Rozdział 2. Debugowanie kodu

Jacek Matulewski
Materiały dla Podyplomowego Studium Programowania i Zastosowania Komputerów, sekcja Projektowanie i tworzenie aplikacji dla platformy .NET (pod patronatem Microsoft)

Nie ma aplikacji bez błędów, ale liczbę błędów można redukować. Pomaga w tym środowisko programistyczne Visual Studio wraz z wbudowanym w nim debuggerem, wskazującym linie kodu, które nie podobają się kompilatorowi, pozwalającym na kontrolę uruchamianego kodu, jak również śledzenie jego wykonywania linia po linii. Przyjrzyjmy się bliżej kilku jego najczęściej wykorzystywanym możliwościom.

Program z błędem logicznym — pole do popisu dla debuggera

Zacznijmy od przygotowania aplikacji, która kompiluje się, a więc nie posiada błędów składniowych, ale której metody zawierają błąd powodujący jej błędne działanie.

1. Utwórz nowy projekt typu *Console Application* o nazwie *Debugowanie*.
2. W klasie `Program`, obok metody `Main` (na przykład nad nią) zdefiniuj metodę `Kwadrat` zgodnie ze wzorem z listingu 2.1.

Listing 2.1. W wyróżnionej metodzie ukryty jest błąd

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Debugowanie
{
    class Program
    {
        static private int Kwadrat(int argument)
        {
            int wartosc;
            wartosc = argument * argument;
            return wartosc;
        }
```

```
            static void Main(string[] args)
            {
            }
        }
    }
```

3. Teraz przejdź do metody `Main` i wpisz do niej polecenia wyróżnione w listingu 2.2.

Listing 2.2. Z pozoru wszystko jest w porządku…

