Excel. Programowanie dla profesjonalistów

Autorzy: Stephen Bullen, Rob Bovey, John Green
Tłumaczenie: Robert Górczyński, Krzysztof Masłowski
ISBN: 83-246-0079-5
Tytuł oryginału: Professional Excel Development: The Definitive Guide to Developing Applications Using Microsoft(R) Excel and VBA(R)
Format: B5, stron: 768

Większości użytkowników Excel kojarzy się wyłącznie z arkuszem kalkulacyjnym używanym w biurach. Zdecydowanie mniej osób wie, że Excel jest również potężnym narzędziem programistycznym, za pomocą którego projektuje się rozbudowane aplikacje, wyposażone w graficzne interfejsy użytkownika i korzystające z danych zewnętrznych, języka XML i usług sieciowych. Dzięki językowi Visual Basic for Applications (VBA) można tworzyć na bazie Excela profesjonalne narzędzia bardzo dobrej jakości.

Książka „Excel. Programowanie dla profesjonalistów" to podręcznik poświęcony projektowaniu aplikacji w środowisku Excela, adresowany do doświadczonych użytkowników tego narzędzia oraz programistów. Autorzy krok po kroku wyjaśniają, jak tworzyć dodatki, implementować usługi sieciowe, projektować formularze userform. Uczą konstruowania wykresów i obsługi błędów, pokazują, w jaki sposób zoptymalizować wydajność aplikacji i jak je dystrybuować.

- Najlepsze praktyki programistyczne w Excelu i VBA
- Projektowanie arkusza
- Tworzenie dodatków
- Sterowanie paskami narzędzi
- Korzystanie z funkcji Windows API
- Budowanie interfejsów użytkownika
- Połączenia z bazami danych
- Usuwanie błędów z kodu źródłowego
- Sterowanie innymi aplikacjami MS Office
- Korzystanie z funkcji Visual Basic 6 i VB.NET
- Połączenia z usługami sieciowymi
- Tworzenie wersji dystrybucyjnej aplikacji

**Poznaj zasady tworzenia aplikacji przy użyciu Excela –
z tych narzędzi korzystają największe firmy świata**

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Najlepsze praktyki programowania w Excelu i VBA

Ten rozdział umieściliśmy niedaleko początku książki, aby zawczasu wyjaśnić, dlaczego w dalszych rozdziałach pewne rzeczy będziemy robili tak, a nie inaczej. Niestety, oznacza to również konieczność poruszenia tutaj pewnych tematów, które w pełni zostaną wyjaśnione dopiero później. Dla osiągnięcia najlepszych rezultatów warto przejrzeć ten rozdział ponownie po przeczytaniu reszty książki.

Czytając ten rozdział, nie zapominaj, że choć opisane w nim praktyki są ogólnie uznane za najlepsze, zawsze będą istniały przypadki, w których najlepszym rozwiązaniem będzie postępowanie niezgodne z najlepszymi praktykami. Najczęstsze przykłady takich sytuacji staraliśmy się wskazać w tym rozdziale i dyskusjach dotyczących najlepszych praktyk omawianych w następnych rozdziałach.

Konwencje nazw

Czym jest konwencja nazw i dlaczego jest istotna?

Termin ***konwencja nazw*** określa system, jakiego używasz, nazywając różne części tworzonej aplikacji. Zawsze gdy deklarujesz zmienną lub tworzysz formularz, nadajesz im nazwy. Niejawnie nazywasz obiekty nawet wtedy, gdy nie nadajesz im nazw wprost, akceptując jedynie nazwy domyślne, czego przykładem może być tworzenie formularza. Jednym ze znaków firmowych dobrej praktyki programowania jest stosowanie dla wszystkich części aplikacji VBA nazw spójnych i zgodnych z jasno określoną konwencją.

Przyjrzyjmy się przykładowi, który pozwoli pokazać, czym jest konwencja nazw. Co możesz wywnioskować o `x` z podanej niżej linii kodu?

```
x = wksDataSheet.Range("A1").Value
```

Ze sposobu użycia możesz rozsądnie wnioskować, że jest zmienną. Ale jaki typ wartości ma przechowywać? Czy jest to zmienna publiczna, czy o zasięgu na poziomie modułu, czy może zmienna prywatna? Do czego służy w programie? W podanej sytuacji nie jesteś w stanie odpowiedzieć na żadne z tych pytań, bez poświęcenia nieco czasu na przeszukanie kodu. Dobra konwencja nazw pozwala na odczytanie takich informacji z samej nazwy zmiennej. Oto przykład poprawiony (szczegóły zostaną omówione w następnym podrozdziale).

```
glListCount = wksDataSheet.Range("A1").Value
```

Teraz już znasz zakres zmiennej (`g` oznacza `global` — zmienną globalną, czyli inaczej publiczną), wiesz, jaki typ danych ma przechowywać (`l` oznacza `Long`) i masz ogólne pojęcie, do czego ta zmienna ma służyć (przechowuje liczbę elementów listy).

Konwencja nazw pomaga w szybkim rozpoznawaniu typu i celu użycia bloków, z których jest budowana aplikacja. To pozwala Ci koncentrować się raczej na zadaniach kodu niż na jego strukturze. Konwencje nazw służą także samodokumentacji kodu, zmniejszając liczbę komentarzy koniecznych do tego, aby był zrozumiały.

W następnym podrozdziale pokazujemy przykład konwencji nazw z przemyślaną strukturą. Najważniejszą sprawą jest jednak wybranie jednej konwencji i jej konsekwentne przestrzeganie. Jeżeli wszyscy uczestniczący w projekcie rozumieją przyjętą konwencję, nie ma właściwie znaczenia, jakie prefiksy są używane lub jak i kiedy są stosowane litery wielkie i małe. Zasady przyjętej konwencji powinny być spójne i konsekwentnie, bez zmian stosowane nie tylko w pojedynczym projekcie, lecz również w dłuższym okresie czasu.

Przykładowa konwencja nazw

Dobra konwencja nazw obejmuje nie tylko zmienne, lecz wszystkie elementy aplikacji. Przykładowa, pokazana przez nas konwencja obejmuje wszystkie elementy typowej aplikacji Excela. Rozpoczniemy od dyskusji o zmiennych, stałych i związanych z nimi elementach, gdyż to właśnie one najczęściej występują we wszelkich aplikacjach. W tabeli 3.1 pokazujemy ogólny format konwencji nazw. Określone elementy i ich cele są opisane dalej.

Tabela 3.1. *Konwencja nazw zmiennych, stałych, typów definiowanych przez użytkownika i wyliczeń*

Element	Konwencja nazw
Zmienne	*<zakres><tablica><typ danych>*NazwaOpisowa
Stałe	*<zakres><typ danych>*NAZWA_OPISOWA
Typy definiowane przez użytkownika	Type NAZWA_OPISOWA *<typ danych>*NazwaOpisowa End Type
Typ wyliczeniowy	Enum *<prefiks projektu>*OpisOgolny *<prefiks projektu>*OpisOgolnyNazwa1 *<prefiks projektu>*OpisOgolnyNazwa2 End Num

Określnik zakresu (<zakres>)

g — `Public` (publiczny)

m — `Module` (na poziomie modułu)

(nic) — na poziomie procedury

Określnik tablicy (<tablica>)

a — `Array` (tablica)

(nic) — nietablica

Określnik typu danych (<typ danych>)

Istnieje tak wiele typów danych, że trudno sporządzić zrozumiałą listę odpowiadających im prefiksów. Typy wbudowane są proste. Problem powstaje przy nazywaniu zmiennych obiektowych odnoszących się do obiektów z różnych aplikacji. Niektórzy programiści stosują prefiks `obj` dla wszystkich nazw obiektowych. Nie można na to przystać. Jednakże obmyślenie zbioru spójnych, niepowtarzających się, racjonalnie uzasadnionych i krótkich prefiksów, określających każdy typ obiektu, jakiego kiedykolwiek będziesz używać, okazuje się być zadaniem nad siły. Staraj się tworzyć jedno-, dwu- i trzyliterowe rozsądnie uzasadnione prefiksy dla najczęściej używanych zmiennych obiektowych, a prefiks `obj` zarezerwuj dla obiektów rzadko pojawiających się w kodzie.

Pisz kod przejrzysty, a przede wszystkim spójny. Twórz prefiksy nie dłuższe niż trzyliterowe. Stosowanie dłuższych w kombinacji z określnikami zakresu i tablicy prowadzi do tworzenia nazw nieporęcznie długich. W tabeli 3.2 zawarto listę sugerowanych prefiksów dla najczęściej używanych typów danych.

Stosowanie nazw opisowych

Choć VBA pozwala na stosowanie nazw zmiennych o długości do 255 znaków, wykorzystuj jedynie niewielką część dozwolonej długości, ale nie leń się i nie skracaj nazw zmiennych zaledwie do kilku znaków. Robiąc to, sprawisz, że Twój kod stanie się trudny do zrozumienia zwłaszcza po upływie czasu i w przyszłości sprawi kłopot sobie lub innej osobie, która będzie nad nim pracować.

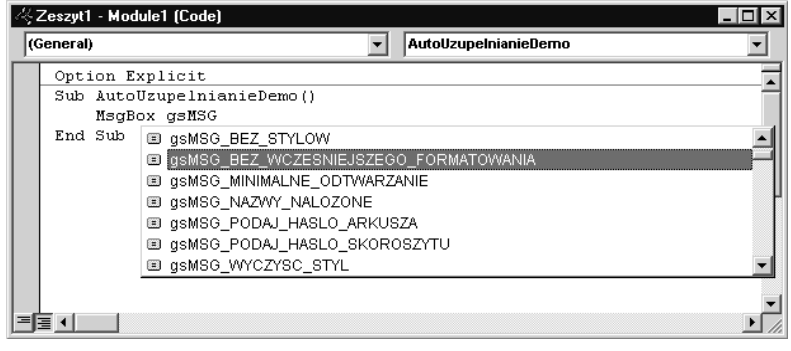
Zintegrowane środowisko programistyczne Visual Basica (Visual Basic IDE — *Integrated Development Environment*) posiada cechę autouzupełniania identyfikatorów (wszystkich nazw używanych w aplikacji). Zwykle aby wstawić nazwę, musisz napisać jedynie kilka pierwszych znaków. Gdy po napisaniu kilku znaków naciśniesz *Ctrl+spacja*, ukaże się lista autouzupełniania, zawierająca wszystkie nazwy rozpoczynające się od podanych znaków. W miarę wpisywania kolejnych znaków lista będzie się skracać. Na rysunku 3.1 kombinacja *Ctrl+spacja* została użyta do wyświetlenia listy stałych łańcuchów komunikatów, jakie można dodać do pola komunikatu.

