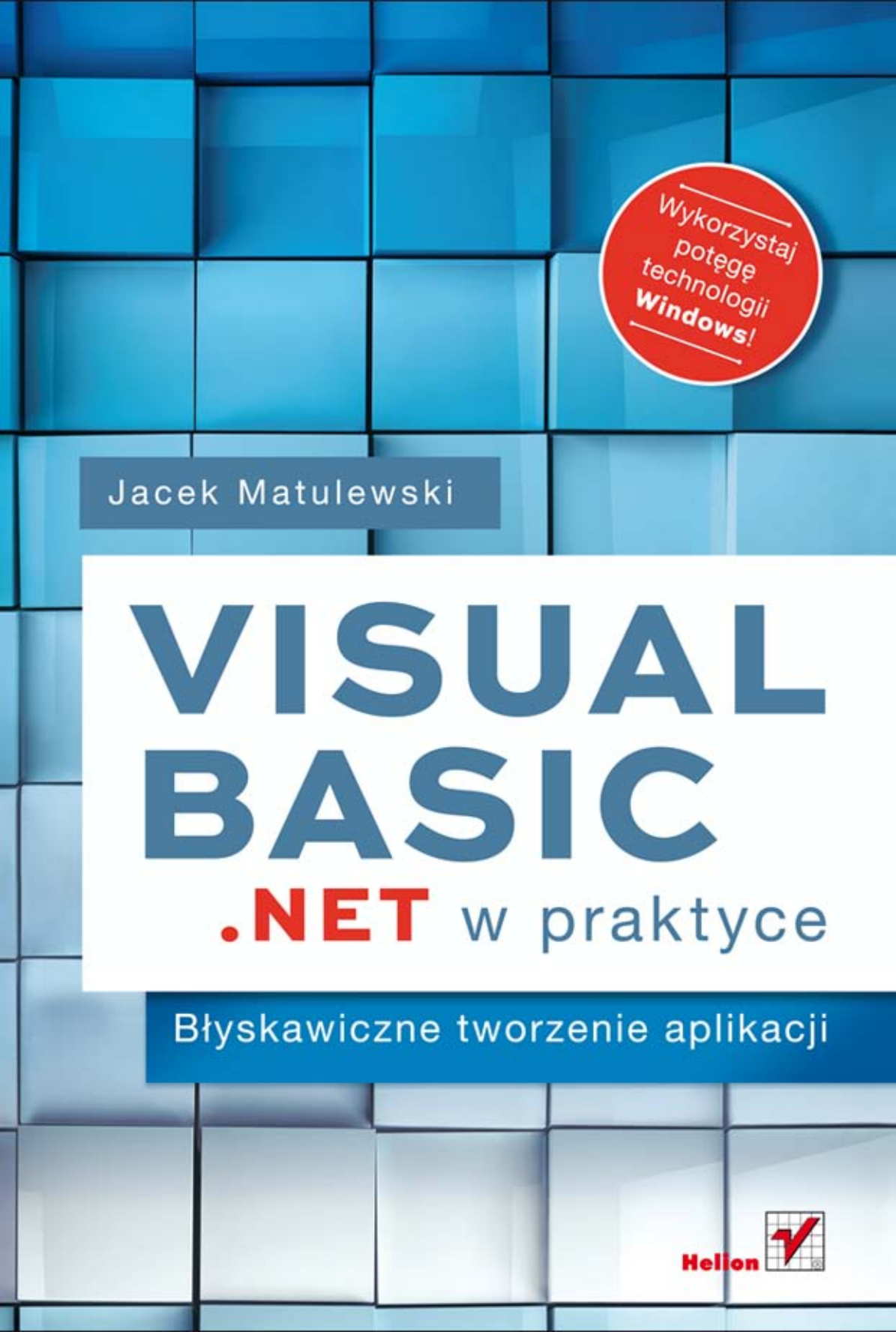

Wykorzystaj potęgę technologii Windows!
Jacek Matulewski
VISUAL BASIC
.NET w praktyce
Błyskawiczne tworzenie aplikacji
Helion

Redaktor prowadzący: Ewelina Burska
Projekt okładki: Magdalena Stasik
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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)

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

Dodatkowe materiały do książki są dostępne pod adresem:
ftp://ftp.helion.pl/przyklady/vbnepr.zip

ISBN: 978-83-246-4898-6

Printed in Poland.

Spis treści

Rozdział 2.

Debugowanie kodu

Skąd biorą się błędy i jak ich unikać?

Za błędy w kodzie jest (niemal) wyłącznie odpowiedzialny programista. Ale to oczywiste stwierdzenie niewiele nam pomaga w ich unikaniu. Zresztą błędów w ogóle nie można uniknąć. Ich powstawanie jest nierozerwalnie związane z samym procesem tworzenia kodu. Co więcej, liczba błędów w kodzie zależy proporcjonalnie od liczby linii kodu, a współczynnik proporcjonalności jest powiązany ze stopniem doświadczenia i niewyspania programisty. Prosty wniosek jest taki, że pisany przez nas kod powinien mieć jak najmniejszą liczbę linii. Do tego celu trzeba dążyć za pomocą wszelkich możliwych sposobów i sztuczek. Przede wszystkim należy unikać powtarzania kodu. Lepiej przygotować jedną metodę, którą możemy porządnie przetestować, niż powielać jej fragmenty w kilku miejscach, co uniemożliwia ich kontrolę w przypadku jakichkolwiek modyfikacji. Zatem lepiej wywołać porządnie napisaną metodę, zamiast korzystać z „kopiuj i wklej". Poza tym, jeżeli możemy użyć gotowego przetestowanego kodu (np. kontrolki), to warto z tego skorzystać. W tworzeniu kodu może nas też wyręczyć środowisko programistyczne. W przypadku aplikacji Windows Forms dzieje się tak zresztą już w momencie, gdy bawiąc się myszą, budujemy interfejs aplikacji.

