Wykład IV – tworzenie „własnych” komponentów

Tworzenie nowych – własnych komponentów to kwintesencja programowania obiektowego. Własne, optymalnie dopasowane do programu, komponenty pozwalają uprościć jego budowę, usprawnić przeróbki i sprawdzanie. Zmniejszają zasadniczo zjawisko zbędnego „powielania kodu”. Każdy napisany komponent można w prosty sposób wykorzystać w przyszłych aplikacjach. W wykładzie I przedstawiona została definicja komponentu. Z punktu widzenia programisty istotna jest jeszcze informacja, że kody źródłowe zdecydowanej większości komponentów są dostępne w katalogu istalacyjnym środowiska programistycznego (zwykle *C:\Program Files\Borland\Delphi7\Source\Vcl*) .
Ponieważ kluczową dodatkową możliwością komponentów w stosunku do prostszych obiektów jest możliwość zmiany ich właściwości w czasie projektowania w *Inspektorze Obiektów* zagadnienie tworzenia nowych komponentów rozpocząć należy od przedstawienia mechanizmu właściwości.

Właściwości

Właściwości komponentu wzbogacają go o wspomnianą dodatkową funkcjonalność – możliwość zmiany jego stanu na etapie projektowania. Wiąże się z tym jedna cecha właściwości widoczna od samego początku – ich obecność w *Inspektorze Obiektów*, jak i druga cecha nieco ukryta – zapisywanie wartości właściwości do pliku wykonywalnego czyli permanentne ich zapamiętywanie. Dodatkowo, aby umożliwić sterowanie tymi mechanizmami (możliwość ich włączenia i wyłączenia dla właściwości) wprowadzona została nowa sekcja do budowy obiektu. Oprócz obecnych wcześniej sekcji *private*, *protected* i *public* komponenty charakteryzuje istnienie sekcji ***published***. Wspomniane wyżej cechy właściwości są aktywne tylko dla właściwości umieszczonych w tej sekcji.
Cofając się do podstaw programowania obiektowego – obiekt składa się z pól i metod wykonujących operacje na tych polach. Enkapsulacja wymusza też, aby dostęp do pól z „zewnatrz” obiektu był ograniczony do pewnych specjalnych metod – metod dostępowych. Zgodnie z tym pole obiektu powinno być umieszczone w części prywatnej obiektu, a metody dostępowe – w publicznej. Od strony programistycznej właściwość to po prostu połączenie pola i jego metod dostępowych w jeden konglomerat. Tak jak w przypadku pól, właściwości mogą być różnych typów – oprócz tak oczywistych jak napisy, liczby zmienne logiczne polem może być typu proceduralnego, funkcyjnego i obiektowego. Właściwości można więc zgrubnie podzielić na trzy grupy:

1. właściwości proste (napisy, liczby itp. Również stablicowane),
2. zdarzenia (typy funkcyjne i proceduralne),
3. właściwości obiektowe.

Wszystkie te typy właściwości podlegają tym samym prawom ogólnym, różnią się zasadniczo w szczegółach implementacyjnych. Wszystkie muszą przede wszystkim zawierać definicją właściwości przy pomocy słowa kluczowego ***property*** oraz mogą zawierać pole i dwie metody dostępowe (do zapisu i odczytu). Wszystkie właściwości mogą występować indywidualnie, bądź w strukturach tablicowych.

Właściwości proste

Właściwości proste definiują, jak to już było wspomniane, wartości liczbowe, napisy, wartości logiczne i typy wyliczeniowe. Definicję właściwości zaczyna się poprzez umieszczenie w części *published* obiektu słowa kluczowego property, poprzedzającego

definicję właściwości, na którą składa się nazwa, definicja typu i sposobu zapisu i odczytu właściwości. Dla przykładu kod poniżej definiuje właściwość *Wiek* :

```
published
 property Wiek:byte |...brak...  |  |...brak...   |;
                    |read FWiek  |  |write FWiek  |
                    |read GetWiek|  |write SetWiek|
```

Sposób zapisu i odczytu właściwości można definiować na trzy sposoby:
- pominąć (właściwość tylko do odczytu lub tylko do zapisu),
- użyć bezpośrednio pola (nieestetyczne – stosować tylko w specyficznych sytuacjach),
- użyć metody dostępowej.

Preferowana jest metoda ostatnia, ale wszystkie kompilator dopuści.
W zależności od przyjętego rozwiązania należy następnie zdefiniować (lub nie) jedna lub dwie metody dostępowe i odpowiednie (tego samego co właściwość typu) pole. Wszystkie te definicje umieszcza się zwykle w części prywatnej obiektu. W najpełniejszej wersji mogą one wyglądać następująco:

```
private
 FWiek:byte
 procedure SetWiek(value:byte);
 function GetWiek:byte;
```

// nazwę pola poprzedza sięzwyczajowo literą F
// nazwę procedury poprzedza się zwyczajowo przedrostkiem Set*, funkcji* Get*. Parametr nosi zwyczajową nazwę* value*. I parametr i wynik funkcji musi być tego typu, który użyty został w definicji właściwości.*