```
static void Main(string[] args)
{
    int x = 1234;
    int y = Kwadrat(x);
    y = Kwadrat(y);
    string sy = y.ToString();
    Console.WriteLine(sy);
}
```

Oczywiście, obie metody można „spakować” do jednej lub dwóch linii, ale właśnie taka forma ułatwi nam naukę debugowania.

Uruchomimy najpierw aplikację (klawisz *Ctrl+F5*), żeby przekonać się, że nie działa prawidłowo. Po kliknięciu przycisku zobaczymy komunikat wyświetlający wynik: –496 504 304. Wynik jest ujemny, co musi budzić podejrzenia, bo podnoszenie do kwadratu liczby całkowitej nie powinno, oczywiście, zwracać wartości mniejszej od zera. Za pomocą kalkulatora możemy przekonać się ponadto, że nawet wartość bezwzględna wyniku nie jest prawdziwa, bo liczba 1234 podniesiona do czwartej potęgi to 2 318 785 835 536. Możemy zatem podejrzewać, że mamy do czynienia z błędem logicznym ukrytym gdzieś w naszym kodzie. I to właśnie jego tropienie będzie motywem przewodnim większej części tego rozdziału.

Kontrolowane uruchamianie aplikacji w Visual C#

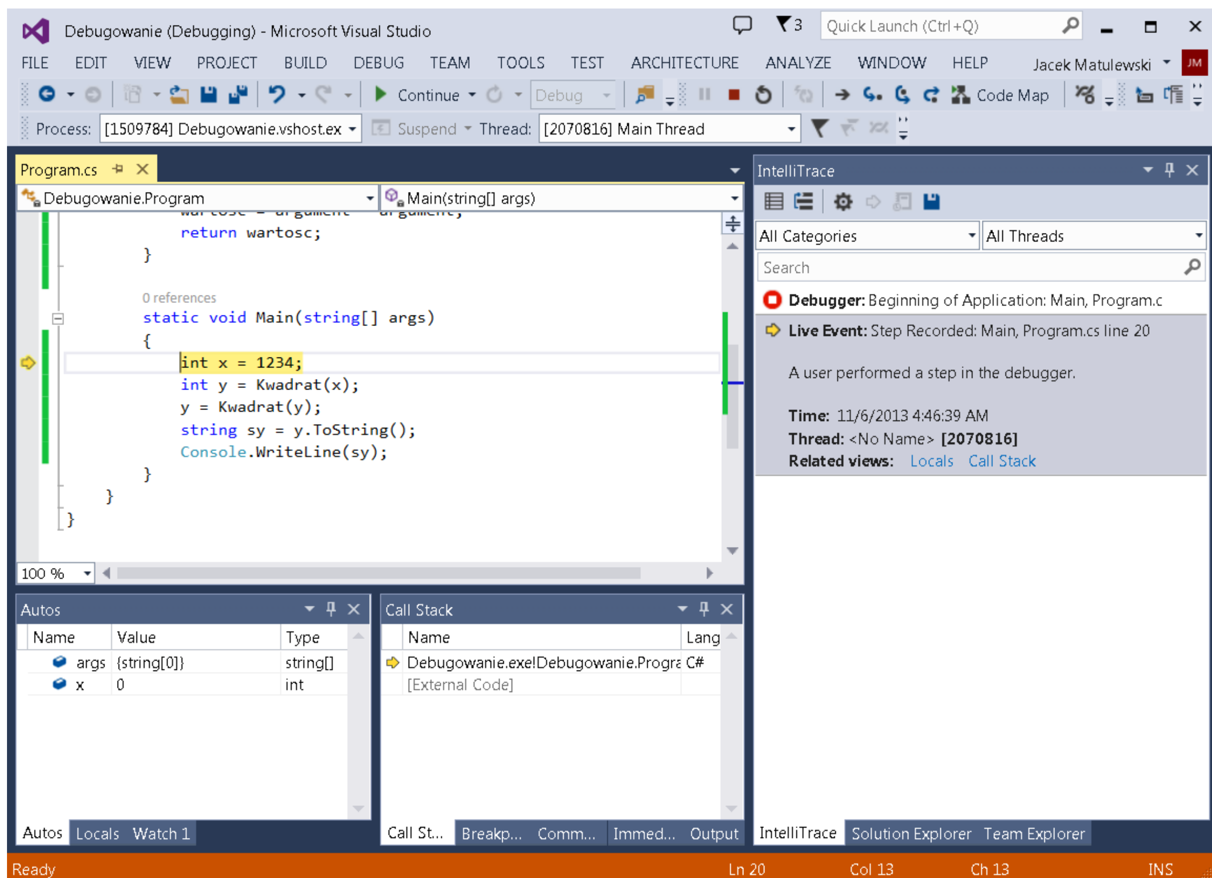
Śledzenie wykonywania programu krok po kroku (F10 i F11)

Naciśnijmy zatem klawisz *F10* lub *F11* (wszystkie klawisze skrótów wykorzystywane podczas debugowania zebrane zostały w tabeli 2.1), aby uruchomić aplikację i przejść do pierwszej linii metody `Main`. Każde naciśnięcie klawisza *F10* powoduje wykonanie jednej