Z pozornie bezużytecznego faktu, że błędów w kodzie nie da się uniknąć, co dobitnie wyraża prawo Murphy'ego: „Nie ma aplikacji bez błędów", można wyciągnąć jeszcze jeden wniosek: otóż, skoro występowanie błędów jest niemal pewne, trzeba się na nie przygotować. Nie można zakładać, że błędy będą zdarzać się na tyle rzadko, iż prawdopodobieństwo ich wystąpienia będzie tak małe, że nie będą dla użytkownika uciążliwe. Inne prawo Murphy'ego mówi bowiem: „Jeżeli błąd może wystąpić, wystąpi na pewno, i to w takim momencie, gdy użytkownikowi dopiecze to najbardziej" (np. gdy kończąc pracę nad dokumentem, będzie chciał zapisać go do pliku). Jak wspomniałem, jest zatem konieczne przygotowanie programu na możliwość wystąpienia błędów. Korzystajmy wobec tego z obsługi wyjątków w każdej budzącej wątpliwości sytuacji, nawet tam, gdzie — jak się nam wydaje — usunęliśmy już wszystkie usterki. Przemyślana reakcja na wyjątki może pozwolić na zminimalizowanie skutków wystąpienia błędu, czasem nawet do tego stopnia, że użytkownik nie musi się dowiedzieć, iż coś złego się wydarzyło.

Teoria Murphy'ego wypowiada się także na temat najmniej lubianej czynności programistów, a więc na temat testowania i tropienia błędów logicznych z kodu. Jak już wiemy, nie ma aplikacji bezbłędnych, liczbę błędów można jedynie redukować. Należy jednak zdawać sobie sprawę, że tzw. znalezienie ostatniego błędu to idea regulatywna debugowania. Należy w nią wierzyć, ale z góry wiadomo, że jest nieosiągalna. Warto też uświadomić sobie zaskakującą, ale w świetle teorii Murphy'ego oczywistą prawdę: poprawiając błędy w kodzie, tworzymy nowe.

Kontrolowane uruchamianie aplikacji

Na szczęście, w procesie usuwania błędów (debugowania) kodu Visual Basic dla platformy .NET mamy potężnego sprzymierzeńca. Jest nim środowisko programistyczne Visual Studio z wbudowanym debuggerem, wskazującym linie kodu, które nie podobają się kompilatorowi, pozwalającym na kontrolę uruchamianego kodu, a nawet na śledzenie jego wykonywania linia po linii. Przyjrzyjmy się bliżej kilku jego możliwościom.

Program z błędem — pole do popisu dla debuggera

Zacznijmy od przygotowania aplikacji, której metody zawierają błąd powodujący wystąpienie wyjątku. W tym celu:

1. Utwórz nowy projekt typu *Windows Forms Application* o nazwie `Debugowanie`.
2. W widoku projektowania umieść na formie przycisk `Button`.
3. Kliknij dwukrotnie ów przycisk, aby utworzyć domyślną metodę zdarzeniową (w tym przypadku do zdarzenia `Button1.Click`). Zostaniesz automatycznie przeniesiony do edytora kodu.
4. Zanim uzupełnisz zawartość metody zdarzeniowej, utwórz definicję metody `Kwadrat` (umieść ją np. bezpośrednio przed utworzonym automatycznie „szkieletem" metody `Button1_Click`, listing 2.1).

Listing 2.1. *Metoda, w której ukryliśmy perfidny błąd*

```
Public Class Form1

    Private Function Kwadrat(argument As Integer) As Integer
        Dim wartosc As Integer
        wartosc = argument * argument
        Return wartosc
    End Function

    Private Sub Button1_Click(sender As System.Object, e As System.EventArgs)
    ↪Handles Button1.Click

    End Sub
End Class
```

5. Teraz przejdź do metody `Button1_Click` i wpisz do niej polecenia wyróżnione w listingu 2.2.

Listing 2.2. *Z pozoru wszystko jest w porządku…*

```
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles
↪Button1.Click
    Dim x As Integer = 1234
    Dim y As Integer = Kwadrat(x)
    y = Kwadrat(y)
    Dim sy As String = y.ToString()
    MessageBox.Show("Wynik: " & sy)
End Sub
```

Oczywiście, obie metody można by „skompresować" do jednej lub dwóch linii, ale właśnie taka forma ułatwi naukę debugowania.