Tabela 3.2. *Prefiksy proponowane do użycia w konwencji nazw*

Prefiks	Typ danych	Prefiks	Typ danych	Prefiks	Typ danych
b	Boolean	cm	ADODB.Command	cbo	MSForms.ComboBox*
byt	Byte	cn	ADODB.Connection	chk	MSForms.CheckBox
cur	Currency	rs	ADODB.Recordset	cmd	MSForms.CommandButton
dte	Date			ddn	MSForms.ComboBox**
dec	Decimal	cht	Excel.Chart	fra	MSForms.Frame
d	Double	rng	Excel.Range	lbl	MSForms.Label
i	Integer	wkb	Excel.Workbook	lst	MSForms.ListBox
l	Long	wks	Excel.Worksheet	mpg	MSForms.MultiPage
obj	Object			opt	MSForms.OptionButton
sng	Single	cbr	Office.CommandBar	spn	MSForms.SpinButton
s	String	ctl	Office.CommandBar-Control	txt	MSForms.TextBox
u	User-defined type - Typ zdefiniowany				
v	Variant	cls	User-defined class variable - odwołanie do klasy	ref	Kontrolka RefEdit
		frm	UserForm	col	kolekcja, zbiór

** Stosowane do kontrolek `ComboBox` typu `DropDownCombo`.*

*** Stosowane do kontrolek `ComboBox` typu `DropDownList`.*



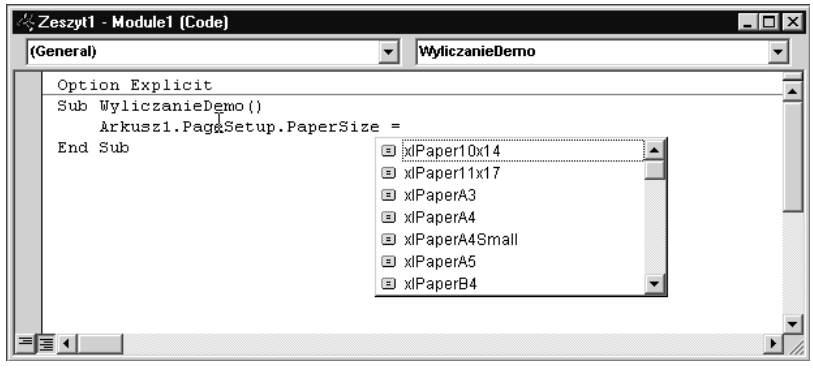
Rysunek 3.1. *Skrót klawiaturowy Ctrl+spacja pozwala na autouzupełnianie długich nazw*

Kilka słów o stałych typu wyliczeniowego

W Excelu 2000 i wersjach nowszych są dostępne stałe tzw. typu wyliczeniowego. Pozwalają one na tworzenie listy spokrewnionych wartości i udostępnienie ich przez logicznie uzasadnione, przyjazne dla użytkownika nazwy. VBA i model obiektowy Excela szeroko korzystają z takich wyliczeń. Możesz to sprawdzić, korzystając z autouzupełniania, na jakie pozwala VBA przy podawaniu wartości właściwości. Jeżeli np. w module VBA napiszesz:

```
Arkusz1.PageSetup.PaperSize =
```

zobaczysz długą listę składowych typu wyliczeniowego podającą różne wymiary papierów do drukowania (rysunek 3.2).



Rysunek 3.2. *Lista wyliczająca dostępne rozmiary papieru*

Te nazwy w rzeczywistości reprezentują stałe liczbowe, których wartości możesz sprawdzić za pomocą przeglądarki obiektów (*Object Browser*), omówionej w rozdziale 16. Zwróć uwagę na strukturę nazw stałych typu wyliczeniowego. Po pierwsze, wszystkie rozpoczynają się od przedrostka identyfikującego aplikację — w tym przypadku `xl` oznacza Excela. Po drugie, początkowa część nazwy jest opisowa, co wizualnie wiąże ze sobą nazwy należące do tego samego typu wyliczeniowego — w tym przypadku jest to `Paper`. Ostatnią częścią nazwy wyliczeniowej jest unikatowy łańcuch, opisujący specyficzną wartość. Przykładowo `xlPaper11x17` opisuje papier o formacie 11x7, a `xlPaperA4`, odpowiednio papier o formacie A4. Taka konwencja nazw wyliczeniowych jest bardzo rozpowszechniona i zastosowano ją również w tej książce.

Przykłady stosowania konwencji nazw

Abstrakcyjne wyjaśnienie związku deskryptorów konwencji nazw z nazwami rzeczywistymi jest trudne, więc w tym podrozdziale pokażemy kilka praktycznych przykładów. Wszystkie pochodzą wprost z komercyjnych aplikacji napisanych przez autorów.

Zmienne

- `gsErrMsg` — zmienna publiczna (*public variable*) typu `String` używana do przechowywania komunikatu o błędzie[1].
- `mauSettings()` — tablica na poziomie modułu typu zadeklarowanego przez użytkownika, używana do przechowywania parametrów (*settings*).

[1] Jeżeli aplikacja ma być używana jedynie w Polsce, warto stosować opisowe nazwy polskie. Jeżeli jednak nazwy odnoszą się do typów, narzędzi itp. środowiska VBA (które pozostaje angielskie), trzeba wystrzegać się używania całkowicie niestrawnej mieszanki językowej, która może okazać się zrozumiała jedynie dla autora. Warto wtedy stosować nazwy angielskie. Decyzja dotycząca części opisowej bywa niekiedy trudna. Prościej jest z przedrostkami określającymi zakres i typ zmiennych bądź stałych, które warto uzależniać od angielskich nazw obiektów i typów (np. zmienne musimy deklarować jako `Long` i `String`, niezależnie od naszych upodobań językowych). To pomieszanie języków utrudnia opracowanie logicznej i przejrzystej konwencji nazw — *przyp. tłum*.

- `cbrMenu` — lokalna zmienna typu `CommandBar`, przechowująca odwołanie do paska menu.

Stałe

- `gbDEBUG_MODE` — stała publiczna (*public constant*) typu boole'owskiego wskazująca, czy projekt działa w trybie debugowania.
- `msCAPTION_FILE_OPEN` — stała na poziomie modułu z wartością typu `String`, przechowująca tytuł (*caption*) zdefiniowanego przez użytkownika okna otwierania plików (w tym przykładzie `Application.GetOpenFilename`).
- `lOFFSET_START` — stała lokalna z daną typu `Long`, przechowująca punkt, od którego obliczamy przesunięcie względem jakiegoś obiektu typu `Range`.

Typy definiowane przez użytkownika

Niżej został podany przykład typu danych zdefiniowanych przez użytkownika, mających służyć do przechowywania wymiarów i położenia obiektu. Nowy typ danych składa się z czterech zmiennych typu `Double`, przechowujących położenie góry i lewej strony obiektu, jego szerokość i wysokość oraz zmienną typu `Boolean`, służącą do wskazywania, czy dane zostały zapisane.

```
Public Type WYMIARY_USTAWIENIA
    bUstawieniaZapisane As Boolean
    dWartGora As Double
    dWartLewa As Double
    dWartWysokosc As Double
    dWartSzerokosc As Double
End Type
```

Zmienne wewnątrz definicji typu użytkownika nazywamy ***zmiennymi składowymi*** (*member variables*). Można je deklarować w dowolnej kolejności, jednakże w naszej konwencji przyjmujemy kolejność alfabetyczną według typów danych, jeżeli tylko nie występują ważne powody grupowania zmiennych w inny sposób.

Typ wyliczeniowy

Poniżej został zdefiniowany typ wyliczeniowy na poziomie modułu stosowany do opisu różnego rodzaju dni. Prefiks `sch` określa nazwę aplikacji. W tym przypadku podane wyliczenie pochodzi z aplikacji Scheduler. `TypDnia` jest częścią nazwy wskazującą cel tego typu wyliczeniowego, zaś indywidualne przyrostki pokazują indywidualne znaczenie każdej składowej typu wyliczeniowego.

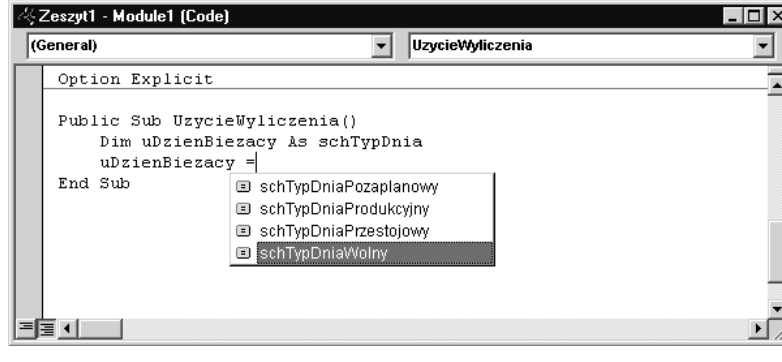
```
Private Enum schTypDnia
    schTypDniaPozaplanowy
    schTypDniaProdukcyjny
    schTypDniaPrzestojowy
    schTypDniaWolny
End Enum
```

Jeżeli nie określisz wartości składowych typu wyliczeniowego, VBA automatycznie przypisuje pierwszej składowej wartość zero, a kolejnym składowym wartości zwiększane o jeden. Możesz to łatwo zmienić, przypisując inną wartość startową, od której VBA rozpocznie inkrementację. Aby nakazać VBA zwiększanie wartości od 1 zamiast od zera, powinieneś napisać:

```
Private Enum schTypDnia
    schTypDniaPozaplanowy = 1
    schTypDniaProdukcyjny
    schTypDniaPrzestojowy
    schTypDniaWolny
End Enum
```

VBA rozpoczyna inkrementację o jeden, zaczynając od ostatniej określonej przez Ciebie wartości. Możesz uniemożliwić automatyczne przypisywanie wartości przez proste ich określenie dla wszystkich składowych.

Na rysunku 3.3. pokazujemy jedną z korzyści, jakie daje stosowanie typu wyliczeniowego. VBA dostarcza listę potencjalnych wartości dla każdej zmiennej zadeklarowanej jako należąca do danego typu wyliczeniowego.



Rysunek 3.3. *Nawet deklarowany przez użytkownika typ wyliczeniowy jest obsługiwany przez autouzupełnianie VBA*

Procedury

Znamy dwa typy procedur — procedury typu `Sub` (podprogramy) i funkcje. Zawsze nadawaj procedurom nazwy opisowe. Powtarzamy ponownie, że nazwy procedur mogą mieć 255 znaków i również pojawiają się na listach autouzupełnień wyświetlanych za pomocą skrótu klawiszowego *Ctrl+spacja*, zatem nie ma powodów, aby poświęcać dłuższe nazwy opisujące cel procedury na rzecz innych, których jedyną zaletą jest to, że są krótkie.

Choć nie jest to powszechną praktyką, uważamy, że poprzedzanie nazwy funkcji prefiksem określającym typ zwracanej danej jest przydatne i ułatwia rozumienie kodu. Wywołując funkcję, zawsze po nazwie dajemy parę okrągłych nawiasów dla odróżnienia od nazwy zmiennej lub procedury `Sub` i robimy to nawet wtedy, gdy funkcja nie ma argumentów. W listingu 3.1 pokazujemy dobrze nazwaną funkcję boole'owską, użytą jako test w instrukcji `If...Then`.

Listing 3.1. *Przykład funkcji nazwanej zgodnie z konwencją nazw*

```
If bWeryfikacjaSciezki("C:\Pliki") Then
    'Jeżeli podana ścieżka istnieje
    'blok instrukcji If...Then jest wykonywany
End Sub
```

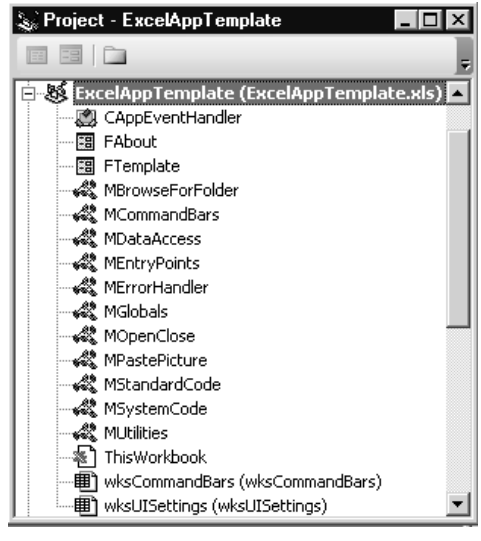
Podprogramy (procedury `Sub`) powinny otrzymywać nazwy opisujące zadania, jakie wykonują. Przykładowo nazwa procedury `ZamykanieAplikacji` nie pozostawia wiele wątpliwości dotyczących wykonywanego zadania. Nazwy funkcji powinny opisywać zwracane wartości. Możemy oczekiwać, że funkcja `sPodajNieuzywanaNazwePliku()` poda nazwę pliku.

Konwencja nazw stosowana do argumentów procedur jest taka sama, jak dla zmiennych na poziomie procedury, np. funkcja `bWeryfikacjaSciezki` pokazana w listingu 3.1 może być zadeklarowana w następujący sposób:

```
Function bWeryfikacjaSciezki(ByVal sSciezka As String) As Boolean
```

Moduły, klasy i formularze UserForm

W naszej przykładowej konwencji nazw nazwy standardowych modułów kodu powinny być poprzedzane przedrostkiem M, moduły klas przedrostkiem C, zaś formularze UserForm przedrostkiem F. Daje to — po zrezygnowaniu z wyświetlania folderów — dodatkowy, przyjemny efekt sortowania nazw w oknie *Project* edytora VBA, co zostało pokazane na rysunku 3.4.