linii kodu, bez względu na to, czy jest to inicjacja zmiennej, czy wywołanie metody. Linia ta jest zaznaczona kolorem żółtym (rysunek 2.1). Czynność taka nazywa się *Step Over* (zob. menu *Debug*), czyli „krok nad”. Nazwa jest dobrze dobrana, bo jeżeli w wykonywanej linii znajduje się wywołanie metody, jest ona wykonywana w całości – nie wchodzimy do niej. 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 C#

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. Żółtym tłem zaznaczone jest bieżące polecenie

Naciskając kilka razy klawisz *F10*, przechodzimy do tej linii metody `Main`, w której znajduje się drugie wywołanie metody `Kwadrat`. Wtedy naciśnijmy *F11*. Wówczas przeniesiemy się do metody `Kwadrat` (żółta linia będzie znajdować się na otwierającym jej ciało nawiasie klamrowym). Naciskajmy *F10*, aby zobaczyć krok po kroku wykonywanie kolejnych poleceń. Zauważmy, że pominięta została linia zawierająca deklarację zmiennej `wartosc`, natomiast jednym z kroków była jej inicjacja. Naciskając *F10* prześledźmy wykonanie metody `Kwadrat`, następnie powrócimy do metody `Main` i tak dalej aż do zakończenia programu.

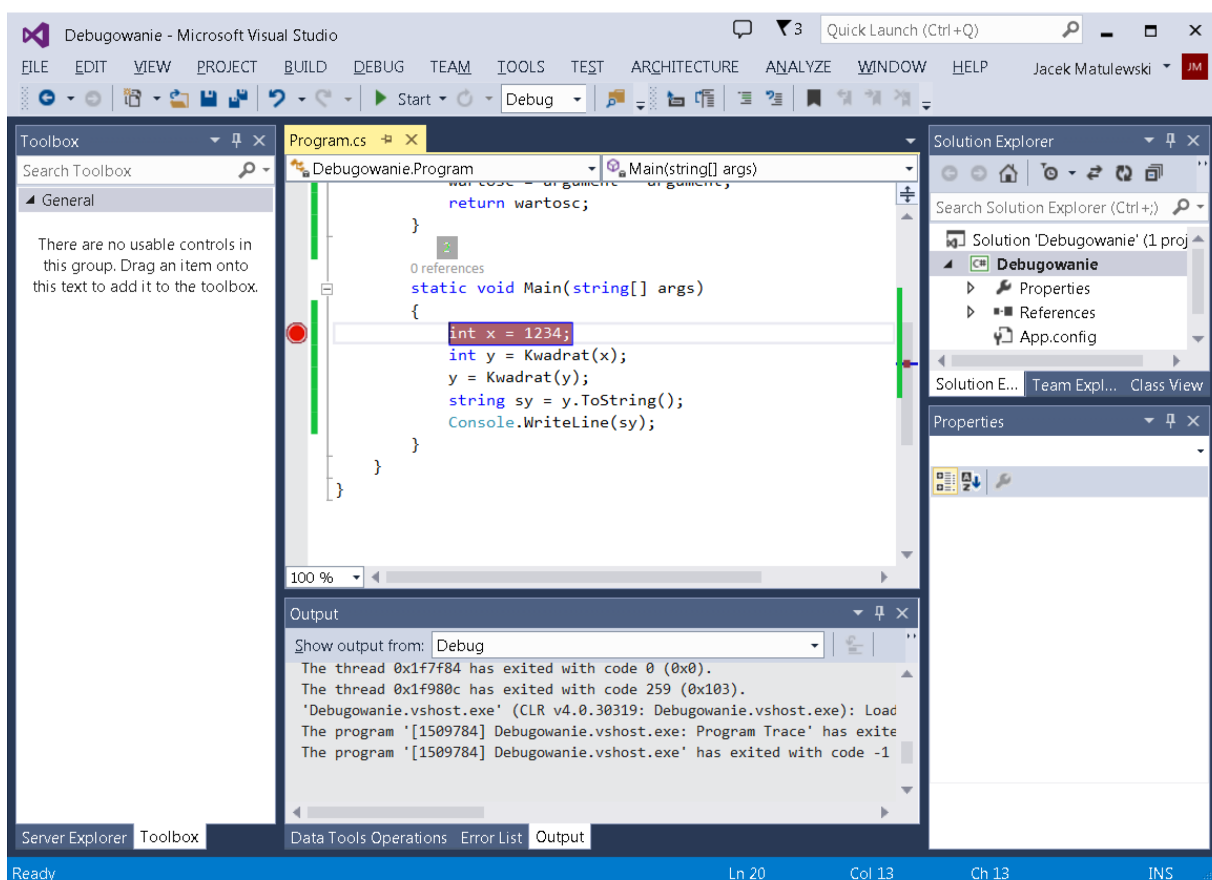
Run to Cursor (Ctrl+F10)