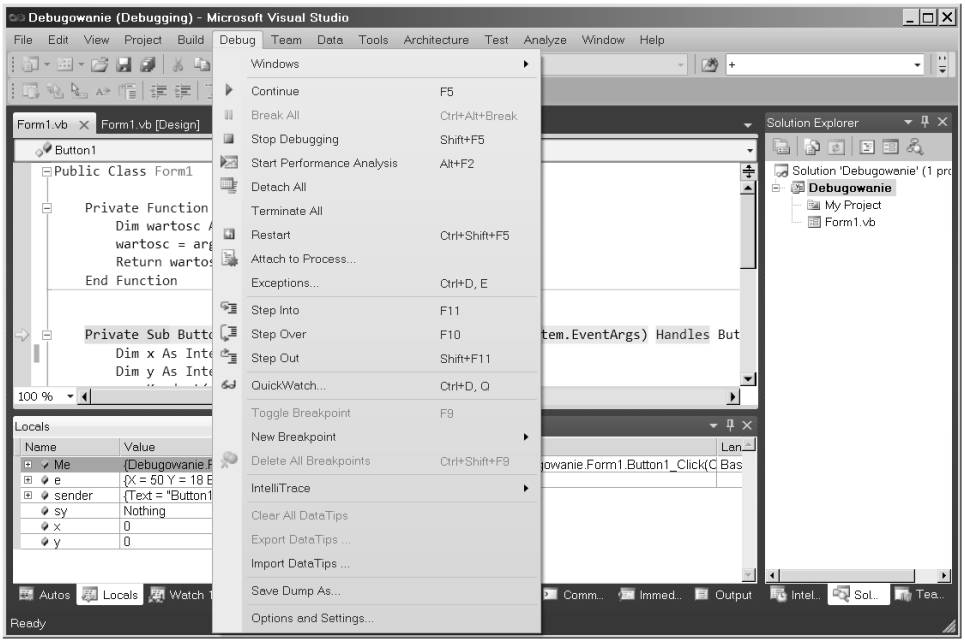
Uruchomimy najpierw aplikację (klawisz *F5*), żeby przekonać się, że nie działa prawidłowo. Po kliknięciu przycisku zamiast komunikatu z wynikiem zobaczymy komunikat o wyjątku z informacją, że nastąpiło przekroczenie zakresu zmiennej (*Arithmetic operation resulted in an overflow*). Miejsce wystąpienia wyjątku wskaże nam środowisko Visual Studio — program zatrzyma się na mnożeniu w metodzie `Kwadrat`. Nie jest jednak oczywiste, które wywołanie tej metody (a wywoływana jest dwa razy) prowadzi do błędu. Mamy zatem do czynienia z błędem logicznym ukrytym gdzieś w naszym kodzie. I to właśnie jego tropienie będzie motywem przewodnim większej części tego rozdziału.

Śledzenie wykonywania programu krok po kroku (klawisze F10 i F11)

Naciśnijmy klawisz *F10* lub *F11* (wszystkie klawisze skrótów wykorzystywane podczas debugowania zebrane zostały w tabeli 2.1). Jeżeli włączona jest wspomniana w poprzednim rozdziale platforma aplikacji (ang. *application framework*), to aplikacja uruchomi się normalnie. Dopiero gdy klikniemy przycisk na formie, jej działanie zatrzyma się i zobaczymy zaznaczoną na żółto sygnaturę metody zdarzeniowej związanej z tym zdarzeniem (rysunek 2.1). Inaczej jest, gdy platforma aplikacji jest wyłączona, a my zdefiniowaliśmy własną metodę `Main`. Wówczas po naciśnięciu *F10* lub *F11* na żółto zaznaczona zostanie jej sygnatura, a środowisko będzie czekało na dalsze polecenia. Każde naciśnięcie klawisza *F10* powoduje wówczas wykonanie jednej linii kodu (tej zaznaczonej na żółto), bez względu na to, czy jest to inicjacja zmiennej, czy wywołanie metody. Czynność taka nazywa się *Step Over* (menu *Debug*), czyli „krok nad". Nazwa bierze się z tego, że jeżeli w wykonywanej linii znajduje się wywołanie metody, jest ona wykonywana w całości. Natomiast *F11* powoduje wykonanie „kroku w głąb" (ang. *step into*), co w przypadku wywołania metody oznacza, że zostaniemy przeniesieni do pierwszej linii jej definicji i tam będziemy kontynuować śledzenie działania aplikacji. W przypadku gdy ta metoda zawiera wywołanie kolejnej metody, klawisze *F10* i *F11* pozwolą zdecydować, czy chcemy ją wykonać w całości, czy przeanalizować linia po linii. Jeżeli zorientujemy się, że zeszliśmy zbyt głęboko — możemy nacisnąć *Shift+F11*, aby wykonać pozostałą część metody i ją opuścić (ang. *step out* — wyjście z).

Tabela 2.1. *Związane z debugowaniem klawisze skrótów środowiska Visual Basic*

Funkcja	Klawisz skrótu
Uruchomienie z debugowaniem	*F5*
Uruchomienie bez debugowania	*Ctrl+F5*
Uruchomienie i zatrzymanie w linii, w której jest kursor	*Ctrl+F10*
Krok do następnej linii kodu (*Step over*)	*F10*
Krok z wejściem w głąb metody (*Step into*)	*F11*
Krok z wyjściem z metody (*Step out*)	*Shift+F11*
Ustawienie *breakpointu* (funkcja edytora)	*F9*
Zakończenie debugowania (zakończenie działania aplikacji)	*Shift+F5*



Rysunek 2.1. *Na rysunku rozwinięte jest menu Debug, zawierające instrukcje sterujące procesem kontrolowanego uruchamiania aplikacji*

Run to Cursor (klawisze Ctrl+F10)