Rysunek 3.4. *Moduły klas, formularze UserForm i moduły standardowe posortowane w oknie Project edytora VBA*

Ta konwencja ułatwia rozumienie kodu używającego modułów klas i formularzy UserForm. Podany niżej przykład pokazuje, jak konwencja ułatwia rozpoznanie, że deklarowana jest zmienna obiektowa klasy określonego typu, a potem tworzony nowy egzemplarz tej klasy.

```
Dim clsMojaKlasa As CMojaKlasa
Set clsMojaKlasa = New CMojaKlasa
```

Zawsze nazwa po lewej jest ***zmienną*** typu danej klasy, zaś obiekt po prawej jest ***klasą***.

Arkusze zwykłe i arkusze wykresów

Ponieważ stosowane w kodzie nazwy (`CodeNames`) arkuszy zwykłych i arkuszy wykresów użytych w aplikacji są przez VBA traktowane jako wewnętrzne zmienne obiektowe, powinny być nadawane zgodnie z przyjętą konwencją nazywania zmiennych. Nazwy arkuszy są poprzedzane przedrostkiem `wks` (*worksheet*), który w kodzie będzie identyfikował ich przynależność do obiektów arkuszy. Odpowiednio nazwy arkuszy wykresów (*chart sheets*) są poprzedzane przedrostkiem `cht`, identyfikującym je jako przynależne do obiektów typu wykres (*chart*).

W obu typach arkuszy po prefiksie powinna być umieszczona nazwa opisowa, określająca cel arkusza w aplikacji. Przykładowo na rysunku 3.4 widać nazwę arkusza `wksCommandBars`, zawierającego tablicę definiującą paski narzędziowe tworzone przez aplikację. W przypadku arkuszy zawartych w dodatkach oraz arkuszy ukrytych nazwy podane na zakładkach arkuszy powinny być identyczne z nazwami kodowymi. Nazwy „zakładkowe" arkuszy widocznych dla użytkownika powinny być dla niego przyjazne i zrozumiałe, a poza tym musisz być przygotowany, że mogą być przez użytkownika zmienione. Później dokładnie wyjaśnimy, dlaczego wewnątrz kodu powinieneś zawsze polegać na kodowych nazwach arkuszy, a nie na ich nazwach „zakładkowych".

Projekt VBA (VBA Project)

Zauważ na rysunku 3.4, że projekt VBA otrzymał tę samą nazwę co skoroszyt z nim powiązany. Zawsze powinieneś projektowi VBA nadawać nazwę pozwalającą na jasne zidentyfikowanie aplikacji, do której należy. Nie ma nic gorszego jak grupa otwartych skoroszytów, których wszystkie projekty mają w VBA domyślną nazwę `VBAProject`. Jeżeli będziesz chciał tworzyć odwołania między projektami, będziesz musiał nadać im unikatowe nazwy.

Konwencje nazw interfejsu użytkownika Excela

Elementy interfejsu użytkownika Excela używane podczas tworzenia aplikacji powinny również otrzymywać nazwy zgodne z dobrze opracowaną i spójną wewnętrznie konwencją. W poprzednim podrozdziale omówiliśmy arkusze zwykłe i arkusze wykresów. Trzy następne główne elementy interfejsu użytkownika Excela, jakie musimy omówić, to kształty (obiekty *shape*), obiekty osadzone (*embedded objects*) i nazwy definiowane.

Kształty — obiekty Shape

Termin ***kształty (Shapes)*** określa zbiór, który może zawierać wiele różnorodnych obiektów, jakie możesz umieszczać nad arkuszem lub arkuszem wykresu. Kształty możemy podzielić ogólnie na trzy główne kategorie: kontrolki, obiekty rysowane i obiekty

osadzone. Kształty należy nazywać tak jak zmienne obiektowe, czyli zaczynać nazwę od prefiksu definiującego typ obiektu, w dalszej części podając nazwę opisującą cel, jakiemu obiekt służy w aplikacji.

Wiele kontrolek umieszczanych w formularzu `UserForm` można również sadowić w arkuszach. W arkuszu mogą się także znaleźć stare kontrolki z paska narzędzi *Formularze*, które choć podobne do kontrolek ActiveX MSForms, mają swe własne plusy i minusy, co bardziej szczegółowo zostało omówione w rozdziale 4. Kontrolki umieszczane w arkuszach powinny być nazywane zgodnie z konwencją stosowaną przy nazywaniu kontrolek umieszczanych w formularzach.

Do arkuszy można także wprowadzać wiele obiektów rysowanych (znanych pod techniczną nazwą kształtów), które — ściśle mówiąc — nie są kontrolkami, choć można do nich przypisywać makra. Należą do tej samej kategorii konwencji nazw, co wiele obiektów używanych w VBA. Byłoby bardzo trudno określić dla nich jednoznaczne przedrostki nazw, więc używaj dobrze określonych przedrostków dla obiektów najpowszechniej używanych, a dla reszty stosuj przedrostki standardowe. Oto przykładowe przedrostki dla trzech powszechnie używanych obiektów rysunkowych:

`pic`	`Picture`	Obraz.
`rec`	`Rectangle`	Prostokąt.
`txt`	`TextBox` (ale nie kontrolka ActiveX)	Pole tekstowe.

Obiekty osadzone (Embedded Objects)

Termin ***obiekty osadzone*** jest tu używany w odniesieniu do obiektów Excela, takich jak tabele przestawne (`PivotTables`), tabele zapytań (`QueryTables`) i wykresy (`ChartObjects`), jak również obiekty kreowane przez aplikacje różne od Excela. Arkusze mogą przechowywać wiele różnych osadzonych obiektów. Znanym przykładem obiektów osadzonych w arkuszu, które nie powstały w Excelu, są równania tworzone przez edytor równań (`Equation` *Editor*) oraz rysunki, które powstają za pomocą `WordArt`. Oto przykładowe przedrostki dla nazw obiektów osadzonych:

`cht`	`ChartObject`	Wykres.
`eqn`	`Equation`	Równanie.
`qry`	`QueryTable`	Tabela zapytania.
`pvt`	`PivotTable`	Tabela przestawna.
`art`	`WordArt`	WordArt.

Nazwy definiowane

W naszej konwencji nazwy definiowane są traktowane nieco inaczej niż inne elementy. W przypadku nazw definiowanych przedrostek powinien szeroko określać cel definiowanej nazwy, a nie typ danych, jakie mają być przechowywane. W aplikacjach Excela, poza najprostszymi, występuje wiele nazw definiowanych i ich grupowanie

według celu w oknie dialogowym *Definiowanie nazw* (*Definiuj nazwy* — w wersjach starszych niż Excel 2003*)* znacznie ułatwia pracę. Jeżeli arkusz zawiera dziesiątki lub setki nazw definiowanych, takie pogrupowanie funkcjonalne przez zastosowanie odpowiednich przedrostków przynosi widoczne korzyści.

Opisowa część nazwy zdefiniowanej dokładne określa, do czego ta nazwa służy w ramach szerszej kategorii. Na podanej niżej liście widzimy kilka przedrostków celu nazw definiowanych.

`cht`	*Chart data range*	Zakres danych wykresu.
`con`	*Named constant*	Nazwana stała.
`err`	*Error check*	Znacznik błędu.
`for`	*Named formula*	Nazwana formuła.
`inp`	*Input range*	Zakres wejściowy.
`out`	*Output range*	Zakres wyjściowy.
`ptr`	*Specific cell location*	Określony adres komórki.
`rgn`	*Region*	Obszar (zakres).
`set`	*UI setting (User Interface)*	Ustawienie interfejsu użytkownika.
`tbl`	*Table*	Tabela.

Wyjątki — kiedy nie stosuje się konwencji nazw

W dwóch przypadkach zechcesz złamać ogólną konwencję nazw. Po pierwsze, gdy masz do czynienia z elementami dotyczącymi wywołań Windows API. Elementy te zostały nazwane przez Microsoft i są szeroko znane w społeczności programistów. Stałe Windows API, typy deklarowane przez użytkownika, deklaracje i argumenty procedur powinny pojawiać się w kodzie dokładnie w takiej postaci, w jakiej znajdujemy je w SDK[2] dla platformy Microsoftu (*Microsoft Platform SDK*), co możemy sprawdzić w serwisie internetowym MSDN pod adresem:

```
http://msdn.microsoft.com/library/
en-us/winprog/winprog/windows_api_start.page.asp
```

Zauważ, że są to odwołania w formacie C/C++.

Drugim przypadkiem, gdy nie zechcesz użyć własnej konwencji nazw, jest stosowanie kodu pochodzącego ze źródeł zewnętrznych, służącego do realizacji specjalnych zadań. Jeżeli zmodyfikujesz nazwy w tej części kodu i będziesz się do nich odwoływać w reszcie aplikacji, ewentualna aktualizacja wstawki, jeżeli pojawi się jej nowa wersja, będzie bardzo trudna.

[2] SDK (*Software Development Kit*) — zestaw narzędzi programistycznych przydatny przy tworzeniu własnych aplikacji dla określonych platform sprzętowych i systemowych — *przyp. tłum.*

Najlepsze praktyki organizacji i tworzenia struktury aplikacji

Struktura aplikacji

Aplikacja jednoskoroszytowa, a aplikacja n-skoroszytowa

Liczba skoroszytów użytych w aplikacji Excela zależy przede wszystkim od dwóch czynników: złożoności samej aplikacji i ograniczeń nałożonych przez warunki dystrybucji i aktualizacji wersji. Aplikacje proste i takie, dla których nie można wymusić określonej kolejności działań instalacyjnych, wymagają możliwie najmniejszej liczby skoroszytów. Aplikacje złożone i takie, których procesem instalacyjnym można w pełni sterować, mogą być dzielone na wiele skoroszytów lub plików innego typu, np. DLL. W rozdziale 2. zostały omówione różne typy aplikacji Excela i odpowiednie dla nich struktury.

Jeżeli możesz swobodnie dzielić aplikację na pliki według własnego uznania, wiele przemawia za tym, aby to czynić. Przyczyny, jakie warto brać pod uwagę, to: podział aplikacji na warstwy logiczne, oddzielenie kodu od danych, oddzielenie elementów interfejsu użytkownika od kodu, hermetyzacja funkcjonalnych elementów aplikacji i nadzór nad konfliktami wynikającymi ze zmian przy pracy zespołowej.

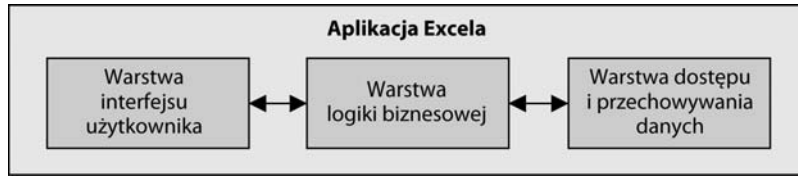
Rozdzielenie warstw logicznych

Niemal każda nietrywialna aplikacja Excela ma trzy wyraźne logiczne warstwy czy sekcje. Są to:

- **Warstwa interfejsu użytkownika**. Warstwa ta składa się z kodu i widocznych elementów potrzebnych aplikacji do interakcji z użytkownikiem. W aplikacji Excela warstwa interfejsu ma część widoczną, do której należą elementy, takie jak arkusze, wykresy, paski narzędziowe, formularze użytkownika oraz niewidoczną, którą stanowi kod potrzebny do sterowania elementami widocznymi. Warstwa interfejsu użytkownika jest jedyną warstwą logiczną zawierającą elementy widoczne dla użytkownika.
- **Warstwa logiki biznesowej (warstwa aplikacji)**. Warstwa logiki biznesowej to w całości kod, który wykonuje zadania zasadnicze, dla których cała aplikacja została zaprojektowana i napisana. Warstwa logiki biznesowej akceptuje dane wejściowe, pochodzące z warstwy interfejsu użytkownika i tamże zwraca wyniki. W przypadku operacji długo trwających warstwa logiki biznesowej przekazuje do warstwy interfejsu użytkownika okresowe informacje w postaci komunikatów statusu lub paska wskazującego stopień wykonania zadania.
- **Warstwa dostępu i przechowywania danych**. Warstwa dostępu i przechowywania danych jest odpowiedzialna za przechowywanie i dostarczanie na żądanie danych potrzebnych aplikacji. Może to być tak proste, jak odczytywanie

danych z komórek i ich zapisywanie w komórkach lokalnego lub ukrytego arkusza czy tak skomplikowane, jak egzekucja przez sieć procedur na serwerze bazodanowym (SQL Server). Warstwa dostępu i przechowywania danych komunikuje się bezpośrednio jedynie z warstwą logiki biznesowej.

