create(PIID: PTypeInfo;
17                    InvokeEvent: TVirtualInterfaceInvokeEvent); overload;
18  destructor Destroy; override;
19  { OnInvoke: Event raised when a method of the implemented interface is called.
20    Assign a OnInvoke handler to perform some action on invoked methods.}
21  property OnInvoke: TVirtualInterfaceInvokeEvent read FOnInvoke
22    write OnInvoke;
23 end;

```

Poniżej znajduje się kilka uwag do tej deklaracji:

- Przede wszystkim zauważ, że klasa `TVirtualInterface` pochodzi od klasy `TInterfacedObject` i implementuje interfejs `IInterface`. Zaimplementowane są w niej trzy metody tego interfejsu umożliwiające prawidłowe zliczanie referencji, tak samo jak w przypadku innych klas implementacyjnych.
- Po drugie, interfejs, który ma być zaimplementowany za pomocą tej klasy, musi zawierać metodę `TypeInfo`. Najprostszym sposobem spełnienia tego warunku jest utworzenie interfejsu pochodnego od `IInvokable`. W przeciwnym wypadku trzeba w kodzie interfejsu użyć dyrektywy kompilatora `{M+}`. Zwróć również uwagę, że komentarz nad deklaracją klasy zawiera informacje, o których pisałem wcześniej — klasa `TVirtualInterface` umożliwia wykonywanie na interfejsie dowolnych operacji podczas realizacji kodu. Super.
- Ponadto pamiętaj, że w klasie tej ponownie zadeklarowana jest metoda `QueryInterface`, zamieniona na metodę wirtualną.

Aby móc coś zrobić za pomocą klasy `TVirtualInterface`, trzeba utworzyć jej klasę pochodną i umieścić w niej dwa elementy: konstruktora i implementację metody `DoEvent`.

Poniżej, w najprostszym możliwym przykładzie, jaki mi przyszedł do głowy, przedstawiona jest klasa pochodna od `TVirtualInterface`:

```

1 type
2   TNajprostszyInterfejsWirtualny = class(TVirtualInterface)
3     constructor Create(PIID: PTypeInfo);
4     procedure WywołajMetodę(Metoda: TRttiMethod; const Argumenty: TArray<TValue>;
5       out Wynik: TValue);
6   end;
7
8   constructor TNajprostszyInterfejsWirtualny.Create(PIID: PTypeInfo);
9   begin
10    inherited Create(PIID, WywołajMetodę);
11  end;
12
13  procedure TNajprostszyInterfejsWirtualny.WywołajMetodę(Metoda: TRttiMethod;
14    const Argumenty: TArray<TValue>; out Wynik: TValue);
15  begin
16    WriteLn('Wywołaj metodę interfejsu');
17  end;

```

Jedyną wykonywaną tu operacją jest wyświetlenie komunikatu w oknie wiersza polecenia, niezależnie od wywołanej metody. Jest to udawana implementacja każdego interfejsu i niezależnie od wywołanej metody zostanie jedynie wyświetlony komunikat.

Konstruktor ma jeden parametr, `IID`, będący wskaźnikiem typu `PTypeInfo` do interfejsu, który ma być zaimplementowany. (Dlatego w przypadku tego interfejsu musi być użyta dyrektywa `{M+}`. Najczęściej i najprościej robi się to za pomocą interfejsu `IInvokable`). Wewnątrz konstruktora wywoływany jest inny konstruktor. Przekazywany jest mu parametr `IID`, jak również referencja do metody `WywołajMetodę` typu `TVirtualInterfaceInvokeEvent`. W ten sposób konstruktor mówi: „Tutaj jest informacja dla implementowanego interfejsu i metoda, która będzie wywoływana po wywołaniu dowolnej metody tego interfejsu”.

W tym przypadku metoda `WywołajMetodę` wykonuje tylko jedną operację — wyświetla komunikat w oknie wiersza poleceń.

Założmy więc, że zadeklarowany jest następujący interfejs:

```
1 type
2   IStartStop = interface(IInvokable)
3     ['{3B2171B0-D1C3-4A8C-B09E-ACAC4D625E57}']
4     procedure Start;
5     procedure Stop(aLiczba: integer);
6   end;
```

Następnie uruchomiona została poniższa aplikacja konsolowa:

```
1 StartStop := TNajprostszyInterfejsWirtualny.Create(TypeInfo(IStartStop))
2 as IStartStop;
3 StartStop.Start;
4 StartStop.Stop(42);
```

Zawartość okna wiersza poleceń będzie następująca:

```
1 Wywołałeś metodę interfejsu
2 Wywołałeś metodę interfejsu
```

Komunikat wyświetlony będzie dwukrotnie, ponieważ w powyższym kodzie wywoływane są dwie metody. Efekt będzie zawsze ten sam, niezależnie od podanego interfejsu i wywoływanej metody.

Oczywiście taki kod do niczego się nie nadaje. Aby był on w jakikolwiek sposób przydatny, potrzebne są informacje o wywoływanych metodach i przekazywanych parametrach, na które będzie można odpowiednio reagować.

Jest to możliwe do zrobienia. Przyjrzyj się sygnaturze metody `WywołajMetodę`. Zauważ, że gdy jest ona wywoływana przez metodę `TVirtualInterface.OnInvoke`, przekazywane są jej informacje RTTI o wywoływanej metodzie, tabela klas `TValue` zawierająca interfejs wraz z argumentami przekazywanymi tej metodzie, jak również parametr wyjściowy typu `TValue` umożliwiający zwrócenie wartości, jeżeli wywoływana metoda jest funkcją.

Wykorzystajmy zatem metodę `WywołajMetodę` do wyświetlenia wszystkich odebranych informacji.