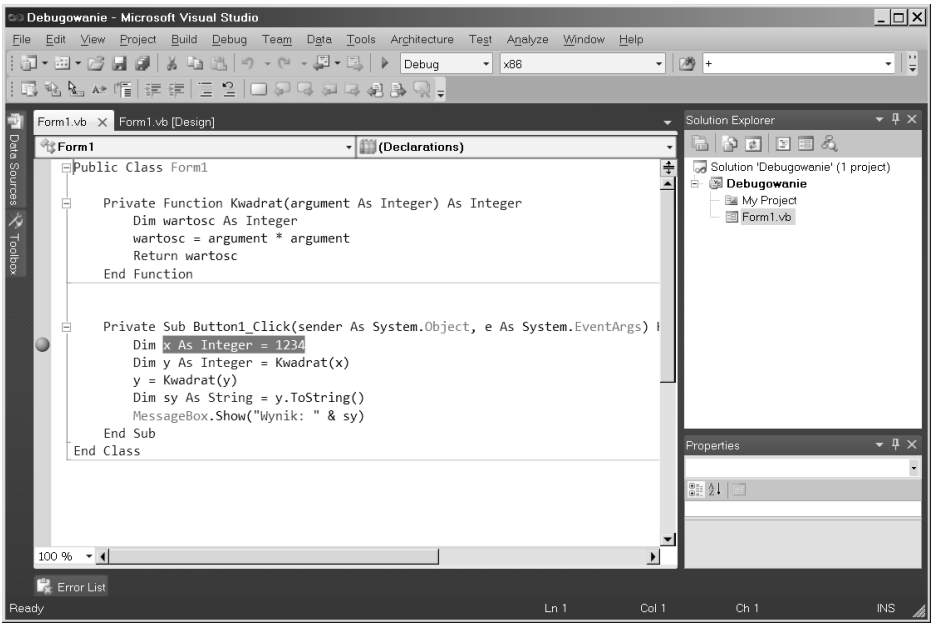
W przypadku gdy platforma aplikacji jest wyłączona i mamy zdefiniowaną metodę `Main`, korzystanie z klawiszy *F10* i *F11* może nie być zbyt wygodne. Dla przykładu, jeżeli interesuje nas tylko kod metody zdarzeniowej `Button1_Click`, to przechodzenie krok po kroku przez zawartość metody `Main` jest całkowicie zbędne. Wówczas wygodniej jest posłużyć się kombinacją klawiszy *Ctrl+F10*. Podobnie jak klawisz *F5*, uruchamia ona aplikację, ale zatrzymuje ją w momencie, gdy wykonana ma być instrukcja z linii, w której znajduje się kursor edytora. Ta kombinacja klawiszy działa także wtedy, gdy aplikacja jest już uruchomiona w trybie debugowania. Po zatrzymaniu

działania aplikacji znowu możemy korzystać z klawiszy *F10* i *F11* do śledzenia działania metod krok po kroku.

> Aby natychmiast przerwać debugowanie programu i powrócić do normalnego trybu edycji Visual Studio, należy nacisnąć klawisze *Shift+F5*.

Breakpoint (klawisz F9)

Gdy przewidujemy, że będziemy wielokrotnie kontrolować wykonywanie pewnych poleceń, np. z metody `Button1_Click`, możemy w jej pierwszej linii ustawić tzw. *breakpoint*. W tym celu przechodzimy do wybranej linii w edytorze kodu i naciskamy klawisz *F9*. Linia zostanie zaznaczona bordowym kolorem oraz czerwoną kropką na lewym marginesie (rysunek 2.2). Po uruchomieniu programu w trybie debugowania jego działanie zostanie zatrzymane, gdy wątek dotrze do tej linii. Możemy wówczas obejrzeć wartości zmiennych (o tym za chwilę), przejść do śledzenia kodu (klawisze *F10* i *F11*) lub nacisnąć klawisz *F5*, aby wznowić jego działanie w normalnym trybie debugowania. Gdy jednak wątek znowu dotrze do linii z ustawionym *breakpointem*, działanie programu jeszcze raz zostanie wstrzymane. Aby anulować *breakpoint*, należy ustawić kursor w odpowiedniej linii i jeszcze raz nacisnąć *F9* lub kliknąć widoczną na lewym marginesie czerwoną kropkę.