Na rysunku 3.5 możesz zobaczyć, że wszystkie trzy warstwy są konieczne do stworzenia kompletnej aplikacji, ale nie muszą być ze sobą nierozerwalnie powiązane. Trzy warstwy są luźno łączone i istotna zmiana w jednej nie musi wymagać istotnej zmiany w innej. Silne wiązanie warstw aplikacji w sposób nieunikniony komplikuje jej konserwację i uaktualnianie.



Rysunek 3.5. *Relacje między trzema warstwami logicznymi aplikacji Excela*

Jeżeli np. warstwa dostępu i przechowywania danych wymaga zmiany dotychczasowej bazy accessowej na bazę danych na SQL Servera, wszelkie zmiany dotyczą jedynie tej warstwy. W dobrze zaprojektowanej aplikacji żadna z dwóch pozostałych warstw nie powinna być tknięta podczas wprowadzania potrzebnych zmian. W sytuacji idealnej dane między warstwą dostępu i przechowywania danych a warstwą logiki biznesowej powinny być przekazywane w postaci zmiennych o typie zdefiniowanym przez użytkownika. To pozwala na najlepsze zrównoważenie efektywności i swobody powiązania. Alternatywnie można stosować obiekty ADO Recordset, ale to wprowadza subtelne problemy powiązań, takie jak kolejność pól zwracanych z bazy danych, od czego lepiej nie uzależniać warstwy logiki biznesowej.

I podobnie, jeżeli aplikacja ma zawierać alternatywny internetowy interfejs prezentacyjny, swobodne powiązanie warstwy interfejsu użytkownika z warstwą logiki biznesowej ułatwi Ci wykonanie tego zadania. Będzie ono łatwiejsze z powodu braku niejawnych założeń wbudowanych w warstwę logiki biznesowej oraz założeń dotyczących konstrukcji interfejsu użytkownika. Elementy akceptujące dane wprowadzane przez użytkownika powinny być w pełni niezależne i samowystarczalne. Warstwa logiki biznesowej musi przekazywać interfejsowi użytkownika potrzebne dane inicjalizacyjne jako dane o prostych właściwościach. Interfejs użytkownika powinien zbierać dane wprowadzane przez użytkownika i przekazywać je wstecz do warstwy logiki biznesowej również jako dane o prostych właściwościach lub, w przypadkach bardziej złożonych, jako dane o typie zdefiniowanym przez użytkownika (UDT User [defined] Data Type). Ponieważ warstwa logiki biznesowej nie powinna zawierać żadnych wewnętrznych informacji o konstrukcji interfejsu użytkownika, bezpośrednie odwołania z warstwy logiki biznesowej do kontrolek formularza użytkownika są zakazane.

Oddzielenie danych i interfejsu użytkownika od kodu

W wielu aplikacjach użytkownika wewnątrz warstwy interfejsu użytkownika bywają stosowane dwie podwarstwy. Stanowią je skoroszyt i elementy arkusza używane do budowania interfejsu oraz kod obsługujący te elementy. Podział na podwarstwy musi

być tu rygorystycznie przestrzegany. Interfejs użytkownika korzystający z arkusza nie powinien zawierać żadnego kodu, zaś kod kontrolujący ten interfejs należy umieścić w całkowicie oddzielonym dodatku.

Powód tej separacji jest dokładnie taki sam, jak opisany poprzednio dla separacji głównych warstw logicznych aplikacji — jest to izolacja efektów wprowadzanych zmian. Ze wszystkich warstw aplikacji najczęściej zmieniana jest warstwa interfejsu użytkownika. Dlatego nie jest wystarczające oddzielenie jej w całości i należy odseparować także zmiany widocznych elementów interfejsu użytkownika od kodu te elementy kontrolującego.

W następnych rozdziałach podamy wzięte z życia przykłady oddzielenia warstw aplikacji, więc nie przejmuj się, jeżeli teraz jakiś element dyskusji nie jest dla Ciebie w pełni zrozumiały.

Organizacja aplikacji w programowaniu proceduralnym

Programowaniem proceduralnym nazywamy metodologię programowania znaną większości twórców oprogramowania. Polega ona na dzieleniu aplikacji na wiele procedur, z których każda wykonuje oddzielne zadanie. Cała aplikacja może zostać napisana w ten sposób, a elementy proceduralne mogą być kombinowane z elementami zorientowanymi obiektowo lub cała aplikacja może zostać napisana metodą zorientowaną obiektowo. W tym podrozdziale skupiamy uwagę na najlepszych praktykach programowania proceduralnego. Techniki programowania obiektowo zorientowanego omówimy w rozdziale 7.

Organizowanie kodu w moduły za pomocą funkcji (kategorii)

Głównym powodem dzielenia kodu na moduły jest zwiększenie jego przejrzystości i ułatwienie późniejszego utrzymania aplikacji. W aplikacji proceduralnej procedury powinny być umieszczane w oddzielnych modułach, zgodnie z logiką działania. W modułach najlepiej grupować procedury pełniące podobne funkcje.

VBA ma nieudokumentowane „miękkie ograniczenie" maksymalnego rozmiaru standardowego modułu, którego wielkość nie powinna przekraczać 64 kB, przy czym wielkość ta dotyczy pliku tekstowego eksportowanego z projektu (narzędzia VBE, zapisane na płycie CD, automatycznie podają wielkość modułu). Twój projekt nie załamie się natychmiast, gdy pojedynczy moduł przekroczy wielkość 64 kB, lecz ciągłe przekraczanie tej granicy niemal na pewno doprowadzi do niestabilności aplikacji.

Funkcjonalna dekompozycja

Funkcjonalna dekompozycja to proces dzielenia aplikacji na oddzielne procedury w taki sposób, aby każda odpowiadała za wykonanie pojedynczego zadania. Teoretycznie wiele aplikacji mógłbyś pisać w postaci wielkich, pojedynczych, monolitycznych procedur.

Jednak takie postępowanie znacznie utrudniałoby proces debugowania i późniejszego utrzymania aplikacji. Stosując funkcjonalną dekompozycję, planujesz aplikację tak, by składała się z wielu procedur, z których każda będzie odpowiedzialna za ściśle określone zadanie, łatwe do zrozumienia, weryfikacji, udokumentowania i utrzymania.

Najlepsze praktyki tworzenia procedur

Zrozumiały zestaw wskazówek opisujących właściwy sposób tworzenia procedur z łatwością wypełniłby cały rozdział. Niżej podamy jedynie listę najważniejszych.

- **Hermetyzowanie** (enkapsulacja). Jeżeli tylko jest to możliwe, procedura powinna być zaprojektowana tak, aby zapewnić pełną hermetyzację wykonywanej operacji. Oznacza to np., że odpowiednio hermetyzowana procedura może zostać skopiowana do kompletnie różnego projektu, gdzie będzie działać równie dobrze, jak w projekcie, w którym powstała. Hermetyzacja pomaga w wielokrotnym stosowaniu kodu, a dzięki logicznemu odizolowaniu operacji upraszcza debugowanie.
- **Eliminowanie duplikowaniu kodu**. Pisząc nietrywialną aplikację Excela, często będziesz odkrywał, że już w wielu miejscach pisałeś kod wykonujący tę samą operację. Jeżeli to odkryjesz, powinieneś wydzielić powtarzający się kod, tworząc oddzielną procedurę. Czyniąc to, zmniejszysz liczbę miejsc, gdzie ta szczególna operacja musi być weryfikowana lub modyfikowana. Wspólna procedura może być modyfikowana tylko w jednym miejscu aplikacji. Wszystko to prowadzi do znacznego podniesienia jakości kodu. Służy to również drugiemu ważnemu celowi — wielokrotnej stosowalności kodu. Jeżeli powszechnie stosowane operacje wydzielisz w oddzielnych procedurach, przekonasz się, że będziesz mógł te procedury wykorzystać w wielu innych aplikacjach. Taki kod tworzy rodzaj biblioteki, której możesz używać w celu podniesienia własnej produktywności przy pisaniu następnych aplikacji. Im więcej logicznie oddzielonych operacji zapiszesz w postaci kompletnych, w pełni przetestowanych procedur bibliotecznych, tym mniej czasu będzie Ci zajmować tworzenie kolejnych aplikacji.
- **Izolowanie złożonych operacji**. W wielu rzeczywistych aplikacjach znajdziesz sekcje logiki biznesowej bardzo złożone i bardzo specyficzne dla danej aplikacji, dla której zostały zaprojektowane, co oznacza także niemożliwość powtórnego ich użycia w innych projektach. Takie sekcje należy izolować w oddzielnych procedurach w celu ułatwienia debugowania i dalszego utrzymania.
- **Redukcja rozmiarów procedury**. Zbyt długie procedury są trudne do zrozumienia, debugowania i utrzymania nawet dla programistów, którzy je napisali. Jeżeli odkryjesz procedurę licząca ponad 150 lub 200 linii kodu, przekonasz się, że zwykle wykonuje ona kilka zadań, a więc powinna być podzielona na kilka jednocelowych procedur.
- **Ograniczanie liczby argumentów procedury**. Im więcej argumentów procedura akceptuje, tym trudniej ją zrozumieć i mniej efektywnie będzie używana. W zasadzie powinieneś ograniczać liczbę argumentów do pięciu

lub mniej. I nie stosuj prostego zastępowania argumentów procedur przez zmienne publiczne lub dostępne na poziomie modułu. Jeżeli okaże się, że procedura wymaga więcej niż pięciu argumentów, będzie to znak, że sama procedura lub logika aplikacji powinny zostać ponownie zaprojektowane.

Najlepsze praktyki określające ogólne zasady tworzenia oprogramowania

W tym podrozdziale są omawiane najlepsze praktyki działania wspólne dla wszystkich obszarów i etapów tworzenia aplikacji. Większość dalszych rozdziałów zaleca stosowanie najlepszych praktyk odnoszących się do tematów poruszonych w tym rozdziale.

Komentowanie kodu

Dobre komentowanie kodu to najważniejsza z dobrych praktyk tworzenia aplikacji Excela. Komentarze powinny w sposób prosty i kompletny objaśniać, jak kod jest zorganizowany, w jaki sposób każdy obiekt i procedura powinny być używane oraz jaki był cel napisania danego kodu. Komentarze służą także do zaznaczania zmian dokonywanych wraz z upływem czasu, ale tym tematem zajmiemy się w dalszej części tego rozdziału.

Komentarze kodu są ważne zarówno dla Ciebie, jak i dla innych programistów, którzy — być może — będą nad Twoim kodem pracowali. Przydatność komentarzy dla innych zdaje się być oczywista. To, czego możesz sobie nie uświadamiać do czasu otrzymania pierwszej okrutnej lekcji, to przydatność komentarzy dla Ciebie. Często się zdarza, że po napisaniu pierwszej wersji kodu, po upływie jakiegoś czasu jego twórcy są proszeni o dokonanie istotnych zmian. Możesz być wtedy zaskoczony, jak obcy wyda Ci się własny kod, nieoglądany od dłuższego czasu. Komentarze pomagają radzić sobie z tym problemem.

Komentarze powinny być stosowane we wszystkich trzech głównych poziomach kodu aplikacji: na poziomie modułu, procedury i pojedynczych sekcji lub linii kodu. Omówimy dalej typy komentarzy odpowiednie dla tych wszystkich poziomów.