```
1 procedure TWirtualnyInterfejsRaportujący.WywołajMetodę(Metoda: TRttiMethod;
2   const Argumenty: TArray<TValue>; out Wynik: TValue);
3 var
4   Argument: TValue;
5   TypArgumentu, NazwaArgumentu: string;
6   TymczTyp: TTypeKind;
7 begin
8   Write('Wywołałeś metodę ', Metoda.Name);
9   if Length(Argumenty) > 1 then
10    begin
11     WriteLn(' która ma ', Length(Argumenty) - 1, ' parameterów:');
12     for Argument in Argumenty do
13       begin
14         TymczTyp := Argument.Kind;
```

## 114 Programowanie w języku Delphi

```
15     if TymczTyp <> tkInterface then
16     begin
17         NazwaArgumentu := Argument.ToString;
18         TypArgumentu := Argument.TypeInfo.Name;
19         WriteLn(NazwaArgumentu, ' typu ', TypArgumentu);
20     end;
21 end;
22 end else
23 begin
24     WriteLn(', która nie ma parametrów.');
```

W tym kodzie są po prostu przeglądane i wyświetlane wartości parametrów `Metoda` i `Argumenty` przekazane metodzie podczas jej wywołania. Pierwszy element tabeli zawiera zawsze informację o typie samego interfejsu, a pozostałe elementy są parametrami umieszczonymi w kolejności ich przekazania. W tym przykładzie kod po prostu wyświetla wartości i ich typy, ale oczywiście można je też przetwarzać wedle uznania.

Powtórzę, jest to interesująca informacja, ale jest to dopiero jeden krok w kierunku poznania sposobu działania klasy `TVirtualInterface`. Utwórzmy kod, który wykonuje jakąś pożyteczną operację.

Poniżej przedstawiony jest podstawowy interfejs:

```
1 type
2   IPrzydatnyInterfejs = interface(IInvokable)
3     ['{16F01BF0-961F-4461-AE8E-B1ACB8D3F0F4}']
4     procedure Witaj;
5     function TekstWstecz(aTekst: string): string;
6     function Iloczyn(x, y: integer): integer;
7 end;
```

Poniżej przedstawiona jest metoda `WywołajMetodę` klasy `TPrzydatnyInterfejs`, wykonująca operacje, do których interfejs jest przeznaczony:

```
1 procedure TPrzydatnyInterfejs.WywołajMetodę(Metoda: TRttiMethod;
2     const Argumenty: TArray<TValue>; out Wynik: TValue);
3 begin
4     if UpperCase(Metoda.Name) = 'WITAJ' then
5     begin
6         WriteLn('Witaj świecie!');
```

Powyższy kod nie wymaga wyjaśnień. Sprawdza po prostu nazwę wywoływanej metody i wykonuje odpowiednią operację, wykorzystując informacje przekazane w parametrze `Argumenty`. Jeżeli wywoływaną metodą jest funkcja, wtedy zmienna `Wynik` jest wykorzystywana jako zwracana wartość.

Jak pamiętasz, pierwszym elementem (czyli tym na pozycji zerowej) tabeli Argumenty jest typ samego interfejsu. W powyższym kodzie przyjęte jest założenie dotyczące liczby i typów parametrów. Ponieważ kod ten może być uruchomiony jedynie przez metody zadeklarowane w interfejsie IPrzydatnyInterfejs, takie założenie można bezpiecznie przyjąć.

Teraz opisane kody powinny być jasne — w przykładach w zasadzie symulowana jest klasa implementacyjna. Tak naprawdę nie jest to implementacja dynamiczna. Kody są jedynie prostymi przykładami statycznego wykorzystania klasy TVirtualInterface. Teraz musisz się dowiedzieć, jak dynamicznie implementować interfejs za pomocą klasy pochodnej od TVirtualInterface.

## 10.1. Trochę lepsza klasa TVirtualInterface

Do tej pory analizowaliśmy tylko kod demonstracyjny. Nie mogę sobie wyobrazić żadnego sposobu jego zastosowania w praktyce. Pokazuje on jednak funkcjonowanie klasy TVirtualInterface i wykorzystanie jej do własnych celów.

Ponieważ TVirtualInterface jest fajną klasą, jest również nieco kłopotliwa w użyciu. A gdyby tak utworzyć klasę pochodną, która wykonywałaby za nas większość ciężkiej roboty i naprawdę ułatwiała tworzenie dynamicznych implementacji wirtualnego interfejsu?

W jednym z wcześniejszych rozdziałów, poświęconym typom generycznym, starałem się nauczyć Cię „myśleć generycznie” i wyjaśnić Ci, dlaczego typy te (które wolę nazywać typami parametrycznymi) są pod wieloma względami bardziej przydatne niż kolekcje i listy. Przyjrzałem się klasie TvirtualInterface i pomyślałem: „Jak widać, jest to klasa, która w rzeczywistości wymaga podania informacji o typie interfejsu, i aby za jej pomocą zrobić coś pożytecznego, trzeba w konstruktorze podać ten typ, hmmm”. Zapewne domyślasz się, co potem zrobiłem.

Rozważmy więc następującą deklarację klasy:

```

1 type
2 TRozszerzonyInterfejsWirtualny<T: IInvokable> = class(TVirtualInterface)
3   protected
4     procedure WywołajMetodę(Metoda: TRttiMethod; const Argumenty: TArray<TValue>;
5       out Wynik: TValue);
6     procedure WywołajMetodęImpl(Metoda: TRttiMethod;
7       const Argumenty: TArray<TValue>; out Wynik: TValue); virtual; abstract;
8   public
9     constructor Create;
10  end;
```

Jest to bardzo prosta klasa pochodna od TVirtualInterface. Jej najbardziej oczywistą cechą jest typ parametryczny T, ograniczony do interfejsu pochodnego od IInvokable. W ten sposób można jawnie określić interfejs, który będzie implementować klasa TRozszerzonyInterfejsWirtualny. Zauważ, że jest to klasa abstrakcyjna, ponieważ taka jest jej metoda WywołajMetodęImpl.

Dzięki typowi parametrycznemu mamy wszystko, co jest potrzebne do zaimplementowania interfejsu. Jak wiesz z poprzedniej części rozdziału, wymagana jest implementacja metody WywołajMetodę. W klasie TRozszerzonyInterfejsWirtualny zastosowana jest technika polegająca na implementacji interfejsu w klasie bazowej i wykorzystaniu „realnej” implementacji w osobnej metodzie wywoływanej przez klasę bazową. Implementacja wygląda następująco:

```

1 constructor TRozszerzonyInterfejsWirtualny<T>.Create;
2 begin
3   inherited Create(TypeInfo(T), WywołajMetodę);
4 end;
5
6 procedure TRozszerzonyInterfejsWirtualny<T>.WywołajMetodę(Metoda: TRttiMethod;
7   const Argumenty: TArray<TValue>; out Wynik: TValue);
8 begin
9   WywołajMetodęImpl(Metoda, Argumenty, Wynik);
10 end;
```

Konstruktor jest całkiem prosty — nie ma parametrów i wywołuje odziedziczony konstruktor, podając w jego parametrach metodę `TypeInfo` z interfejsem i metodę `WywołajMetodę` typu `TVirtualInterfaceInvokeEvent`. W kodzie tej metody wywoływana jest po prostu metoda `WywołajMetodęImpl`, która jest metodą abstrakcyjną i dlatego musi być zastąpiona w klasie pochodnej inną metodą.

Aby więc wykorzystać tę klasę, wystarczy jedynie utworzyć jej klasę pochodną i podać interfejs jako typ parametryczny wraz z implementacją metody `WywołajMetodęImpl`. W celu zaimplementowania interfejsu `IPrzydatnyInterfejs` z poprzedniego przykładu wystarczy jedynie wpisać:

```
1 TRozszerzonyPrzydatnyInterfejs = class(TRozszerzonyInterfejsWirtualny
2                                     <IPrzydatnyInterfejs>)
3 protected
4     procedure WywołajMetodęImpl(Metoda: TRttiMethod;
5     const Argumenty: TArray<TValue>; out Wynik: TValue); override;
6 end;
```

Metodę `WywołajMetodęImpl` trzeba zaimplementować za pomocą tego samego kodu co metodę `WywołajMetodę` w klasie `TPrzydatnyInterfejs`.

Opisany sposób to nic specjalnego, ale lubię go, ponieważ upraszcza proces implementacji interfejsu wirtualnego i jest następnym przykładem praktycznego zastosowania typów parametrycznych. Podoba mi się również, że — jak wspomniałem wcześniej — jednoznacznie deklaruje się w nim implementowany interfejs.

## 10.2. Naprawdę użyteczny przykład

Wystarczy już tych nie całkiem przydatnych przykładów. Zgadzam się, że były one obrazowe, ale nie całkiem przydatne w praktyce.

Prawdziwa użyteczność klasy `TVirtualInterface` ujawnia się podczas tworzenia kodu, w którym nie wiadomo, jaki interfejs będzie implementowany przez użytkownika. We wszystkich dotychczasowych przykładach pokazane były klasy implementujące znany interfejs. Wyjątkiem był przykład z klasą `TWirtualnyInterfejsRaportujący`, w którym wyświetlane były informacje o dowolnym interfejsie podanym w argumencie. Ponieważ obiecałem, że klasy `TVirtualInterface` można użyć do zrobienia czegoś przydatnego, zrobmy kolejny krok naprzód.

Praktycznym zastosowaniem klasy `TVirtualInterface` jest utworzenie biblioteki imitacji klas wykorzystywanych do testowania modułów. Wspomniałem wcześniej o platformie `Delphi Mocks Framework`, której autorem jest Vince Parrett, autor słynnego narzędzia `FinalBuilder` (platformę tę opiszę dokładnie w następnym rozdziale). Inną doskonałą platformę testową, wchodzącą w skład pakietu `DSharp`, zbudował Stefan Glienke. W obu produktach wykorzystana jest klasa `TVirtualInterface` do imitowania implementacji dowolnego interfejsu (aczkolwiek kod `DSharp` implementuje własną, bardzo pomysłową wersję tej klasy, która działa w wersji języka Delphi XE). W obu przypadkach można oczywiście wskazać w klasie dowolny interfejs i z powodzeniem testować moduły. A gdyby tak przygotować przykład bardzo prostego obiektu imitującego, który można byłoby wykorzystać do praktycznych testów?

## 10.3. Interfejs `IProstaAtrapa`

W rozdziale „Testy jednostkowe” opiszę terminologię testową. Dokładnie omówię tam różnice pomiędzy atrapami (*stub*) i imitacjami (*mock*) klas. Dowiesz się tam, że **atrapa** jest to „klasa, która nie powoduje uzyskania ani pozytywnego, ani negatywnego wyniku testu i istnieje tylko po to, aby test można było wykonać”. A gdyby tak utworzyć uniwersalną atrapę, czyli klasę, która implementowałaby dowolny interfejs i nie robiła nic więcej? To nie powinno być zbyt trudne, prawda?

Mamy już klasę implementującą interfejs, ale chcemy znaleźć sposób, aby klasa ta stała się interfejsem. Atrapa musi być tego samego typu co testowany interfejs, prawda?

Przed wszystkim, ponieważ zawsze będziemy kodować abstrakcje, należy zadeklarować interfejs:

```
1 IProstaAtrapa<T> = interface
2   ['{6AA7C2F0-E62F-497B-9A77-04D6F369A288}']
3   function WywoływanyInterfejs: T;
4 end;
```

Następnie zaimplementujmy go za pomocą klasy pochodnej od TRozszerzonyInterfejsWirtualny<T>:

```
1 TProstaAtrapa<T: IInvokable> = class(TRozszerzonyInterfejsWirtualny<T>,
2                                     IProstaAtrapa<T>)
3 protected
4   procedure WywołajMetodęImpl(Metoda: TRttiMethod;
5     const Argumenty: TArray<TValue>; out Wynik: TValue); override;
6   public
7     function WywoływanyInterfejs: T;
8 end;
```

Ponieważ klasa TProstaAtrapa<T> pochodzi od TRozszerzonyInterfejsWirtualny<T>, może implementować każdy wskazany w niej interfejs. Zastępowana jest w niej metoda WywołajMetodęImpl klasy TRozszerzonyInterfejsWirtualny<T>, jak również implementowana metoda WywoływanyInterfejs interfejsu IProstaAtrapa<T>.

Najpierw przyjrzyjmy się metodzie WywołajMetodęImpl:

```
1 procedure TProstaAtrapa<T>.WywołajMetodęImpl(Metoda: TRttiMethod;
2     const Argumenty: TArray<TValue>; out Wynik: TValue);
3 begin
4   // Ponieważ jest to atrapa, nie można nic robić!
5 end;
```

Niewiele tu widać — ten kod nic nie robi. Jednak do testów to wystarczy. Dokładnie tego oczekuje się od atrapy klasy — aby nie robiła nic. Nie zajmujemy się skutkami wywołania metod. Ważne jest jedynie, aby można było wywoływać je w testowanym kodzie.

W tym momencie pojawia się funkcja WywoływanyInterfejs. Klasa zna typ testowanego interfejsu, ponieważ jest on przekazywany jako typ parametryczny. Klasa wie sama z siebie, jak zaimplementować zadany interfejs. Powinien zatem istnieć sposób uzyskania referencji do implementowanego interfejsu, prawda?

```
1 function TProstaAtrapa<T>.WywoływanyInterfejs: T;
2 var
3   pInfo : PTypeInfo;
4 begin
5   pInfo := TypeInfo(T);
6   if QueryInterface(GetTypeData(pInfo).Guid, Result) <> 0 then
7     begin
8       raise Exception.CreateFmt('Niestety, TProstaAtrapa<T> nie może zmienić typu %s' +
9         ' na interfejs', [string(pInfo.Name)]);
10    end;
11 end;
```

Ponieważ klasa TProstaAtrapa<T> wie, czym jest typ T, można wywołać funkcję QueryInterface z informacją o typie T i uzyskać referencję do interfejsu. Oczywiście referencję tę można następnie dowolnie wykorzystywać podczas testów interfejsu w ramach testu modułu.

Zatem teraz można bezpiecznie wywoływać metody testowanego interfejsu. Rozważmy następujący interfejs:

```
1 IPrzydatnyInterfejs = interface(IInvokable)
2   ['{16F01BF0-961F-4461-AEBE-B1ACB8D3F0F4}']
3   procedure Witaj;
4   function TekstWstecz(aTekst: string): string;
5   function Iloczyn(x, y: integer): integer;
```

```

6 end;
7
8 {...}
9
10 WriteLn('Implementacja klasy TProstaAtrapa');
11 ProstaAtrapa := TProstaAtrapa<IPrzydatnyInterfejs>.Create;
12 WriteLn('Między tym a poniższym wierszem nie powinno się nic pojawić');
13 ProstaAtrapa.WywoływanyInterfejs.Witaj;
14 ProstaAtrapa.WywoływanyInterfejs.Iloczyn(4, 4);
15 ProstaAtrapa.WywoływanyInterfejs.TekstWstecz('Przykładowy tekst');
16 WriteLn('Między tym a powyższym wierszem nie powinno się nic pojawić');
17 WriteLn;

```

Podczas wywoływania metod tego interfejsu nic się nie dzieje, bo takie jest założenie: atrapa klasy nie powinna nic robić. Ważne jest, aby można było je wywoływać podczas testowania modułu:

```

1 begin
2 {...}
3 MojaKlasaTestowa :=
4   TKlasaImplementującaCokolwiek.Create(ProstaAtrapa.WywoływanyInterfejs)
5 {...}
6 end;

```

## 10.4. Klasa TProstaImitacja

Istnieje zatem przydatny, dynamiczny sposób wykorzystania klasy `TVirtualInterface`. Klasa `TProstaAtrapa<T>` świetnie się sprawdzi podczas testów, z których ma nie wynikać absolutnie nic. Jednak czasami potrzebny jest interfejs, który robi coś więcej, niż tylko istnieje. W takim przypadku tworzy się imitację klasy. W rozdziale poświęconym testowaniu modułów **imitacja** zdefiniowana jest jako „sztuczna klasa, udająca działanie testowanej klasy. W zależności od jej działania wynik testu może być pozytywny lub negatywny”. Imitacja zatem musi nie tylko istnieć, jak atrapa, lecz także coś robić, czyli działać w określony przez programistę sposób.

Jedną z operacji najczęściej wykonywanych przez imitację jest zwracanie zdefiniowanego w jej kodzie wyniku po umieszczeniu w argumentach określonych danych. A gdyby tak utworzyć prostą imitację, która umożliwiłaby definiowanie działania wywoływanej metody?

Oczywiście najpierw trzeba utworzyć kod interfejsu:

```

1 IProstaImitacja<T> = interface(IProstaAtrapa<T>)
2   ['{9619542B-A53B-4C0C-B915-45ED140E6479}']
3   procedure DodajWynik(aNazwaMetody: string; aWynik: TValue);
4 end;

```

Interfejs ten pochodzi od interfejsu `IProstaAtrapa<T>` (pamiętaj, że w przypadku interfejsów termin „pochodzenie” nie odpowiada ściśle rzeczywistości) i dodaje metodę `DodajWynik`. Jest to metoda, która będzie wykorzystana do definiowania wyniku wywoływanej metody interfejsu.

Poniżej przedstawiona jest klasa implementacyjna:

```

1 TProstaImitacja<T: IInvokable> = class(TProstaAtrapa<T>, IProstaImitacja<T>)
2 private
3   FWyniki: TDictionary<string, TValue>;
4 protected
5   procedure WywołajMetodęImpl(Metoda: TRttiMethod;
6     const Argumenty: TArray<TValue>; out Wynik: TValue); override;
7 public
8   constructor Create;
9   destructor Destroy; override;
10  procedure DodajWynik(aNazwaMetody: string; aWynik: TValue);
11 end;

```



Przede wszystkim należy zwrócić uwagę, że klasa TProstaImitacja<T> pochodzi od klasy TProstaAtrapa<T>, zatem może implementować dowolny interfejs. Oczywiście implementuje ona również metodę DodajWynik. Parametrem tej metody jest nazwa wywoływanej metody interfejsu oraz wartość, która ma być przez nią zwracana. W ten sposób można dowolnie zdefiniować działanie klasy testowej.

W tym bardzo prostym przykładzie przyjęte jest założenie, że testowane będą tylko funkcje i metody interfejsu. W ramach tego przykładu nie ma potrzeby testowania procedur, które z zasady nie wykonują żadnych operacji, przynajmniej w tym prostym przykładzie. Jak się przekonasz, w pełni funkcjonalna platforma testowa umożliwia śledzenie, czy dana procedura była wywoływana i ile razy, jak również umożliwia zbieranie innych informacji o procedurach. W tym przykładzie nie są uwzględniane parametry przekazywane metodzie, zwracany jest jedynie jej wynik. Pamiętaj, że jest to prosty, ale w pewnych sytuacjach przydatny przykład.

Implementacja klasy TProstaImitacja<T> jest całkiem łatwa. Wewnątrz niej do śledzenia nazw wywoływanych metod i zwracanych przez nie wartości jest wykorzystywany słownik TDictionary<TKey, TValue>. Dane do niego są wpisywane za pomocą metody DodajWynik. Poniżej przedstawiona jest jej implementacja:

```
1 procedure TProstaImitacja<T>.DodajWynik(aNazwaMetody: string; aWynik: TValue);
2 begin
3   FWyniki.Add(aNazwaMetody, aWynik);
4 end;
```

Dane te po dodaniu do słownika są wykorzystywane przez klasę do śledzenia wywoływanych metod. Jeżeli zostanie wywołana jakaś metoda interfejsu, wtedy klasa pobiera ze słownika odpowiedni wynik i zwraca go:

```
1 procedure TProstaImitacja<T>.WywołajMetodęImpl(Metoda: TRttiMethod;
2   const Argumenty: TArray<TValue>; out Wynik: TValue);
3 begin
4   Wynik := FWyniki[Metoda.Name];
5 end;
```

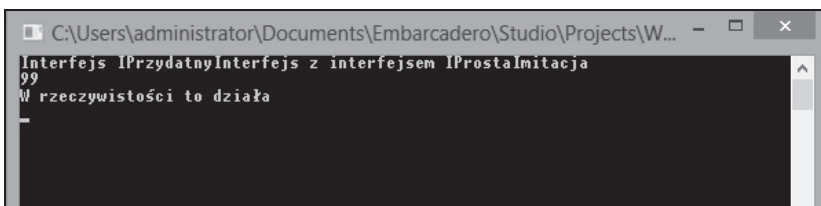
Oczywistą wadą powyższego kodu jest brak obsługi błędów. Jeżeli zostanie wywołana metoda interfejsu, dla której nie jest dostępna spodziewana zwracana wartość, wtedy zostanie zgłoszony wyjątek. Inny mankament polega na tym, że nie są uwzględniane wartości parametrów. Platforma testowa z prawdziwego zdarzenia musi umożliwiać definiowanie odpowiedzi w zależności od wartości podanych parametrów. Rozwiązanie tego problemu pozostawiam jako ćwiczenie dla Czytelnika.

Teraz zatem, podczas korzystania z tej klasy, zwracane będą zadane dane.

Poniższy kod:

```
1 WriteLn('Interfejs IPrzydatnyInterfejs z interfejsem IProstaImitacja');
2 ProstaImitacja := TProstaImitacja<IPrzydatnyInterfejs>.Create;
3 ProstaImitacja.DodajWynik('Iloczyn', 99);
4 ProstaImitacja.DodajWynik('TekstWstecz', 'W rzeczywistości to działa');
5 WriteLn(ProstaImitacja.WywoływanyInterfejs.Iloczyn(6, 7));
6 WriteLn(ProstaImitacja.WywoływanyInterfejs.TekstWstecz(
7   'Argumenty nie mają znaczenia'));
8 WriteLn;
```

powoduje wyświetlenie następujących informacji:



```
C:\Users\administrator\Documents\Embarcadero\Studio\Projects\W... - [x]
Interfejs IPrzydatnyInterfejs z interfejsem IProstaImitacja
99
W rzeczywistości to działa
```

Zwróć uwagę, że nie są to odpowiedzi, których należy się spodziewać, biorąc pod uwagę wartości parametrów (spodziewanym wynikiem mnożenia 6 przez 7 jest liczba 42), ale informacje, które wskaże się za pomocą metody `DodajWynik`.

Teraz można wykorzystać interfejs `IProstaImitacja<T>` do określenia odpowiedzi wywoływanej metody. Być może chciałbyś przetestować metodę zwracającą wartość `Boolean`. Możesz użyć interfejsu `IProstaImitacja<InterfejsZMetodaBoolean>` do sprawdzenia, co się stanie, gdy metoda ta zwróci wartość `True` lub `False`.

## 10.5. Wnioski

Mamy zatem coś: przydatną implementację klasy `TVirtualInterface`. Choć przedstawione tu przykłady są naprawdę proste, można je wykorzystać do testów w praktyce, szczególnie implementację interfejsu `IProstaAtrapa<T>`. Atrapy klas są często stosowane podczas testów modułów i chociaż przedstawiona implementacja jest bardzo prosta, może być użyta do testowania każdego interfejsu.

Pamiętaj, że klasę tę można wykorzystać wtedy, jeżeli znany jest interfejs i sposób jego implementacji. Są jednak przypadki, w których nie wiadomo, jaki interfejs będzie potrzebny do rozwiązania określonego problemu, i trzeba sprawdzić, który z nich okaże się najlepszy w określonej sytuacji. Atrapy i imitacje klas doskonale się do tego celu nadają. Są to potężne i przydatne narzędzia. Mam nadzieję, że niniejszy rozdział utwierdził Cię w tym przekonaniu.

# Skorowidz

## A

abstrakcja, 34  
abstrakt, 31  
antywzorzec, 147  
atrapa, 116, 155, 169, 170, 176  
zwracająca wartość, 173  
atrybut, 103, 104, 105, 107, 109, 145  
deklaracja, 104  
Inject, 146  
nieznany, 107  
Setup, 158  
SetupFixture, 158  
TearDown, 158  
TearDownFixture, 158  
Test, 158, 164  
TestFixture, 158, 164  
autołączenie, 144

## B

Beck Kent, 162  
biblioteka  
DLL, 18  
imitacji klas, 116  
OmniThread, *Patrz:* OmniThread  
RTL, 28, 33  
Spring4D, *Patrz:* Spring4D  
uruchomieniowa, 44, *Patrz:* RTL  
zgłaszanie wyjątków, 22  
blok  
try, 18  
try...catch, 20

## C

Class Under Test, *Patrz:* klasa testowana  
Code Insight, 11, 39  
CUT, *Patrz:* klasa testowana

## D

DataSnap, 11  
Delphi Mocks Framework, 16, 111, 156, 170, 182  
Delphi Runtime Library, *Patrz:* RTL  
Delphi Sorcery Framework, 182

Delphi Spring Framework, 182  
destruktor, 44  
diagram Venna, 122  
domknięcie, 55  
DSharp, 16  
DUnit, 154, 158, 164  
DUnitX, 15, 154, 158, 163, 164, 182  
dyrektywa, 100  
\$M+, 112, 171  
\$METHODINFO, 101  
\$RTTI, 100, 101  
\$STRONGLINKTYPES, 102  
\$WEAKLINKRTTI, 100  
dziedziczenie  
interfejsów, *Patrz:* interfejs dziedziczenie  
klas, *Patrz:* klasa dziedziczenie

## E

enumerator, 75, 76, 77, 78, 84

## F

FinalBuilder, 156, 170  
FireMonkey, 11  
funkcja, *Patrz:* metoda  
funkcjonalność, 26, 27  
definicja abstrakcyjna, *Patrz:* interfejs  
RTTI, *Patrz:* RTTI

## G

Gamma Erich, 34  
Git, 182  
GUID, 28

## H

Hanselman Scott, 155  
hierarchia klas, *Patrz:* klasa hierarchia

## I

identyfikator GUID, *Patrz:* GUID  
imitacja, 116, 118, 155, 156, 169, 172, 176, 177, 178, 179  
konfigurowanie, 173  
informacje RTTI, *Patrz:* RTTI  
instrukcja  
for...in, 66, 75, 76  
try...finally, 58  
uses, 26, 123  
using, 58  
with, 58  
interfejs, 25, 26, 27, 34, 44, 111, 122, 151  
deklarowanie, 26, 28, 31  
dziedziczenie, 28, 30  
generyczny, 43  
IComparer, 44  
identyfikator, 28  
IEnumerable, 73, 81, 82, 83, 86, 87  
deklaracja, 82  
IEnumerator, 77  
IInterface, 32, 112  
IInvokable, 171  
IList, 82  
implementacja, 26, 28, 29, 30, 31, 111, 116, 142, 155, 170  
dynamiczna, 111  
zmiana, 35  
instancja, 28  
izolowanie, 129  
kod, 27  
pamięć, *Patrz:* pamięć zajmowana przez interfejsy  
pochodny, 30  
segregacja, *Patrz:* zasada segregacji interfejsów  
w Delphi, 26

## J

jednostka, 154

## K

Kelly Barry, 183  
 klasa, 26, 92  
 abstrakcyjna, 27  
 bazowa, 29, 30  
 dziedziczenie, 28  
 Exception, 21  
 generyczna, 11  
 TDictionary, 44  
 TList, 44, 45  
 TObjectDictionary, 44  
 TObjectList, 44  
 TObjectStack, 44  
 TQueue, 44  
 TStack, 44  
 TThreadedList, 44  
 TThreadedQueue, 44  
 hierarchia, 123  
 implementacyjna, 29, 30, 111  
 implementowanie kilku  
 interfejsów, 142  
 metadane, *Patrz:* metadane  
 pochodna, 111, 112  
 pozorna, 169, 176  
 ServiceLocator, 132, 134, 139, 141,  
 147, 148, 150  
 sprzężanie, 121, *Patrz też:* kod  
 sprzężony  
 sztuczna, 155, 156  
 TAggregatedObject, 33  
 TApplication, 20  
 TClientDataset, 19  
 TCollections, 72, 73  
 TContainedObject, 33  
 TCustomAttribute, 103  
 TEnumerator, 80  
 testowa, 161  
 testowana, 154  
 TInterfacedObject, 32, 33, 112  
 TMock, 172, 176  
 TObject, 103  
 TRegistration, 140  
 TRttiContext, 93  
 TRttiField, 94  
 TRttiInterfaceType, 100  
 TRttiMethod, 95  
 TRttiProperty, 95  
 TRttiType, 93, 94, 100  
 TTestCase, 158  
 TVirtualInterface, 111, 112, 115,  