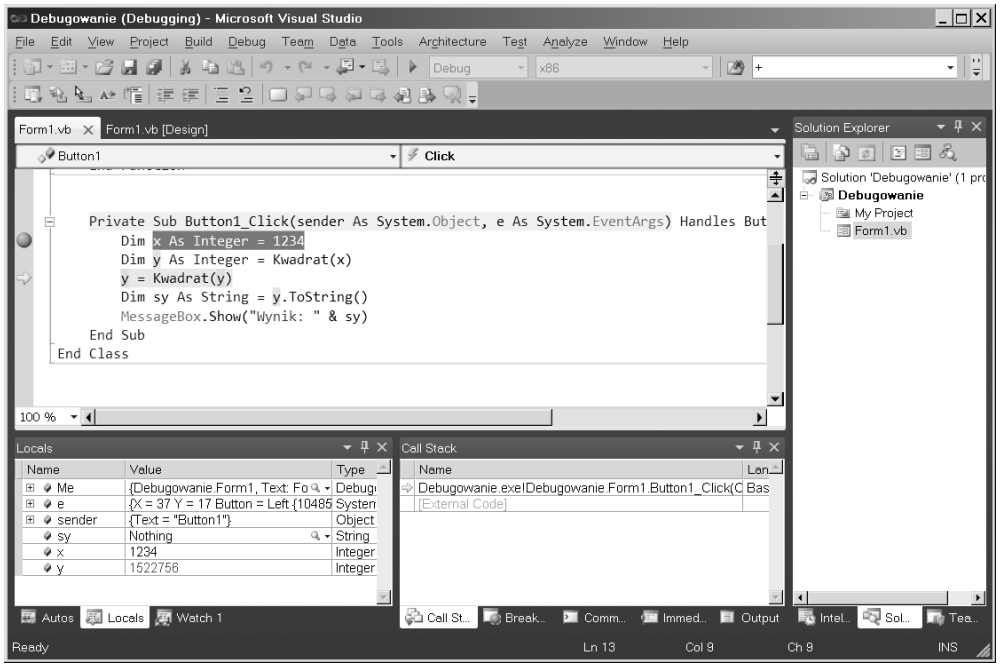


Rysunek 2.2. *Wyróżnienie linii kodu, w której ustawiony jest breakpoint*

Breakpoint bywa szczególnie pożyteczny przy śledzeniu wykonywania pętli. Jeżeli ustawimy go w interesującej nas linii wewnątrz pętli, każda jej iteracja zostanie przerwana, co pozwala np. na przyjrzenie się wartościom zmiennych.

Okna Locals i Watch

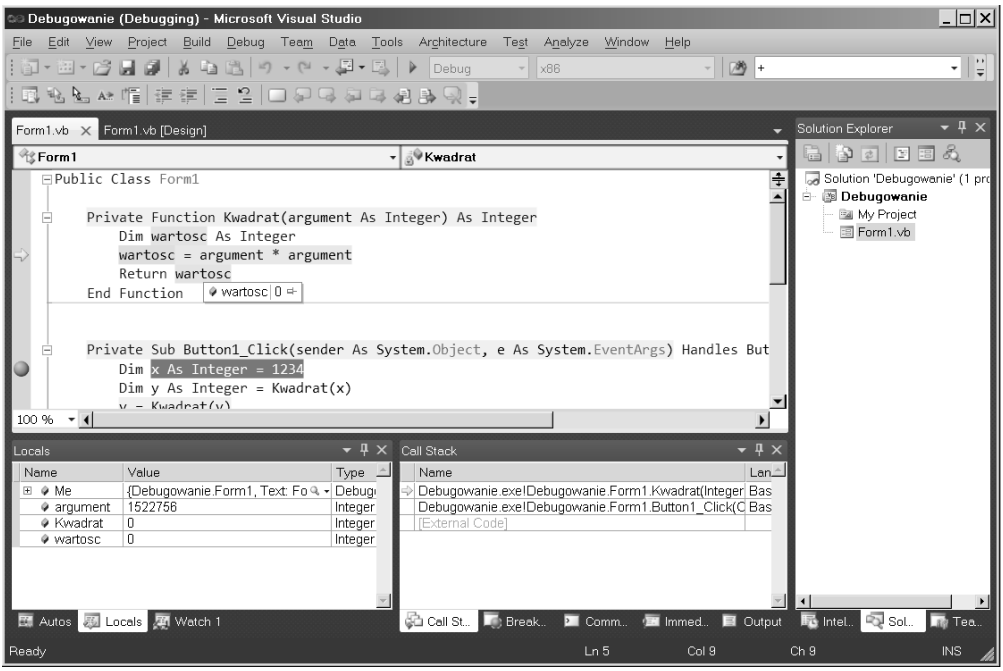
Możliwość obserwacji wartości zmiennych używanych w programie to bardzo ważne narzędzie debuggerów. Dzięki niemu możemy w każdej chwili sprawdzić, czy wartości zmiennych są zgodne z naszymi oczekiwaniami, a to pozwala ocenić, czy program działa prawidłowo. Wykonywanie aplikacji linia po linii z równoczesnym wpatrywaniem się w wartości zmiennych jest w praktyce najczęściej wykorzystywanym sposobem na odszukanie owego „ostatniego błędu". W Visual Studio służy do tego okno *Locals* (rysunek 2.3), które zawiera listę wszystkich pól zadeklarowanych wewnątrz obiektów i zmiennych lokalnych zadeklarowanych w aktualnie wykonywanej metodzie wraz z ich wartościami. Jeżeli zmienne są obiektami, możemy zobaczyć także ich pola składowe. Poza tym mamy do dyspozycji okno *Watch*, do którego możemy dodać nie tylko poszczególne pola i zmienne, ale również poprawne wyrażenia języka Visual Basic, np. `x*x`.



Rysunek 2.3. *Podglądanie wartości zmiennych w edytorze*


Uwaga

Wartości zmiennych można również zobaczyć, jeżeli w trybie debugowania w edytorze przytrzymamy przez chwilę kursor myszy nad zmienną widoczną w kodzie. Po chwili pojawi się okienko podpowiedzi zawierające wartość wskazanej w ten sposób zmiennej (rysunek 2.4). Należy jednak pamiętać, że pokazywana wartość dotyczy aktualnie wykonywanej linii, a nie linii wskazanej kursorem. W Visual Studio 2010 taką podejrzaną w oknie edytora wartość można przypiąć (ikona pinezki), przez co podczas pracy w trybie debugowania pozostaje stale widoczna na ekranie (rysunek 2.4). Możliwość podglądania wartości zmiennych w oknie edytora dotyczy także obiektów — zobaczymy wówczas wartości wszystkich dostępnych pól i własności.