Komentarze na poziomie modułu

Jeżeli użyłeś, opisanej wcześniej w tym rozdziale, konwencji nazywania modułów, każdy sprawdzający kod będzie miał z grubsza pojęcie o celu kodu zawartego w module. Powinieneś to wesprzeć krótkim komentarzem na początku każdego modułu, gdzie zawrzesz dokładniejszy opis celu kodu zapisanego w module.

Gdy mówiąc o komentarzach, używamy terminu ***moduł***, określamy nim moduły standardowe, moduły klas i moduły kodu związanego z formularzami użytkownika.

Dobry komentarz na poziomie modułu powinien być umieszczony na samym początku i wyglądać tak, jak przykład pokazany w listingu 3.2.

Listing 3.2. *Przykładowy komentarz na poziomie modułu*

```
'
' Opis:    Krótki opis celu kodu
'          zapisanego w tym module.
'
Option Explicit
```

Komentarze na poziomie procedury

Zwykle komentarze na poziomie procedury są najdokładniejszymi komentarzami w całej aplikacji. Tam opisujesz cel procedury, uwagi o jej używaniu, szczegółową listę argumentów i informacje o celu ich użycia oraz — w przypadku funkcji — oczekiwaną zwracaną wartość. Komentarze na poziomie procedury mogą także służyć jako podstawowe narzędzie rejestrowania zmian, gdzie można podawać daty i opisy zmian wprowadzanych w procedurze. Dobry komentarz na poziomie procedury, taki jak pokazany w listingu 3.3, może być umieszczony bezpośrednio nad pierwszą linią kodu procedury. Komentarz w listingu 3.3 został napisany dla funkcji. Komentarz dla procedury typu `Sub` od komentarza dla funkcji różni się jedynie brakiem bloku `Zwraca` (*Returns*), gdyż — jak wiadomo — procedury tego typu nie zwracają żadnych wartości.

Listing 3.3. *Przykładowy komentarz na poziomie procedury*

```
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Komentarze:   Lokalizuje wykres, który ma być
'              obiektem działania lub - jeżeli
'              wykresów jest wiele - prosi
'              o wybranie właściwego.
'
' Argumenty:    chtWykres   Wartość zwracana przez
'                           funkcję. Odwołanie do
'                           wykresu, na którym ma być
'                           wykonane działanie lub
'                           nic, jeżeli użytkownik
'                           anuluje działanie.
'
' Zwraca:       Boolean     Prawda — gdy powodzenie,
'                           Fałsz — gdy błąd lub
'                           anulowanie przez
'                           użytkownika.
'
' Data      Programista     Działanie.
'---------------------------------------------------------
' 04/07/02  Rob Bovey       Utworzenie.
' 14/10/03  Rob Bovey       Wychwytywanie błędu,
'                           gdy wykres bez serii.
' 18/11/03  Rob Bovey       Wychwytywanie błędu,
'                           gdy brak aktywnego
'                           arkusza.
```

Komentarze wewnętrzne

Komentarze wewnętrzne są umieszczane wewnątrz samego kodu, aby opisać jego cel, jeżeli ten może się zdawać nieoczywisty. Powinny opisywać raczej ***intencję*** kodu niż wykonywane ***operacje***. Rozróżnienie intencji i operacji nie zawsze jest jasne, więc w listingach 3.4 i 3.5 podaliśmy dwa przykłady tego samego kodu, odpowiednio ze złym i dobrym komentarzem.

Listing 3.4. *Przykład złego wewnętrznego skomentowania kodu*

```
' Pętla tablicy asPlikiWejsciowe
For lIndex = LBound(asPlikiWejsciowe) To UBound(asPlikiWejsciowe)
    ' ...
Next lIndex
```

Komentarz w listingu 3.4 jest całkowicie nieprzydatny. Po pierwsze, przede wszystkim opisuje jedynie linię kodu znajdującą się bezpośrednio poniżej, nie dając żadnej informacji o celu całej pętli. Po drugie, komentarz jest jedynie dokładnym opisem tej linii. Jest to informacja, jaką łatwo uzyskać, rzucając okiem na linię kodu. Usunięcie komentarza pokazanego w listingu 3.4 nie przyniesie żadnej szkody.

Listing 3.5. *Przykład dobrego wewnętrznego skomentowania kodu*

```
' Import określonej listy plików wejściowych
' do obszaru roboczego arkusza danych.
For lIndex = LBound(asPlikiWejsciowe) To UBound(asPlikiWejsciowe)
    ' ...
Next lIndex
```

Komentarz z listingu 3.5 wnosi do kodu nową wartość. Nie tylko opisuje intencję zamiast operacji, lecz również wyjaśnia strukturę pętli. Po przeczytaniu tego komentarza będziesz wiedzieć, czego masz szukać, szperając w kodzie wnętrza pętli.

Ale od każdej zasady są wyjątki, więc również podana wcześniej zasada tworzenia komentarzy wewnątrz kodu nie zawsze obowiązuje. Najważniejsze odstępstwo dotyczy komentarzy mających objaśniać struktury sterujące. Instrukcje `If...Then` i `Do...Loops` w miarę rozbudowy czynią kod trudniejszym do zrozumienia, gdyż staje się niemożliwe śledzenie go w jednym oknie. Trudno wówczas pamiętać, do jakiego miejsca kodu jest przekazywane sterowanie aplikacją. Przykładowo podczas testowania długiej procedury często trafiamy na coś takiego, co widać na wyimku listingu pokazanym w listingu 3.6.

Listing 3.6. *Trudna do śledzenia struktura sterująca*

```
            End If

            lLiczbaPlikowWejsc = lLiczbaPlikowWejsc - 1

        Loop

    End If
```

Z listingu 3.6 nie można odczytać, jakie testy logiczne sterują dwiema instrukcjami `If...Then` oraz jakie wyrażenie steruje pętlą `Do...While`. Po wypełnieniu tych struktur rzeczywistym kodem nie będzie możliwe znalezienie odpowiedzi na powyższe pytania bez przewijania listingu tam i z powrotem, gdyż cały blok przestanie być widoczny w jednym oknie kodu. Możesz łatwo zmniejszyć tę niedogodność, dodając komentarze w stylu tych, które zostały pokazane w listingu 3.7.

Listing 3.7. *Zrozumiała struktura sterująca*

```
            End If  ' If bZawartoscWazna Then

            lLiczbaPlikowWejsc = lLiczbaPlikowWejsc - 1

        Loop        ' Do While lLiczbaPlikowWejsc > 0

    End If          ' If bPlikiWejscZnalezione Then
```

Choć komentarze w listingu 3.7 są jedynie powtórzeniem kodu, przyczepionym do instrukcji kończących struktury sterujące, dzięki nim sens tych struktur staje się jasny i oczywisty. Tego typu komentarzy należy używać wszędzie tam, gdzie struktury sterujące są zbyt rozciągnięte, aby można je było obejrzeć w jednym oknie kodowym.

Unikanie najgorszego błędu komentowania kodu

Problem jest oczywisty, mimo to jednak najpoważniejszym, najczęstszym i najbardziej szkodliwym błędem komentowania jest brak aktualizacji komentarzy przy modyfikacji kodu. Często spotykamy projekty na pierwszy rzut oka przygotowane zgodnie z dobrą praktyką komentowania kodu, w których po bliższym sprawdzeniu okazuje się, że komentarze dotyczą jakiejś pierwotnej wersji projektu i ich związek z wersją bieżącą jest niemal żaden.

Jeżeli staramy się zrozumieć kod, błędne komentarze są gorsze niż ich brak, gdyż prowadzą na mylne ścieżki. Dlatego zawsze aktualizuj komentarze, zaś stare albo kasuj, albo utrzymuj w formie pozwalającej śledzić ciąg wprowadzanych kolejno zmian. Zalecamy kasowanie nieaktualnych komentarzy wewnątrz kodu, aby uniknąć zaśmiecania go nieprzydatnymi informacjami, bowiem nadmiar linii komentujących sprawia, że kod jest trudny do zrozumienia.

Do śledzenia wprowadzanych zmian używaj komentarzy na poziomie procedury.

Czytelność kodu

Czytelność kodu jest funkcją jego fizycznego układu. Dobry wizualny rozkład kodu pozwala na proste wnioskowanie znacznej ilości informacji o logicznej strukturze programu. To sprawa kluczowa. Dla komputera rozkład kodu nie ma żadnego znaczenia. Jedynym celem jest ułatwienie zrozumienia kodu ludziom. Tak samo jak konwencja nazw, również spójna i dobra konwencja rozkładu kodu jest rodzajem samodokumentacji. Podstawowym narzędziem rozkładu kodu jest puste miejsce (*white*

space). Pustymi miejscami są znaki spacji, tabulatory i puste linie. W następnych akapitach omówimy najważniejsze sposoby używania pustych miejsc w celu zbudowania dobrze zaprojektowanego rozkładu kodu.

Grupuj spokrewnione elementy kodu, zaś elementy niemające ze sobą związku oddzielaj pustymi liniami. Można uznać, że sekcje kodu procedury oddzielone pustymi liniami pełnią taką samą rolę, jak akapity w tekście książki. Pozwalają określić, co jest ze sobą powiązane. W listingu 3.8 widzimy przykład pokazujący, jak puste linie zwiększają czytelność kodu. Nawet bez dodawania komentarzy wiadomo, które linie kodu są ze sobą powiązane.

Listing 3.8. *Użycie pustych linii do grupowania kodu w sekcje*

```
' Ustawianie właściwości aplikacji.
Application.ScreenUpdating = True
Application.DisplayAlerts = True
Application.EnableEvents = True
Application.StatusBar = False
Application.Caption = Empty
Application.EnableCancelKey = xlInterrupt
Application.Cursor = xlDefault

' Usuwanie wszystkich pasków narzędziowych użytkownika.
For Each cbrBar In Application.CommandBars
    If Not cbrBar.BuiltIn Then
        cbrBar.Delete
    Else
        cbrBar.Enabled = True
    End If
Next cbrBar

' Przywrócenie paska menu arkusza.
With Application.CommandBars(1)
    .Reset
    .Enabled = True
    .Visible = True
End With
```

Wewnątrz sekcji spokrewnionych linii związek poszczególnych grup pokazujemy za pomocą wyrównania. Wcięcia służą do pokazania logicznej struktury kodu. W listingu 3.9 demonstrujemy jedną sekcję listingu 3.8, gdzie wyrównania i wcięcia dają dobry efekt. Przyglądając się temu fragmentowi kodu, od razu zrozumiesz, które elementy tworzą całości oraz rozpoznasz logiczny bieg wykonywanego kodu.

Listing 3.9. *Właściwe użycie wyrównania i wcięć*

```
' Usuwanie wszystkich pasków narzędziowych użytkownika.
For Each cbrBar In Application.CommandBars
    If Not cbrBar.BuiltIn Then
        cbrBar.Delete
    Else
        cbrBar.Enabled = True
    End If
Next cbrBar
```

W celu ułatwienia czytania kodu długie instrukcje i deklaracje dzielimy za pomocą znaku kontynuacji linii. Pamiętaj, że dzielenie kodu w ten sposób jedynie dla pokazywania całych linii bez konieczności przewijania okna niekoniecznie musi być dobrą praktyką. W listingu 3.10 widać przykład rozsądnego stosowania kontynuacji linii.

Listing 3.10. *Rozsądne stosowanie kontynuacji linii*

```
' Złożone wyrażenia łatwiej zrozumieć,
' jeżeli są właściwie kontynuowane.
If (uData.lMaxLocationLevel > 1) Or _
  uData.bHasClientSubsets Or _
  (uData.uDemandType = bcDemandTypeCalculate) Then

End If

' Stosowanie kontynuacji linii ułatwia czytanie
' długich deklaracji API.
Declare Function SHGetSpecialFolderPath Lib "Shell32.dll" _
  (ByVal hwndOwner As Long, _
  ByRef szBuffer As String, _
  ByVal lFolder As Long, _
  ByVal bCreate As Long) As Long
```

Najlepsze praktyki programowania w VBA

Najlepsze ogólne praktyki programowania w VBA

Dyrektywy dla modułu

- `Option Explicit`. Zawsze w każdym module używaj instrukcji `Option Explicit`. Nie sposób przecenić ważności takiego postępowania. Bez `Option Explicit` każda literówka spowoduje utworzenie przez VBA nowej zmiennej typu `Variant`. Tego rodzaju błąd jest bardzo zdradziecki, gdyż czasem nawet nie powoduje natychmiastowego błędu wykonania. Jednak ostatecznie rezultaty działania aplikacji będą błędne. Błąd tego rodzaju ma wielkie szanse, by nie zostać zauważonym aż do rozpowszechnienia aplikacji i w niektórych okolicznościach będzie go trudno wykryć.