Metody dostępowe mają za zadanie przede wszystkim w kontrolowany sposób udostępnić pole obiektu. O ile działanie funkcji odczytującej sprowadza się zwykle jedynie do zwrócenia w formie wyniku wartości pola i można ją w ogóle pominąć (użyć definicji właściwości z bezpośrednim dostępem do pola przy odczycie), to procedura zapisujaca powinna skontrolować wartość właściwości, i jeśli jest ona dopuszczalna – zapisać ją do pola i uaktualnić odpowiednio stan obiektu (np. odrysować obiekt ze zmienionym podpisem gdy właściwość go definiuje). Metody dostępowe przykładowo mogą więc mieć postać:

```
procedure TMojObiekt.SetWiek(value:byte);
 begin
  if (sprawdzenie czy wartość odpowiednia) then
   begin
    FWiek:=value;
    //tutaj uaktualnienie obiektu
   end;
 end;

function TMojObiekt.GetWiek:byte;
 begin
  result:=FWiek;
 end;
```

W specyficznych sytuacjach mogą być tworzone właściwości nie posiadające własnego pola. Są to zwykle właściwości tylko do odczytu, posiadające metodę obliczającą wartość właściwości np. z innych pól. Przykładem może być właściwość zwracająca pole powierzchni obiektu prostokąt:

```
...
...
private
 Fa:single;
 Fb:single;
 function GetPolePowierzchni:single;
public
 property PolePowierzchni:single read GetPolePowierzchni;
...
...

implementation

function TProstokąt.GetPolePowierzchni:single;
begin
 result:=Fa*Fb;
end;
```

Zdarzenia

Zdarzenia, to właściwości, które połączone są z polami typu proceduralnego lub funkcyjnego. Ich definicja różni się od definicji właściwości prostych jedynie tym, że typem danych użytym w definicji jest właśnie typ proceduralny lub funkcyjny. Pierwszym krokiem jest tu więc zdefiniowanie odpowiedniego typu proceduralnego lub funkcyjnego. Jego definicja jest zbliżona do definicji wszystkich innych typów:

```
TMojeZdarzenie = function(Sender:TObject; DodatkowyPar:integer):boolean of
object;
```

Można wykorzystać któryś z predefiniowanych typów, jak choćby szeroko stosowany TNotifyEvent, którego definicja wygląda następująco:

```
TNotifyEvent = procedure(Sender:TObject) of object;
```

Parametr *Sender* wskazujący na obiekt, jaki wywołał procedurę zdarzenia, jest wskazany, ale nie obligatoryjny. Zapis *of object* określa, że typ odnosi się do metody obiektu, a nie zwykłej procedury.
Znając już typ właściwości (po zdefiniowaniu go lub po wybraniu predefiniowanego) zdarzenie tworzy się analogicznie jak w przypadku właściwości prostych (używając typu proceduralnego/funkcyjnego w odpowiednich miejscach) z tym, że często wykorzystuje się najprostszy sposób definiowania – poprzez bezpośrednie odwołanie do pola. Na wspomnienie zasługuje jeszcze sposób wykorzystania właściwości proceduralnych/funkcyjnych w komponencie, czyli wywoływanie zdarzeń. W wybranym miejscu metody komponentu wywołuje się po prostu nazwę pola stowarzyszonego z właściwością tak jak odpowiednią funkcję lub procedurę – podając w nawiasach parametry.

Uwaga. Nie ma gwarancji, że dane zdarzenie zostało przez użytkownika komponentu „podpięte” do jakiejś istniejącej procedury lub funkcji (to znaczy że nie wiadomo, czy zmienna proceduralna/funkcyjna wskazuje na miejsce w pamięci, gdzie znajduje się właściwy kod do wykonania). Zawsze należy zabezpieczyć się przed ewentualnymi błędami sprawdzając przed wywołaniem zdarzenia, czy jest pod nie „podpięta” procedura obsługi. Służy do tego funkcja ***assigned***, zwracająca *true*, gdy pod sprawdzane zdarzenie podpięty jest kod.

Bezpieczny kod wywołania zdarzenia może wyglądać następująco:

```
If assigned(FMojeZdarzenie) then FMojeZdarzenie(Self);   // zakładamy tu, że FMojeZdarzenie jest typu TNotifyEvent
```

Właściwości obiektowe

Jeśli tworzony jest rozbudowany obiekt łączący w sobie funkcjonalności kilku zasadniczo różnych obiektów, to aby uniknąć nadmiaru pracy możliwości tych obiektów można połączyć wybierając jeden z nich jako nadrzędny i utworzyć nową klasę dziedzicząc właśnie z niego, zaś możliwości innych obiektów uzyskać tworząc je jako pola obiektu nadrzędnego. Jeśli istnieje potrzeba zmiany właściwości tych obiektów w czasie projektowania, to obiekty te muszą być właściwościami komponentu nadrzędnego, który je utworzył. W ten sposób pojawia się ostatni typ właściwości – właściwości obiektowe.