Rysunek 2.4. *U dołu okna VS widoczne są dwa podokna: z lewej okno Locals, z prawej — Call Stack*

Przejdźmy w edytorze kodu (zakładka *Form1.vb*) do linii w metodzie `Button1_Click`, w której zdefiniowana jest zmienna `y`. Zaznaczmy tę zmienną i prawym przyciskiem myszy rozwińmy menu kontekstowe. Z niego wybierzmy polecenie *Add Watch* (dostępne tylko, gdy uruchomione jest debugowanie). Zmienna ta zostanie dodana do listy w oknie *Watch*, w której zobaczymy jej nazwę, wartość i typ. Wartość jest aktualizowana przy każdym naciśnięciu klawisza *F10* lub *F11*, co pozwala na śledzenie jej zmian w trakcie wykonywania metody. Możemy się przekonać, wykonując kolejne polecenia metody `Button1_Click` ponownymi naciśnięciami klawisza *F10*, że po pierwszym uruchomieniu metody `Kwadrat` zmienna `y` ma wartość równą `1522756`. Jest zatem poprawna. Niestety, przy drugim uruchomieniu metody `Kwadrat` pojawi się wyjątek. Aby sprawdzić, co jest przyczyną błędu, wchodzimy do jej „środka", używając klawisza *F11*. W metodzie `Kwadrat` nie ma zmiennej `y`, a jej wartość przejmuje zmienna `argument`. Dodajmy ją zatem do listy obserwowanych zmiennych, wpisując jej nazwę do pierwszej kolumny w podoknie *Watch*. Dodajmy także wyrażenie `argument*argument`. W trakcie drugiego przebiegu metody `Kwadrat` przy tym wyrażeniu pojawi się komunikat *Constant expression not representable in type „Integer"*, czyli „wartość nie może być przedstawiona w zmiennej typu »Integer«".

Informacja ta wyjaśnia przyczynę błędu — podczas mnożenia został przekroczony, i to znacznie, zakres możliwych wartości zmiennej `Integer` (1 522 756·1 522 756 = 2 318 785 835 536 > 2 147 483 647). Innymi słowy, 32 bity, jakie oferuje typ `Integer`, nie wystarczają do zapisania wyniku. Aby poprawić ów błąd, przynajmniej w pewnym zakresie wartości argumentów, należałoby użyć 64-bitowego typu `Long`. Ponadto, ponieważ kwadrat liczby całkowitej nie może być ujemny, warto pozbyć się także znaku

i użyć typu `ULong`. Kod pozbawiony błędu widoczny jest na listingu 2.3. Na razie tych zmian jednak nie wprowadzajmy — korzystając z obecności błędu, chciałbym zaprezentować mechanizm przechwytywania wyjątków.

Listing 2.3. *Poprawiony kod, którego na razie nie wstawiajmy do projektu*

```
Private Function Kwadrat(argument As Integer) As Long
    Dim wartosc As ULong
    wartosc = CULng(argument) * CULng(argument)
    Return wartosc
End Function

Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles
↪Button1.Click
    Dim x As Integer = 1234
    Dim y As ULong = Kwadrat(x)
    y = Kwadrat(y)
    Dim sy As String = y.ToString()
    MessageBox.Show("Wynik: " & sy)
End Sub
```

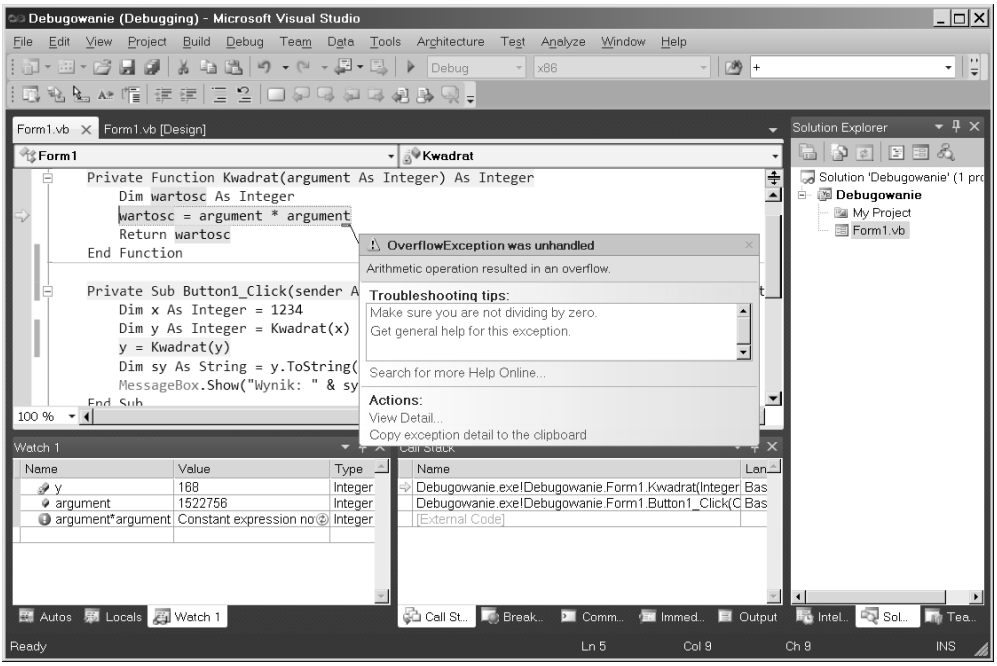

Uwaga

Na rysunku 2.4 z prawej strony na dole widoczne jest okno *Call Stack* (z ang. stos wywołań). Wymienione są w nim wszystkie metody, począwszy od metody `Button1_↪Click`, a skończywszy na metodzie, której polecenie jest obecnie wykonywane. W bardziej rozbudowanych programach pomaga to w zorientowaniu się, co się aktualnie dzieje w aplikacji, szczególnie po jej zatrzymaniu w wyniku działania *breakpointu*.