  Instrukcja `Option Eplicit` wymusza jawne deklarowanie wszystkich używanych zmiennych i powoduje wyświetlenie przez VBA błędu kompilacji (po wybraniu z menu VBE polecenia *Debug/Compile*) natychmiast po zidentyfikowaniu nierozpoznawalnej nazwy. Dzięki temu znajdowanie literówek staje się łatwe. Aby mieć pewność, że instrukcja `Option Explicit` będzie automatycznie umieszczana na górze każdego tworzonego modułu, powinieneś z menu VBE wybrać polecenie *Tools/Options/Editor* i włączyć opcję *Require Variable Declaration*. Takie postępowanie jest bardzo zalecane.

- `Option Private Module`. Instrukcja `Option Private Module` sprawia, że wszystkie procedury z modułu, w którym została użyta, są niedostępne za pomocą menu użytkownika Excela oraz w innych projektach Excela. Używaj tej instrukcji, aby ukryć procedury, które nie mogą być wywoływane spoza aplikacji.

Metoda `Applicatin.Run` pozwala obchodzić instrukcję `Option Private Module` i uruchamiać prywatne procedury mimo wprowadzonego ograniczenia.

- `Option Base 1`. Instrukcja `Option Base 1` nadaje wartość 1 dolnym granicom indeksów wszystkich tablic, dla których granica ta nie została oddzielnie ustalona. Nie używaj tej instrukcji. Lepiej dla wszystkich używanych indeksów tablic określaj granice górne i dolne. Procedura utworzona w module, w którym użyto instrukcji `Option Base 1`, może nie działać poprawnie po skopiowaniu do modułu, gdzie tej instrukcji brakuje, a to oznacza niespełnienie jednej z podstawowych zasad projektowania aplikacji, jaką jest możliwość powtórnego użycia.
- `Option Compare Text`. Instrukcja `Option Compare Text` w module, w którym została użyta, wymusza tekstowe, zamiast binarnego, porównywanie łańcuchów tekstowych. Przy takim porównaniu litery wielkie i małe są traktowane jako tożsame, zaś przy porównaniu binarnym są uznawane za różne. Instrukcji `Option Compare Text` należy unikać z takich samych powodów, jak instrukcji `Option Base 1`. Powoduje ona inne działanie procedur po umieszczeniu ich w modułach, gdzie nie została użyta. Ponadto porównania tekstowe są o wiele bardziej kosztowne obliczeniowo niż porównania binarne. Zatem użycie instrukcji `Option Compare Text` powoduje spowolnienie wszystkich porównań tekstowych dokonywanych w module, w którym została użyta. Większość funkcji Excela i VBA wykorzystywanych do porównywania tekstów posiada argument pozwalający na określenie, czy ma być wykonane porównanie binarne, czy tekstowe. Gdy trzeba użyć porównania tekstowego, o wiele lepiej używać tych argumentów.

Jedynie w kilku rzadkich przypadkach użycie `Option Compare Text` jest wymagane. Najczęstszym jest porównywanie łańcuchów za pomocą operatora VBA `Like`, bez rozróżniania liter wielkich i małych. Jedynym sposobem zmuszenia operatora `Like` do nieodróżniania liter wielkich i małych jest zastosowanie instrukcji `Option Compare Text`. Powinieneś wówczas procedury wymagające użycia tej instrukcji umieścić w oddzielnym module, aby je odseparować od innych, którym skażenie tą instrukcją nie jest potrzebne. Upewnij się, że przyczyny takiego działania wyjaśniłeś odpowiednio w komentarzu na poziomie modułu.

Zmienne i stałe

Unikaj ponownego używania zmiennych. Każda zmienna zadeklarowana w programie powinna służyć tylko jednemu celowi. Wielokrotne używanie do różnych celów tych samych zmiennych oszczędza jedynie linię deklaracji zmiennej, ale sprowadza na program wielkie niebezpieczeństwo potencjalnego zamieszania. Jeżeli próbujesz określić, w jaki sposób procedura działa i pamiętasz, do czego służyła określona zmienna w innym miejscu, w sposób naturalny zakładasz, że przy kolejnym użyciu ma wykonać to samo. Jeżeli nie jest to prawdą, trudno będzie zrozumieć logikę takiego kodu.

Unikaj używania zmiennych typu `Variant`. Jeżeli tylko jest to możliwe, nie stosuj zmiennych typu `Variant`. Niestety, VBA nie jest językiem wymagającym dużej dyscypliny pisania kodu. Dlatego możesz po prostu deklarować zmienne bez określania

ich typu, a VBA utworzy je jako zmienne typu `Variant`. Przyczyny unikania używania zmiennych `Variant` są następujące.

- **Zmienne typu `Variant` są bardzo nieefektywne**. Wynika to z faktu, że zmienne `Variant` są bardzo skomplikowanymi strukturami, przystosowanymi do przechowywania danych wszelkich typów, które mogą być stosowane w języku programowania VBA. Do wartości `Variant` nie ma bezpośredniego dostępu i nie mogą być one modyfikowane bezpośrednio, jak w przypadku zmiennych podstawowych typów, takich jak `Long` i `Double`. Do wykonania jakiejkolwiek operacji na zmiennej `Variant` VBA musi używać ukrytej przed użytkownikiem serii wywołań Windows API.
- **Dane przechowywane w zmiennych typu `Variant` mogą zachowywać się w sposób niespodziewany**. Ponieważ zmienne `Variant` są zaprojektowane do przechowywania danych dowolnego typu, to, co wkładamy do takiej zmiennej, nie zawsze jest tym samym, co otrzymamy z powrotem. Uzyskując dostęp do danej `Variant`, VBA usiłuje narzucić (*coerce*) typ, jaki uważa za najbardziej sensowny w kontekście danego działania. Jeżeli musisz stosować zmienne `Variant`, zawsze przed użyciem dokonuj jawnego rzutowania danych na właściwy typ.

Bądź świadom diabła, który tkwi w szczegółach narzucania typu danych (ETC — evil type corecion). ETC to kolejny objaw wynikający z faktu, że VBA nie jest językiem wymagającym ścisłej dyscypliny pisania kodu. W rezultacie zdarza się, że VBA dokonuje automatycznej konwersji jednego typu danych na inny, całkowicie niezwiązany z poprzednim. Najczęściej zdarza, że zmienne `String` przechowujące liczby są konwertowane na typ `Integer`, a zmienne `Boolean` na równoważniki typu `String`. Nie mieszaj zmiennych różnego typu, bez informowania VBA, w jaki sposób te zmienne mają być traktowane, co powinieneś robić przez ich rzutowanie na właściwy typ za pomocą specjalnych funkcji rzutowania (np. `CStr`, `CLng` lub `CDbl`).

Unikaj deklaracji o składni As New. Nigdy do deklarowania zmiennych nie stosuj składni `As New`. Przykładowo podana niżej postać deklaracji zmiennej obiektowej nie powinna być nigdy użyta:

```
Dim rsData As new ADODB.Recordset
```

Jeżeli VBA napotka linię kodu, w której ta zmienna zostanie użyta bez wcześniejszego zainicjalizowania, automatycznie utworzy jej nowy egzemplarz. ***Nigdy*** nie będzie to działanie, jakiego pragniesz. Dobra praktyka programowania wymaga, aby programista utrzymywał pełną kontrolę nad wszystkimi obiektami używanymi w programie. Jeżeli VBA napotyka w kodzie niezainicjalizowaną zmienną obiektową, niemal na pewno jest to spowodowane przez błąd, o którym chciałbyś być natychmiast poinformowany. Dlatego właściwe deklarowanie zmiennych obiektowych powinno wyglądać, tak jak poniżej:

```
Dim rsData As ADODB.Recordset
Set rsData = New ADODB.Recordset
```

Jeżeli użyjesz takiej deklaracji i inicjalizacji, a zmienna obiektowa zostanie zniszczona gdzieś w procedurze, po czym nieświadomie się do tej zmiennej odwołasz, VBA natychmiast powiadomi Cię o tym, generując błąd egzekucji: „Object variable or With block variable not set”.

Zawsze przeprowadzaj pełną kwalifikację nazw obiektowych. Zawsze w deklaracjach zmiennych i w kodzie stosuj pełną kwalifikację nazw obiektowych, podając nazwę wraz z przedrostkiem klasy. Należy tak robić, gdyż wiele bibliotek posiada obiekty tak samo nazwane. Jeżeli zadeklarujesz zmienną obiektową, podając jedynie nazwę obiektu, a aplikacja odwołuje się do wielu bibliotek, VBA utworzy zmienną z pierwszej biblioteki z listy *Tools/References*, w jakiej znajdzie tak nazwany obiekt. Nie zawsze jest to obiekt, jakiego potrzebujesz.

Kontrolki formularzy `UserForm` są przykładem najczęstszych błędów powodowanych przez deklarowanie nazw nie w pełni kwalifikowanych. Jeżeli np. chcesz zadeklarować zmienną obiektową odwołującą się do kontrolki `TextBox` na Twoim formularzu, możesz próbować zrobić to następująco:

```
Dim txtBox As TextBox
Set txtBox = Me.TextBox1
```

Niestety, gdy tylko VBA spróbuje wykonać drugą linię kodu, zostanie wygenerowany błąd „Type mismatch". Stanie się tak dlatego, że biblioteka obiektowa Excela zawiera obiekt `TextBox`, a na liście *Tools/References* znajduje się przed biblioteką `MSForms`. Właściwy sposób zapisania tego kodu jest następujący:

```
Dim txtBox As MSForms.TextBox
Set txtBox = Me.TextBox1
```

Nigdy nie ustalaj sztywnych granic indeksów tablic. Jeżeli przebiegasz w pętli przez wartości tablicy, nigdy nie używaj sztywnych granic indeksowania. W zamian stosuj funkcje `LBound` i `UBound`, tak jak w listingu 3.11.

Listing 3.11. *Właściwy sposób przebiegania w pętli przez wartości tablicy*

```
Dim lIndex As Long
Dim alListItems(1 To 10) As Long

' Tutaj załaduj tablicę.

For lIndex = LBound(alListItems) To UBound(alListItems)
    ' Zrób coś z każdą wartością.
Next lIndex
```

Powodem takiego postępowania jest fakt, że dolne i górne granice indeksów tablic często zmieniają się podczas tworzenia i konserwacji aplikacji. Jeżeli w pętli pokazanej powyżej ustalisz sztywne granice 1 i 10, będziesz musiał pamiętać o aktualizowaniu tych wartości po każdej zmianie tablicy `alListItems`. Pętla utworzona z użyciem `LBound` i `UBound` aktualizuje się sama.

Po każdej instrukcji Next zawsze podawaj nazwę licznika pętli. W listingu 3.11 demonstrujemy jeszcze jedną dobrą praktykę kodowania. Po instrukcji `Next` zawsze należy podawać nazwę zmiennej indeksowej pętli. Przestrzeganie tej zasady, choć nie jest to ściśle wymagane przez VBA, sprawia, że kod jest łatwiejszy do zrozumienia, zwłaszcza gdy odległość między instrukcjami `For` i `Next` jest duża.

Używaj stałych. Stałe są bardzo przydatnymi elementami programowania. Służą między innymi następującym celom.