 116, 118, 170  
 klucz, 66  
 kod  
 abstrakt, *Patrz:* abstrakt  
 interfejsu, *Patrz:* interfejs kod  
 rozprzężanie, 25, 26, 27, 111, 124,  
 129, 131

rozprzężony, 25, 34  
 sprzężanie, 26  
 sprzężony, 121  
 luźno, 34, 35, 124  
 za bardzo, 35  
 wzorzec, 36  
 źródłowy, 182  
 kolejka, 66, 69  
 kolekcja, 44, 65  
 element, 84, 85, 86  
 generyczna, 66, 73  
 ICollection, 72, 73  
 IDictionary, 73  
 IList, 72, 73  
 instancja, 84  
 IQueue, 72, 73  
 ISet, 73  
 IStack, 73  
 niegeneryczna, 72, 80  
 pochodna, 72  
 podzbiór, 81  
 TDictionary, 66, 70  
 TList, 66  
 TObjectDictionary, 66, 72  
 TObjectList, 66, 72  
 TObjectQueue, 66, 72  
 TObjectStack, 66, 72  
 TQueue, 66, 69  
 TStack, 66, 68, 69  
 TThreadedQueue, 66  
 TThreadList, 66  
 kompilator, 28, 32, 76  
 dyrektywa, *Patrz:* dyrektywa  
 ustawienia domyślne, 89  
 konstruktor, 44, 121, 123  
 Create, 40, 76, 131  
 wstrzykiwanie, *Patrz:*  
 wstrzykiwanie konstruktora  
 kontener  
 Spring Container, *Patrz:* moduł  
 Spring.Container  
 wstrzykiwania zależności, *Patrz:*  
 wstrzykiwanie zależności  
 kontener  
 kontrawariancja, 48  
 kowariancja, 48

## L

lista, 66

## M

Mercurial, 182  
 Meszaros Gerard, 155  
 metadane, 89  
 metoda, 89

\_AddRef, 31, 32  
 \_Release, 31, 32  
 Add, 73  
 All, 85  
 anonimowa, 11, 51, 52, 57, 58, 59,  
 61, 63, 141  
 deklarowanie, 52  
 jako zmienna, 55  
 Any, 85  
 AsObject, 84  
 AsPooled, 140, 141  
 Assert.AreEqual, 161  
 Assert.IsTrue, 161  
 AsSingleton, 140  
 AsSingletonPerThread, 140, 141  
 AsTransient, 140, 141  
 CheckEquals, 158, 161  
 CheckNotEquals, 158  
 CheckTrue, 161  
 Concat, 86  
 Contains, 85  
 Count, 86  
 CreateAnonymousThread, 61  
 DelegateTo, 141  
 DoCurrent, 80  
 DoGetCurrent, 80  
 DoGetEnumerator, 80  
 DoMoveNext, 80  
 ElementAt, 85  
 EqualsTo, 86  
 Expect, 173  
 First, 84, 85  
 FirstOrDefault, 85  
 ForEach, 86  
 generyczna, 43, 44  
 GetAttributes, 107  
 GetCurrent, 78  
 GetDeclaredMethods, 95  
 GetEnumerator, 75, 76, 78, 84  
 GetMethods, 95  
 GetProperties, 95  
 InjectConstructor, 146, 148  
 InjectField, 146, 148  
 InjectMethod, 148  
 InjectProperty, 146, 148  
 Invoke, 98  
 IsClassMethod, 98  
 IsEmpty, 86  
 IsManaged, 100  
 IsOrdinal, 100  
 IsRecord, 100  
 IsSet, 100  
 IsStatic, 98  
 klasy, 98  
 LastOrDefault, 85  
 łączenie w ciąg, 84  
 Max, 85

Min, 85  
 MoveNext, 75  
 odczytująca dane, 29  
 Peek, 69  
 private, 29  
 przeciążona, 158  
 przypisująca, 136  
 QueryInterface, 32, 117  
 Queue, 61  
 RegisterType, 139  
 rejestrująca, 139  
 Remove, 73  
 Reversed, 86  
 Setup, 158  
 Single, 85  
 SingleOrDefault, 85  
 Skip, 85  
 SkipWhile, 85  
 statyczna, 98  
 Take, 85  
 TakeWhile, 84, 86  
 TClass, 96  
 TearDown, 158  
 testowa, 158  
 ToArray, 86  
 ToList, 86  
 ToSet, 86  
 TryGetFirst, 84  
 TryGetLast, 84  
 TValue, 91  
 TypeInfo, 112  
 Verify, 173, 178  
 warunek konieczny, 17  
 When, 173  
 Where, 85  
 WillReturn, 173  
 wstrzykiwanie, *Patrz:*  
   wstrzykiwanie metody  
   wysyłająca komunikat, 44  
   wywoływanie, 97  
   wywoływanie metody, 122  
 mock, *Patrz:* imitacja  
 Model-View-Controller, *Patrz:* MVC  
 Model-View-ViewModel, *Patrz:* MVVM  
 moduł  
   DUnitX.Loggers.Console, 165  
   DUnitX.TestFramework, 164, 165  
   DUnitX.Windows.Console, 165  
   Generics.Collections.pas, 44  
   komunikacja, 35  
   RTTI.pas, 107  
   Spring.Collections.pas, 44, 72, 73  
   Spring.Container, 131, 135, 138,  
     141, 145  
   Spring.Services, 132  
   sprzężenie luźne, 25  
   System.Generics.Collections, 66

System.pas, 103  
 TypInfo.pas, 46  
 MVC, 36  
 MVVM, 36

## O

obiekt  
   czas życia, 140  
   domyślny, 142  
   tworzenie, 141  
 odwrócenie sterowania, 121  
 ograniczenie, 39, 42  
   class, 40, 41  
   constructor, 40, 41  
   interface, 42  
   record, 41  
 OmniThread, 182  
 operator rekordów, 44  
 Osherove Roy, 155

## P

palindrom, 68  
 pamięć zajmowana przez interfejs, 31  
 Parrett Vince, 156, 170  
 platforma  
   imitacyjna, 154, 170  
   izolacyjna, 155, 169, 170  
   testowa, 164  
 plik  
   binarny, 89, 104  
   DPR, 100  
   FMX, 100  
   VCL, 100  
 pole, 89, 121  
 polimorfizm, 39  
 prawo Demeter, 122, 123  
 predykat, 61, 62, 83  
 program testujący, 154  
 programowanie  
   abstrakcyjne, 34, 35  
   uwzględniające testy, *Patrz:* TDD  
   założenia, 17

## R

refaktoryzacja, 166, 167  
 referencja, 28  
   do interfejsu, 33, *Patrz:* referencja  
     zliczana  
   do obiektu, 33  
   śledzenie, 32  
   zliczana, 31  
 rejestrowanie  
   generyczne, 139  
   interfejsu, 139  
   klas, 139

rekord, 41, 98, 99, 100  
   operator, *Patrz:* operator  
     rekordów  
   przekazywany jako typ  
     parametryczny, 42  
   TRttiContext, 93  
   TValue, 90, 91  
 RTL, 23, 56, 57, 61, 73, 80  
 RTTI, 89, 92, 93, 96, 98, 99, 100, 104,  
   131, 158  
 Run-time Type Information,  
   *Patrz:* RTTI

## S

sekcja  
   implementation, 27, 123  
   initialization, 165  