Stan wyjątkowy

Jeżeli nie wprowadziliśmy zmian z listingu 2.3 do projektu, w razie uruchomieniu projektu z debugowaniem (klawisz *F5*) po kliknięciu przycisku `Button1` pojawi się komunikat debuggera informujący o wyjątku (rysunek 2.5). Przyjrzyjmy się temu komunikatowi. Przede wszystkim na pasku tytułu widoczne jest ostrzeżenie *OverflowException was unhandled*, co oznacza, że wyjątek został zgłoszony przez fragment kodu, który nie był otoczony obsługą wyjątków. Nie znajdował się zatem w sekcji `try` konstrukcji `try..catch` ani w metodzie, która byłaby z tej sekcji wywołana. W efekcie w przypadku uruchomienia aplikacji poza środowiskiem Visual Studio wyjątek ten nie byłby w żaden sposób obsłużony przez aplikację i musiałaby się nim zająć sama platforma .NET, co raczej nie powinno mieć miejsca. W oknie komunikatu debuggera widoczna jest także treść przekazywanego przez wyjątek komunikatu oraz link do dokumentacji klasy wyjątku (w naszym przypadku klasy `OverflowException`). Po kliknięciu *View Detail...* można obejrzeć stan przesyłanego obiektu w widocznym na dole okna odpowiedniku okna *Locals*.

Po wystąpieniu wyjątku środowisko Visual Studio wstrzymuje działanie aplikacji na tej samej zasadzie jak *breakpoint*, tzn. zwykle możemy przywrócić jej działanie, np. za pomocą klawiszy *F5* lub *F10*. Możemy także zupełnie przerwać działanie aplikacji i przejść do poprawiania kodu, naciskając kombinację klawiszy *Shift+F5*.



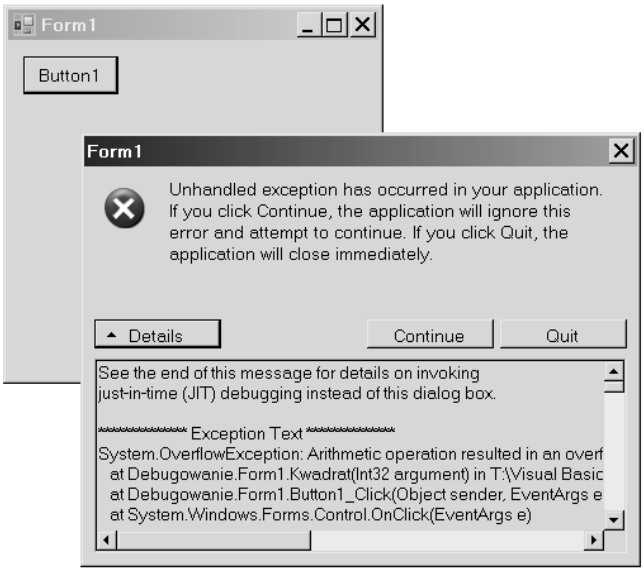
Rysunek 2.5. *Komunikat o wyjątku zgłoszony przez Visual Studio*

Jeżeli aplikacja zostanie uruchomiona poza debuggerem przez naciśnięcie kombinacji klawiszy *Ctrl+F5* lub samodzielne uruchomienie pliku *.exe* poza środowiskiem Visual Studio, nieobsłużony wyjątek jest przechwytywany przez platformę .NET. O takiej sytuacji użytkownik powiadamiany jest komunikatem widocznym na rysunku 2.6. Proszę zwrócić uwagę na to, że w przeciwieństwie do platformy Win32, w której analogiczny komunikat oznaczałby awaryjne zamknięcie aplikacji, tu istnieje możliwość kontynuowania jej działania (przycisk *Kontynuuj* widoczny na rysunku 2.6). Oczywiście, wyjątek pojawi się znowu, jeżeli klikniemy przycisk `Button1` na formie i uruchomiona zostanie metoda `Kwadrat`, ale aplikacja nie zawiesza się. Warto docenić tę własność platformy .NET — jeżeli nasza aplikacja byłaby edytorem, a błąd pojawiałby się np. podczas drukowania, platforma .NET, nie zamykając całej aplikacji w przypadku wystąpienia błędu, dałaby nam możliwość m.in. zachowania niezapisanego dokumentu na dysku.

Przechwytywanie wyjątków w konstrukcji try..catch

Teraz spróbujmy wykryć wystąpienie wyjątku z poziomu aplikacji. W tym celu w metodzie zdarzeniowej przycisku `Button1_Click` otoczymy wywołanie metody `Kwadrat` konstrukcją przechwytywania wyjątków.

1. Naciśnij *Shift+F5*, aby zakończyć działanie debuggera.
2. Do metody `Button1_Click` dodaj obsługę wyjątku zgodnie ze wzorem na listingu 2.4.