- Eliminują „magiczne liczby", zastępując je rozpoznawalnymi nazwami, np. co może oznaczać liczba 50 w podanej niżej linii kodu?

  ```
  If lIndex < 50 Then
  ```

  Nie można się tego dowiedzieć. Może to wiedzieć jedynie autor kodu, jeżeli jeszcze nie zapomniał. Jeśli jednak zobaczysz tekst napisany, tak jak poniżej, z łatwością odgadniesz jego sens.

  ```
  Const lMAKS_LICZBA_PLIKOW_WEJSC As Long = 50

  ' Jakiś kod.

  If lIndex < lMAKS_LICZBA_PLIKOW_WEJSC Then
  ```

  Jeżeli podczas pisania kodu będziesz chciał poznać wartość stałej, możesz kliknąć prawym przyciskiem jej nazwę i wybrać z menu podręcznego polecenie *Definition*, a zostaniesz natychmiast przeniesiony do linii, gdzie ta stała została zdefiniowana. W trybie przerwania wykonania jest jeszcze łatwiej. Zwykłe naprowadzenie kursora myszy na stałą spowoduje wyświetlenie okienka podpowiedzi z jej wartością.

- Stałe poprawiają efektywność kodowania i ułatwiają unikanie błędów duplikacji danych. Załóżmy, że w poprzednim przykładzie odwoływałeś się wielokrotnie do maksymalnej liczby plików wejściowych i w pewnym momencie będziesz musiał przystosować program do obsługi większej liczby plików. Jeżeli w różnych miejscach wpisałeś na sztywno maksymalną liczbę plików wejściowych, będziesz musiał odszukać wszystkie te miejsca i zmienić podaną wartość. Jeśli użyłeś stałej, musisz zmienić jej wartość tylko w jednej deklaracji, a zostanie ona automatycznie zaktualizowana we wszystkich miejscach kodu, gdzie występuje. Taka sytuacja jest bardzo częstą przyczyną błędów, których z łatwością możesz uniknąć, używając stałych zamiast wartości wpisywanych na sztywno.

Zasięg zmiennej

Zmienne publiczne są niebezpieczne. Mogą być bez ostrzeżenia modyfikowane w dowolnym miejscu aplikacji, z powodu tego ich wartość bywa nieprzewidywalna. Przeczy to jednej z podstawowych zasad programowania, czyli hermetyzacji. Zawsze twórz zmienne z minimalnym możliwym zasięgiem. Zaczynaj od zmiennych lokalnych (na poziomie procedury) i rozszerzaj ten zasięg jedynie wtedy, gdy jest to bezwzględnie konieczne.

Tak jak w przypadku większości naszych zasad, tak i tym razem istnieje kilka wyjątków, gdy zmienne publiczne są przydatne, a nawet konieczne.

- Gdy zmienne przed użyciem muszą być przekazane daleko w głąb stosu. Jeżeli np. procedura A czyta pewne dane, potem przekazuje je procedurze B, która przekazuje je procedurze C, która przekazuje je procedurze D, gdzie ostatecznie są w jakimś celu użyte, lepiej przekazać je bezpośrednio z procedury A do D za pomocą zmiennej publicznej.

- Pewne naturalnie publiczne klasy, jak klasa obsługi zdarzeń na poziomie aplikacji, wymagają publicznych zmiennych obiektowych, które nigdy nie znajdą się poza zasięgiem podczas działania aplikacji.

Wczesne i późne wiązanie

Rozróżnienie między wczesnym wiązaniem (*early binding*) a późnym wiązaniem (*late binding*) jest często źle rozumiane i mylone ze sposobem tworzenia obiektu. Wczesne lub późne wiązanie zmiennej zależy ***jedynie*** od sposobu zadeklarowania zmiennej przechowującej odwołanie do obiektu. Zmienne zadeklarowane z typem określonego obiektu są zawsze wcześnie wiązane. Zmienne zadeklarowane jako zmienne typu `Object` lub `Variant` są zawsze późno wiązane. W listingu 3.12 pokazujemy przykład późnego wiązania odwołania, a w listingu 3.13 — wczesnego.

Listing 3.12. *Późne wiązanie odwołania do obiektu ADO Connection*

```
Dim objPolaczenie As Object

' Niezależnie od sposobu utworzenia obiektu, będzie on
' późno wiązany z powodu zadeklarowania zmiennej
' As Object.
Set objPolaczenie = New ADODB.Connection
Set objPolaczenie = CreateObject("ADODB.Connection")
```

Listing 3.13. *Wczesne wiązanie odwołania do obiektu ADO Connection*

```
Dim cnPolaczenie As ADODB.Connection

' Niezależnie od sposobu utworzenia obiektu, będzie on
' wcześnie wiązany z powodu typu danych użytego
' w deklaracji zmiennej.
Set cnPolaczenie = New ADODB.Connection
Set cnPolaczenie = CreateObject("ADODB.Connection")
```

Jeżeli używasz wczesnego wiązania z obiektem spoza modelu obiektowego Excela, pamiętaj, że wcześniej musisz ustanowić odwołanie do odpowiedniej biblioteki obiektowej za pomocą polecenia *Tools/References* z menu Visual Basic Editora. Aby np. utworzyć wczesne wiązanie zmiennej odwołującej się do obiektów ADO, musisz ustanowić odwołanie do biblioteki Microsoft ActiveX Data Objects 2.x Library, gdzie x oznacza wersję ADO, jakiej zamierzasz używać.

Powinieneś stosować wczesne wiązania zmiennych wszędzie, gdzie jest to możliwe. Wczesne wiązanie zmiennych, w porównaniu z późnym wiązaniem, daje następujące korzyści.

- **Poprawa wydajności**. Jeżeli używasz zmiennej obiektowej, której typ danych jest znany VBA już podczas kompilacji, VBA może poszukać miejsca dla wszystkich właściwości i metod obiektu, z jakich w kodzie korzysta i zapisać je wraz z kodem. Jeżeli potem, podczas działania programu, nastąpi odwołanie do takiej metody lub własności, VBA po prostu uruchomi kod zapisany lokalnie

(jest to pewne uproszczenie, gdyż w rzeczywistości VBA przechowuje offset kodu, który ma być uruchamiany ze znanego punktu startowego w pamięci, będącego początkiem struktury nazywanej `VTable` danego obiektu).

Jeżeli stosujesz późne wiązanie zmiennej obiektowej, VBA nie może z góry przewidzieć, jaki typ obiektu zmienna będzie przechowywać, więc w czasie kompilacji nie może optymalizować wywołania żadnej właściwości ani metody. Oznacza to, że podczas działania programu po każdym wywołaniu metody lub właściwości zmiennej wiązanej późno VBA musi poszukać tej zmiennej, aby ustalić, jaki typ obiektu zawiera, wyszukać nazwę wywołanej metody lub właściwości, by odnaleźć ją w pamięci i wykonać kod przechowywany pod danym adresem. Jest to proces znacznie wolniejszy niż w przypadku zmiennej wcześnie wiązanej.

- **Ścisłe weryfikowanie typu**. Jeżeli w przykładzie późnego wiązania z listingu 3.12 przypadkowo przypiszesz zmiennej obiektowej odwołanie do obiektu `ADO Command` zamiast `ADO Connection`, VBA nie wykryje błędu. O tym, że został popełniony błąd, dowiesz się później w dalszej części kodu, gdy spróbujesz użyć metody lub właściwości nieobsługiwanej przez obiekt `Command`. W przypadku wczesnego wiązania VBA natychmiast wykryje, że próbujesz przypisać do zmiennej obiektowej odwołanie błędnego typu i powiadomi Cię o tym komunikatem „Type mismatch". Nieprawidłowe odwołania do właściwości i metod mogą być wykrywane wcześniej, jeszcze przed uruchomieniem kodu. VBA spróbuje odnaleźć w bibliotece nazwę właściwości lub metody już w czasie kompilacji i zgłosi błąd, jeżeli takiej nazwy nie uda się odszukać.
- **Dostępność IntelliSense**. Wczesne wiązanie zmiennych obiektowych ułatwia również samo programowanie. Ponieważ VBA wie dokładnie, jaki typ obiektu zmienna będzie reprezentować, może zanalizować właściwą bibliotekę obiektową i wyświetlić rozwijaną listę wszystkich właściwości i metod dostępnych dla danego obiektu zaraz po napisaniu kropki po nazwie zmiennej.

Jak możesz oczekiwać, w niektórych przypadkach należy używać późnego wiązania zamiast wczesnego. Dwie główne przyczyny używania raczej późnego niż wczesnego wiązania są następujące.

1. Późniejsza wersja biblioteki obiektowej nie jest zgodna z wcześniejszą.

   Taka sytuacja zdarza się aż nazbyt często. Jeżeli ustalisz odwołania do późniejszej wersji biblioteki obiektowej, a potem spróbujesz uruchomić aplikację na komputerze z wcześniejszą wersją, natychmiast pojawi się błąd kompilacji „Can't find project or library" i odwołanie na maszynie docelowej zostanie oznaczone prefiksem `MISSING`. W przypadku tego błędu najbardziej zdradzieckie jest to, że linia kodu źródłowego oznaczona jako błędna zwykle nie ma nic wspólnego z biblioteką obiektową, która jest rzeczywistą przyczyną problemu.

   Jeżeli musisz używać obiektów aplikacji, która powoduje takie błędy i chcesz umożliwić użytkownikom korzystanie z dowolnej wersji aplikacji, powinieneś zastosować wyłącznie późne wiązania dla wszystkich zmiennych odwołujących się do obiektów w tej aplikacji. Jeżeli tworzysz nowe obiekty, powinieneś użyć funkcji `CreateObject` z niezależnym od wersji identyfikatorem `ProgID` obiektu, jaki chcesz utworzyć, zamiast składni `= New ObjectName`.

2. Chcesz użyć aplikacji, której — być może — nie ma na komputerze użytkownika i sam nie możesz jej zainstalować.

   W takim przypadku powinieneś skorzystać z późnego wiązania, co pozwoli uniknąć błędu kompilacji, jaki wystąpiłby natychmiast po próbie uruchomienia aplikacji odwołującej się do nieistniejącej na danym komputerze biblioteki obiektowej. Twoja aplikacja może wówczas sprawdzić istnienie potrzebnej biblioteki obiektowej i — jeżeli nie będzie ona zainstalowana — spokojnie zakończyć działanie.

Jeżeli nawet ostatecznie użyjesz w kodzie późnego wiązania, wczesne wiązanie tak bardzo zwiększa efektywność programowania, że warto je stosować podczas pisania i testowania aplikacji. Zamiany na późne wiązania dokonaj dopiero w ostatniej fazie testów przed dystrybucją.

Kodowanie defensywne

Kodowanie defensywne odnosi się do wielu praktyk programistycznych obmyślanych w celu zapobieżenia powstawaniu błędów i unikaniu ich późniejszego poprawiania.

Pisz aplikację w najstarszej wersji Excela, w jakiej — jak oczekujesz — może być uruchamiana

Choć zespół Microsoftu tworzący Excela bardziej niż inne przyłożył się do zapewnienia zgodności z poprzednimi wersjami, nie udało się jednak wykluczyć wielu subtelnych różnic. Jeżeli dobrze znasz najnowszą wersję, jest bardzo prawdopodobne, że napiszesz aplikację, która nie będzie działała w wersjach poprzednich, gdyż użyjesz jakiejś cechy, jaka kiedyś nie istniała.

Rozwiązaniem jest rozpoczynanie tworzenia aplikacji od najstarszej wersji, w jakiej ma ona działać. Może to zmusić Cię do utrzymywania na jednym komputerze wielu wersji Excela lub, co jest o wiele lepsze, posiadania kilku komputerów z zainstalowanymi różnymi wersjami. Tak czy inaczej, w praktyce jest to sprawa podstawowa. Jeżeli utworzysz aplikację w Excelu 2000, po czym dasz ją użytkownikowi Excela 97 i przekonasz się, że nie działa, będziesz musiał debugować i usuwać wiele fragmentów niesprawnego kodu. Oszczędzisz wiele czasu i nerwów przez rozpoczęcie tworzenia aplikacji w Excelu 97.

Jawnie używaj ByRef i ByVal

Jeżeli procedura posiada argumenty, można je deklarować jako `ByRef` i `ByVal`.