   interface, 26, 27  
 singleton, 141  
 słownik, 66, 70, 119  
 Spring Framework for Delphi, *Patrz:*  
   Spring4D  
 Spring4D, 11, 15, 44, 72, 73  
 stos, 37, 38, 66, 68  
   wskaźników, 37  
 stub, *Patrz:* atrapa  
 Subversion, 182  
 superklasa, 45  
 SUT, *Patrz:* system testowany  
 system testowany, 154  
 System Under Test, *Patrz:* system  
   testowany

## T

tabela, 99  
 TDD, 162  
 test  
   granic, 159  
   integracyjny, 154  
   jednostkowy, 153, 154, 156, 157, 181  
   sprawdzający zakresy, 159, 160  
   tworzenie, 159  
   nazwa, 161  
   niezależność, 160  
   pokrycia kodu, 160  
   regresyjny, 160  
 Test Driven Development, *Patrz:*  
   TDD  
 typ  
   generyczny, 37, 38, 43, 45, 46, 48,  
     49, 65  
   ograniczenie, *Patrz:*  
     ograniczenie  
       z klasami, 37  
   ICollection, 73  
   IEnumerable, 84  
   Integer, 90

typ

- ISet, 73
- jako parametr, 38
- konwersja, 38, 91
- parametryczny, *Patrz:* typ generyczny
- porządkowy, 99, 100
- PTypeInfo, 113
- TEnumerator, 80
- TObject, 40
- TRttiType, 93
- TValue, 90, 91
- variant, 90
- wyliczeniowy, 46, 98
- zarządzalny, 100

**U**

Urlocker Zack, 28

**V**

Venna diagram, *Patrz:* diagram Venna

**W**

Wheeler Mason, 183  
 właściciel, 100  
 właściwość, 29, 44, 89, 121

- ClassType, 98
- FreeOnTerminate, 61
- OwnsObjects, 72
- Parent, 100
- WillReturn, 175
- wskaźnik, 38
- wstrzykiwanie
  - konstruktora, 135, 137, 144, 145
  - metody, 137, 144, 145
  - przypisującej, 137
  - metody przypisującej, 136
  - pola, 142, 145
  - właściwości, 142, 145
- wstrzykiwanie zależności, 36, 121, 124, 135, 151, 181
- kontener, 131
- wyjątek, 17, 20, 178
- EConvertError, 20
- EDatabaseError, 20, 21
- EMathError, 20
- EOutOfMemory, 22
- komunikat, 23
- niestandardowy, 22
- obsługa, 17, 19, 20
  - centralna, 20
  - strukturalna, 17
- polykanie, 18, 21
- przechwytywanie, 18, 21, 22, 23
- Unsatisfied Dependency, 142

- związany z obsługą bazy danych, 20, 21
- wzorzec
  - IEnumerable, 44
  - testowy, 155

**Z**

- zależność, 121, 170, 176
  - nieokreślona, 142
  - tworzenie, 131
  - wstrzykiwanie, *Patrz:* wstrzykiwanie zależności
  - zgłaszająca wyjątek, 178
- zasada
  - AAA, 159
  - enkapsulacji, 38
  - jednej odpowiedzialności, 138
  - polimorfizmu, *Patrz:* polimorfizm segregacji interfejsów, 31
- zbiór, 100
- zdarzenie, 59
  - deklaracja, 59
  - OnException, 20
  - OnReconcileError, 19
- zmienna, 29
  - metoda anonimowa, *Patrz:* metoda anonimowa jako zmienna
- znak średnika, 52

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ  
2. PREZENTUJ KSIĄŻKI  
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

**Wbrew pozorom** na świecie pracuje bardzo wielu programistów posługujących się językiem Delphi. Sęk w tym, że większość z nich nie ma pojęcia o niezwykłych i użytecznych funkcjonalnościach dostępnych w nowszych wersjach tego języka oraz nie umie zastosować tych narzędzi we własnej pracy. Jeśli czujesz, że i Ty zaliczasz się do tego grona, a brak umiejętności sprawia, że nie możesz rozwinąć skrzydeł, ta książka jest dla Ciebie. Opisano w niej niezwykle przydatne narzędzia oraz sytuacje, w których te narzędzia sprawdzają się szczególnie dobrze, a także kilka ogólnie dostępnych platform.

**W gruncie rzeczy ta książka** jest poświęcona wyłącznie tworzeniu nowego, dobrego kodu w języku Delphi. Nie znajdziesz w niej nic o projektowaniu okienek, języku VCL ani platformie FMX, ale jeśli chcesz się dowiedzieć więcej o kodowaniu interfejsów zamiast implementacji, o właściwych sposobach korzystania z wyjątków i ich obsłudze, o testowaniu i poprawianiu kodu, o zastosowaniu platformy String for Delphi do lepszego zarządzania kolekcjami danych albo o uzyskiwaniu wglądu w kod podczas jego wykonywania za pomocą nowego, potężnego narzędzia RTTI, niniejsza publikacja z pewnością Cię zachwyci.

- Wyjątki i ich obsługa
- Interfejsy i typy generyczne
- Metody anonimowe
- Kolekcje i enumeratory w Delphi
- Interfejs IEnumerate
- Informacje RTTI i atrybuty
- Klasa TVirtualInterface
- Wstrzykiwanie zależności
- Testy jednostkowe
- Testy z użyciem platformy izolacyjnej

**Wykorzystaj w pełni moc i elegancję Delphi!**

**Nick Hodges** — programista zaangażowany w projekt Delphi od początku jego istnienia. Był testerem pierwszej wersji języka i członkiem zespołu TeamB, a także menedżerem produktu i kierownikiem działu badawczo-rozwojowego języka Delphi. Jest członkiem Advisory Board na corocznej konferencji firmy Borland, aktywnym prelegentem, blogerem i autorem artykułów poświęconych różnym zagadnieniom języka Delphi.



sięgnij po **WIĘCEJ**



KOD KORZYŚCI

**Helion**

38390 numer katalogowy  
księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:  
● <http://helion.pl/promocje>  
Książki najchętniej czytane:  
● <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
● <http://helion.pl/nowosci>

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

ISBN 978-83-283-0797-1



9 788328 307971

Informatyka w najlepszym wydaniu

cena: 49,00 zł