Nick Hodges

PROGRAMOWANIE W JĘZYKU 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

Helion

Tytuł oryginału: Coding in Delphi

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-0797-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *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.

Spis treści

ROZDZIAŁ

10

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 niezwykłej 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('Wywołałeś 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, `PIID`, 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 `PIID`, 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.

Załóżmy więc, że zadeklarowany jest następujący interfejs:

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

Następnie uruchomiona została poniższa aplikacja konsolowa:

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

Zawartość okna wiersza poleceń będzie następująca:

```
Wywołałeś metodę interfejsu
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 wywoływaną metodą jest funkcja.

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

```
procedure TWirtualnyInterfejsRaportujący.WywołajMetodę(Metoda: TRttiMethod;
  const Argumenty: TArray<TValue>; out Wynik: TValue);
var
  Argument: TValue;
  TypArgumentu, NazwaArgumentu: string;
  TymczTyp: TTypeKind;
begin
  Write('Wywołałeś metodę ', Metoda.Name);
  if Length(Argumenty) > 1 then
  begin
    WriteLn(', która ma ', Length(Argumenty) - 1,' parameterów:');
    for Argument in Argumenty do
    begin
      TymczTyp := Argument.Kind;
```

```
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.');
25   end;
26 end;
```

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-AEBE-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!');
7   end else
8   begin
9     if UpperCase(Metoda.Name) = 'TEKSTWSTECZ' then
10     begin
11       Wynik := ReverseString(Argumenty[1].AsString)
12     end else
13     begin
14       if UpperCase(Metoda.Name) = 'ILOCZYN' then
15       begin
16         Wynik := Argumenty[1].AsInteger * Argumenty[2].AsInteger;
17       end else
18       begin
19         raise Exception.Create('Błędny parametr metody WywołajMetodę');
20       end;
21     end;
22   end;
23 end;
```

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 `Tvirtual ↪Interface` 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:

```
type
 TRozszerzonyInterfejsWirtualny<T: IInvokable> = class(TVirtualInterface)
  protected
    procedure WywołajMetodę(Metoda: TRttiMethod; const Argumenty: TArray<TValue>;
      out Wynik: TValue);
    procedure WywołajMetodęImpl(Metoda: TRttiMethod;
      const Argumenty: TArray<TValue>; out Wynik: TValue); virtual; abstract;
  public
    constructor Create;
 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:

```
constructor TRozszerzonyInterfejsWirtualny<T>.Create;
begin
  inherited Create(TypeInfo(T), WywołajMetodę);
end;

procedure TRozszerzonyInterfejsWirtualny<T>.WywołajMetodę(Metoda: TRttiMethod;
  const Argumenty: TArray<TValue>; out Wynik: TValue);
begin
  WywołajMetodęImpl(Metoda, Argumenty, Wynik);
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, zróbmy 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 `Tvirtual`
`↪Interface` 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?

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

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

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

```
TProstaAtrapa<T: IInvokable> = class(TRozszerzonyInterfejsWirtualny<T>,
                                     IProstaAtrapa<T>)
protected
  procedure WywołajMetodęImpl(Metoda: TRttiMethod;
    const Argumenty: TArray<TValue>; out Wynik: TValue); override;
  public
    function WywoływanyInterfejs: T;
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`:

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

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

```
IPrzydatnyInterfejs = interface(IInvokable)
  ['{16F01BF0-961F-4461-AEBE-B1ACB8D3F0F4}']
  procedure Witaj;
  function TekstWstecz(aTekst: string): string;
  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 umożliwiałaby 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;
```

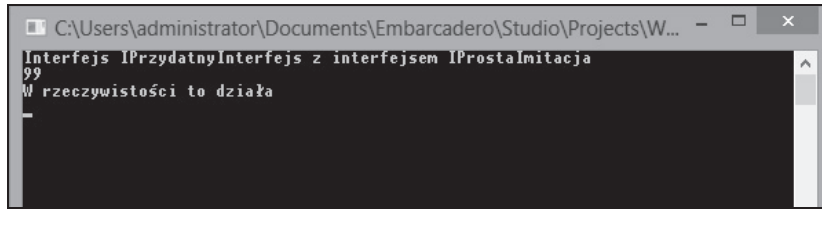
Oczywistą wadą powyższego kodu jest brak obsługi błędów. Jeżeli zostanie wywołana metoda interfejsu, dla której nie jest dostępna spodziewana zwracana wartość, wtedy zostanie zgłoszony wyjątek. Inny mankament polega na tym, że nie są uwzględniane wartości parametrów. Platforma testowa z prawdziwego zdarzenia 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:



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<InterfejsZMetodąBoolean>` 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

K

L

M

U

V

W

Z

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 IEnumerable
- 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
http://helion.pl

ISBN 978-83-283-0797-1

9 788328 307971

Informatyka w najlepszym wydaniu

cena: 49,00 zł