- `ByRef` oznacza przekazywanie adresu zmiennej w pamięci zamiast wartości tej zmiennej. Jeżeli wywołana procedura modyfikuje argument `ByRef`, modyfikacja będzie widoczna w procedurze wywołującej.
- `ByVal` oznacza przekazywanie do procedury wartości zmiennej. Procedura może zmieniać argument `ByVal`, ale te zmiany nie będą widoczne w procedurze wywołującej. W rzeczywistości procedura używa argumentów `ByVal`, tak jak zmiennych deklarowanych lokalnie.

Zawsze deklaruj jawnie argumenty procedury jako `ByRef` lub `ByVal`. Jeżeli pominiesz specyfikację, argumenty zostaną domyślnie zadeklarowane jako `ByRef`. Jeżeli nie ma konieczności, aby procedura wywołująca znała zmiany argumentów, deklaruj argumenty jako `ByVal`, co zabezpieczy Cię przed zmianami przekazywanymi wstecz do procedury wywołującej.

Jedyne wyjątki to przekazywanie długich łańcuchów, które są o wiele efektywniej przekazywane za pomocą argumentów `ByRef`, oraz argumenty tablicowe, których nie można przekazać w trybie `ByVal`. Musisz pamiętać, że deklarowanie argumentów procedury `ByVal` powoduje większe narażenie na narzucanie typu danych (ETC — *evil type coercion*). Argumenty procedury `ByRef` ***muszą*** przekazywać typ danych, jaki został zadeklarowany, gdyż w przeciwnym razie zostanie zgłoszony błąd kompilacji. Natomiast w przypadku przekazywania do procedury wartości za pomocą argumentu `ByVal` VBA będzie usiłował wymusić zgodność typu danych.

Jawnie wywołuj domyślną właściwość obiektu

Z wyjątkiem właściwości `Item` obiektu `Collection`, nigdy nie należy wywoływać właściwości obiektu w sposób niejawny jedynie przez używanie w wyrażeniu jego nazwy. W listingu 3.14 pokazujemy właściwy i niewłaściwy dostępu do właściwości obiektu na przykładzie kontrolki `MSForms.TextBox` (`Text` jest domyślną właściwością `MSForms.TextBox`).

Listing 3.14. *Właściwości domyślne*

```
' Sposób właściwy.
txtUsername.Text = "moje nazwisko"

' Sposób niewłaściwy.
txtUsername = "moje nazwisko"
```

Unikanie niejawnego używania właściwości domyślnych sprawia, że kod jest czytelniejszy i chroni przed błędami, jakie mogą wystąpić, jeżeli właściwości domyślne zostaną zmienione w przyszłych wersjach Excela lub VBA.

Weryfikacja argumentów przed ich użyciem w procedurach

Jeżeli procedura akceptuje tylko argumenty wejściowe o pewnych określonych właściwościach, np. wartościach z określonego przedziału, sprawdzaj, czy te warunki są spełnione przed próbą użycia tych argumentów w procedurze. Chodzi o to, aby wychwycić błędne dane wejściowe tak wcześnie, jak to tylko możliwe, aby wygenerować zrozumiały komunikat o błędzie i uprościć debugowanie.

Gdziekolwiek to możliwe, twórz narzędzia do weryfikowania zachowania procedur. Takie narzędzia są nakładaną na procedury „uprzężą”, która pozwala na ich wielokrotne wywoływania i testowanie z różnymi zestawami argumentów oraz sprawdzanie prawdziwości uzyskiwanych wyników. Budowanie takich „uprzęży” omówiliśmy w rozdziale 16.

Używaj liczników chroniących przed nieskończonymi pętlami

Twórz pętle z automatycznymi zabezpieczeniami przed użyciem warunków powodujących nieskończone działanie pętli. Jednym z najpowszechniejszych błędów spotykanych w pętlach `Do...While` lub `While...Wend` jest tworzenie sytuacji, w których warunki sterujące pętlą nigdy nie zostaną spełnione. Wówczas pętla może działać w nieskończoność (albo do czasu, gdy będziesz miał szczęście przerwać działanie aplikacji przez naciśnięcie kombinacji klawiszy *Ctrl+Break*, lub — gdy tego szczęścia zabraknie — przez jej zamknięcie za pomocą Menadżera Zadań systemu Windows). Zawsze dołączaj licznik, który automatycznie przerwie działanie, gdy liczba przebiegów pętli przekroczy największą wartość, jaką można uznać za możliwą do osiągnięcia w praktyce. W listingu 3.15 pokazujemy pętlę `Do..While` ze strukturą zabezpieczającą przed nieskończonym działaniem.

Listing 3.15. *Licznik zabezpieczający przed nieskończonym działaniem pętli*

```
Dim bKontynuujPetle As Boolean
Dim lLicz As Long

bKontynuujPetle = True
lLicz = 1

Do

    ' Kod umieszczony w tym miejscu powinien
    ' nadać zmiennej bKontynuujPetle wartość False,
    ' gdy pętla wykona swoje zadanie.

    ' Licznik zabezpieczający powoduje bezwarunkowe
    ' wyjście z pętli po wykonaniu 10000 iteracji.
    lLicz = lLicz + 1
    If lLicz > 10000 Then Exit Do

Loop While bKontynuujPetle
```

Jedynym celem umieszczenia zmiennej `lLicz` wewnątrz pętli jest wymuszenie zakończenia jej biegu, jeżeli kod w jej wnętrzu nie zdoła nadać zmiennej sterującej wartości kończącej iteracje zanim ich liczba osiągnie 10000 (stosowna liczba graniczna może być różna i zależna od sytuacji). Ten typ konstrukcji nie jest zbytnim obciążeniem dla pętli, jeżeli jednak wydajność jest sprawą kluczową, używaj licznika bezpieczeństwa do czasu uzyskania pewności, że kod wewnątrz pętli działa właściwie, po czym usuń dodatkowe instrukcje lub zamień je w komentarz.

Wcześnie i często używaj polecenia Debug/Compile

Nigdy nie pozwól na to, aby Twój kod pobłądził więcej niż o kilka zmian od czasu bezbłędnego wykonania polecenia *Debug/Compile*. Odchodzenie od tej zasady grozi długimi i nieefektywnymi sesjami debugowania.

Odwołując się do obiektów arkuszy, używaj nazw kodowych

Do arkuszy roboczych i arkuszy wykresów odwołuj się zawsze przez nazwy kodowe (`CodeNames`). Uzależnianie odwołań od nazw na zakładkach arkuszy jest ryzykowne, ponieważ Ty sam lub inni użytkownicy mogą te nazwy zmienić, niszcząc w ten sposób kod korzystający z takich odwołań.

Weryfikuj typy danych obiektów wybieranych

Jeżeli tworzona procedura ma operować na obiektach określonego typu wybieranych przez użytkownika, zawsze sprawdzaj typ wybranego obiektu za pomocą funkcji `TypeName` lub konstrukcji `If TypeOf...Is`. Jeżeli np. musisz działać na zakresach wybieranych przez użytkownika, zawsze przed pójściem dalej sprawdzaj, czy wybierane obiekty są rzeczywiście typu `Range`. Zostało to pokazane na listingu 3.16.

Listing 3.16. *Sprawdzanie, czy wybrany obiekt jest właściwego typu*

```
' Kod został zaprojektowany do działania na obiektach Range.
If TypeOf Selection Is Excel.Range Then
    ' OK, to jest obiekt Range.
    ' Kontynuuj działanie.
Else
    ' Błąd, to nie jest obiekt Range.
    MsgBox "Proszę wybrać zakres komórek", vbCritical, "Błąd!"
End If
```

Nadzorowanie zmian

Nadzorowanie zmian, zwane także kontrolą wersji, na najbardziej elementarnym poziomie polega na stosowaniu dwóch praktyk: zachowywaniu zestawu poprzednich wersji (dzięki czemu możesz zawsze powrócić do wersji sprzed wprowadzenia zmian i błędów programistycznych) oraz dokumentowaniu wszystkich zmian dokonywanych w aplikacji z upływem czasu.

Zachowywanie wersji

Gdy najbardziej doświadczeni programiści rozmawiają o kontroli wersji, myślą o używaniu dedykowanego oprogramowania w rodzaju Microsoft Visual Source Safe. Jednakże oprogramowanie tego typu jest drogie, trudne do opanowania i źle integruje się z aplikacjami budowanymi w Excelu. Dzieje się tak dlatego, że Excel nie zapisuje w sposób naturalny modułów w oddzielnych plikach tekstowych. Metoda kontroli wersji, jaką tu zalecamy, jest szybka, prosta, nie wymaga żadnego specjalnego oprogramowania, a przynosi te same zasadnicze korzyści, jak metody tradycyjne.

Najważniejszym celem stosowania systemu kontroli wersji jest zapewnienie możliwości odzyskania wcześniejszej wersji projektu po napotkaniu znaczących problemów w wersji bieżącej. Jeżeli po wprowadzeniu znaczących modyfikacji kodu rzeczy poszły w złą stronę i nagle pozostałeś z poważnie uszkodzonym plikiem, Twoja sytuacja może być bardzo trudna, jeśli nie masz kopii poprzedniej wersji, która pozwoliłaby na odzyskanie działającej aplikacji.

Prosty rodzaj systemu kontroli wersji, pozwalający na uwolnienie się od kłopotów tego rodzaju, może zostać wdrożony w sposób następujący. Po pierwsze, w folderze, którego używasz do zapisywania wersji bieżącej, utwórz folder o nazwie *KopieZapasowe* (Backup). Zawsze, gdy zamierzasz wprowadzać znaczące rozszerzenia lub modyfikacje projektu, a przynajmniej raz dziennie używaj programu kompresującego, np. WinZipa, do skompresowania wszystkich plików z folderu roboczego do pliku z nazwą: *Kopia_RRRRMMDDGG.zip*, gdzie R oznacza rok, M — miesiąc, D — dzień, a G — godzinę. Taki format nazw zapewni ich niepowtarzalność i pozwoli na łatwe sortowanie w oknie Eksploratora Windows. Utworzony plik przesuń do podfolderu *KopieZapasowe* i wróć do pracy.

Po napotkaniu problemu będziesz mógł wrócić do najnowszej wersji kopii. Oczywiście wymaga to poświęcenia nieco czasu, ale jeżeli będziesz pilnie wykonywać kopie, zminimalizujesz straty. Za każdym razem, gdy będziesz pewien, że masz już w pełni przetestowaną nową wersję projektu, będziesz mógł wykasować z folderu *KopieZapasowe* większość zapisanych tam plików. Warto jednak pozostawiać przynajmniej pojedyncze kopie z każdego tygodnia pracy przez cały okres życia projektu.

Dokumentowanie zmian za pomocą komentarzy

Gdy utrzymujesz kod, dokonujesz znaczących zmian w logice procedur, należy również dodać krótką notatkę z opisem zmiany, datą i nazwiskiem dokonującego zmiany, umieszczoną w komentarzu na poziomie procedury, co można zobaczyć w listingu 3.3. Wszystkie nietrywialne modyfikacje kodu powinny być opatrzone wewnętrznymi komentarzami, zawierającymi datę dokonania zmiany, nazwisko osoby, która zmianę wprowadziła, zwłaszcza gdy nad projektem pracuje grupa.

Wnioski

Niezależnie od tego, czy zastosujesz konwencję nazw zaproponowaną w tej książce, czy przygotujesz własną, stosuj ją konsekwentnie w całej aplikacji i nie zmieniaj z upływem czasu. Dzięki temu kod będzie samoudokumentowany i łatwy do zrozumienia. Poszczególne warstwy logiczne aplikacji koduj jako byty niezależne. Dzięki temu zmiany w jednej warstwie nie będą zmuszały do przebudowywania znacznej części aplikacji. Suto komentuj kod na wszystkich poziomach. Dzięki temu w przyszłości będzie łatwiej zrozumieć cel jakiejś sekcji programu bez żmudnego odczytywania szczegółów kodu. Postępowanie zgodne z dobrymi praktykami opisanymi w tym rozdziale pozwoli Ci na budowanie aplikacji odpornych, zrozumiałych i łatwych do utrzymania.