Rysunek 2.6. *Wyjątek nieobsłużony w aplikacji jest przechwytywany przez środowisko .NET (aplikacja uruchomiona jest bez debuggera)*

Listing 2.4. *Wszystkie dotychczasowe polecenia metody Button1_Click umieściliśmy w jednej sekcji Try*

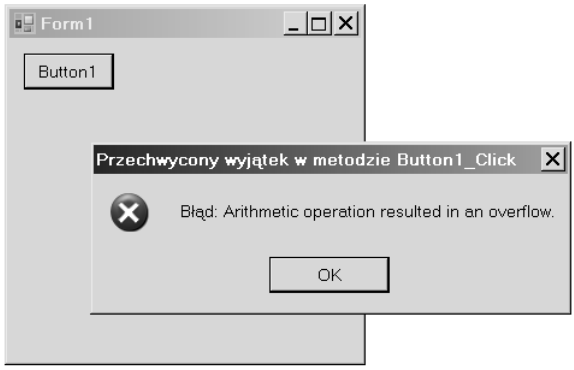
```
Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) Handles
↪Button1.Click
    Try
        Dim x As Integer = 1234
        Dim y As Integer = Kwadrat(x)
        y = Kwadrat(y)
        Dim sy As String = y.ToString()
        MessageBox.Show("Wynik: " & sy)
    Catch ex As Exception
        MessageBox.Show("Błąd: " & ex.Message,
                        "Przechwycony wyjątek w metodzie Button1_Click",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Error)
    End Try
End Sub
```

3. Skompiluj i uruchom aplikację w trybie debugowania (klawisz *F5*).

Teraz po uruchomieniu aplikacji i kliknięciu przycisku zamiast komunikatu debuggera lub komunikatu platformy .NET zobaczymy własny komunikat o błędzie (rysunek 2.7). Debugger nie zainterweniuje w przypadku obsłużonego wyjątku.

Wszystkie polecenia metody `Button1_Click` umieściliśmy w jednej sekcji `Try`. To ma sens, gdy każde wystąpienie błędu powoduje, że wykonywanie dalszej części metody pozbawione jest celu. Jeśli natomiast po pojawieniu się błędu chcielibyśmy podjąć jakąś akcję ratunkową i próbować kontynuować działanie metody, to każda budząca wątpliwości instrukcja powinna być otoczona oddzielną konstrukcją obsługi wyjątków.

Dodatkowe informacje na temat wyjątków można znaleźć w rozdziale 3.



Rysunek 2.7. *Komunikat wyświetlany użytkownikowi po przechwyceniu wyjątku*

Zgłaszanie wyjątków

Ostatnią rzeczą, o której chciałbym wspomnieć w kontekście wyjątków, jest ich zgłaszanie. Jeżeli w kodzie wykryjemy błąd, dla przykładu jeżeli sprawdzimy, że argument metody `Kwadrat` jest zbyt duży, to możemy samodzielnie zgłosić wyjątek. W takim przypadku należy użyć instrukcji `Throw` zgodnie ze wzorem widocznym na listingu 2.5. Kod ten używa dodatkowego pola (zmiennej zdefiniowanej w obrębie klasy) o nazwie `MaxArgumentKwadrat`, której definicja również widoczna jest na poniższym listingu.

Listing 2.5. *Wartość pierwiastka obliczana jest tylko raz, więc kontrola argumentu nie obciąża nadmiernie procesora*

```
Dim MaxArgumentKwadrat As Integer = Math.Sqrt(Integer.MaxValue)

Private Function Kwadrat(argument As Integer) As Integer
    Dim wartosc As Integer
    If argument < MaxArgumentKwadrat Then
        Throw New Exception("Argument metody kwadrat ma zbyt dużą wartość")
    End If
    wartosc = argument * argument
    Return wartosc
End Function.
```

Gdy teraz po uruchomieniu aplikacji klikniemy przycisk `Button1`, zamiast dotychczasowego komunikatu informującego o przepełnieniu zobaczymy nasz własny, bardziej konkretny komunikat.

Skorowidz

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

VISUAL BASIC to język programowania o bardzo długiej tradycji. Kiedy kilka lat temu przeniesiono go na platformę .NET, zyskał nowe możliwości rozwoju. Dziś chętnie używają go osoby, które potrzebują wygodnego narzędzia, pozwalającego szybko stworzyć aplikację do prezentacji i analizy konkretnych danych. Jasna, niezbyt skomplikowana składnia, doskonała biblioteka gotowych kontrolek i bezkolizyjna współpraca z bazą danych to najważniejsze atuty VB.NET.

Jeśli jesteś początkującym użytkownikiem Visual Basic na platformie .NET, trafiłeś w dziesiątkę. Tu znajdziesz podstawy języka oraz wszystkie funkcje potrzebne do swobodnego projektowania aplikacji — bez nadmiernego wchodzenia w szczegóły. Zorientujesz się, jak działa środowisko Visual Studio i jak za pomocą jego narzędzi tworzyć aplikacje VB.NET. Dowiesz się więcej o debugowaniu kodu, programowaniu zorientowanym obiektowo i używaniu kontrolek, zarówno tych z biblioteki Windows Forms, jak i zaprojektowanych osobiście. Odkryjesz, jak można połączyć aplikację z bazą danych i jak wykorzystać technologie systemu Windows. To wszystko, czego Ci trzeba!

- Środowisko Visual Studio i język Visual Basic
- Przegląd komponentów biblioteki Windows Forms
- Projektowanie zorientowane obiektowo i projektowanie kontrolek
- Zarządzane biblioteki DLL i mechanizm rozpoznawania typów
- Aplikacje konsolowe i informacje o systemie
- Wątki, równoległa pętla Parallel.For oraz programowanie asynchroniczne (Async/Await)
- Podstawy ADO.NET i SQL Server
- Wprowadzenie do zapytań LINQ na przykładzie kolekcji (LINQ to Objects)
- LINQ to DataSet i LINQ to SQL
- Odczytywanie i zapisywanie danych w plikach XML
- Rejestr systemu Windows, mechanizm PInvoke i komunikaty Windows

Wypróbuj **Visual Basic** i twórz dynamiczne rozwiązania!

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-246-4898-6

Cena 79,00 zł