Prześledziliśmy krok po kroku działanie całego programu. Tak można postępować w takich prostych projektach, jak ten. W bardziej złożonych, debugowanie programu instrukcja po instrukcji byłoby czasochłonne. Szczególnie, że zwykle szukamy błędów w jakimś określonym fragmencie kodu. Jak w takim razie od razu dostać się do interesującej nas metody? Służy do tego kombinacja klawiszy *Ctrl+F10*. Uruchamia aplikację tak samo jak klawisz *F5*, ale zatrzymuje ją w momencie, w którym 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 – wykonywany jest kod aż do zaznaczonej kursorem linii. 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*. Pasek stanu środowiska powinien zmienić kolor z pomarańczowego na niebieski.

Breakpoint (F9)

Gdy przewidujemy, że będziemy wielokrotnie kontrolować wykonywanie pewnych poleceń, np. metody `Kwadrat`, możemy 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 linii, w której ustawiliśmy *breakpoint*. Możemy wówczas przejść do śledzenia kodu (*F10* i *F11*) lub nacisnąć klawisz *F5*, aby wznowić jego działanie w normalnym trybie debugowania. Gdy jednak wątek znowu dotrze do *breakpointu*, 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.



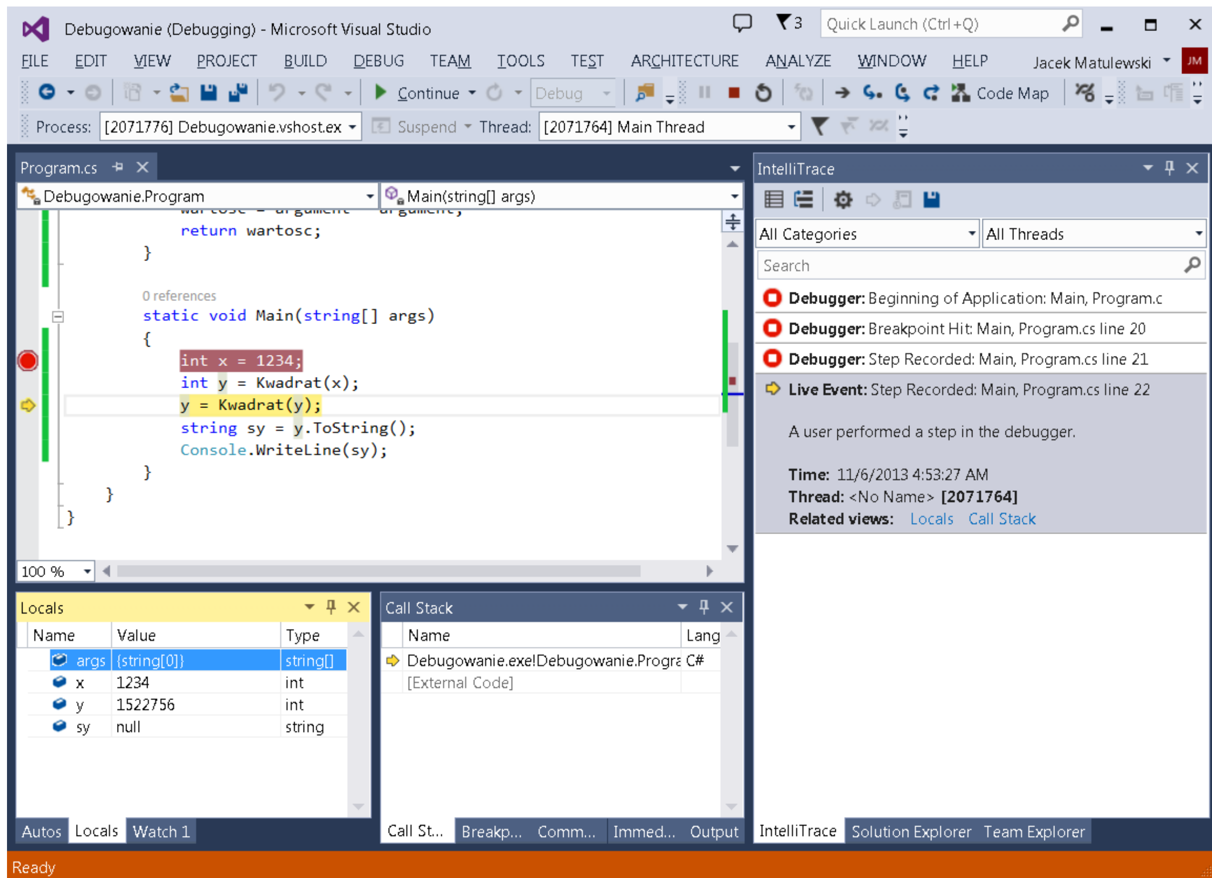
Rysunek 2.2. Wyróżnienie linii kodu, w której ustawiony jest breakpoint

Breakpoint bywa bardzo użyteczny przy śledzeniu wykonywania pętli. Jeżeli ustawimy go w interesującej nas linii wewnątrz pętli, każda jej iteracja zostanie przerwana i będziemy mieli możliwość np. przyjrzenia się wartościom zmiennych.

Okna Locals i Watch

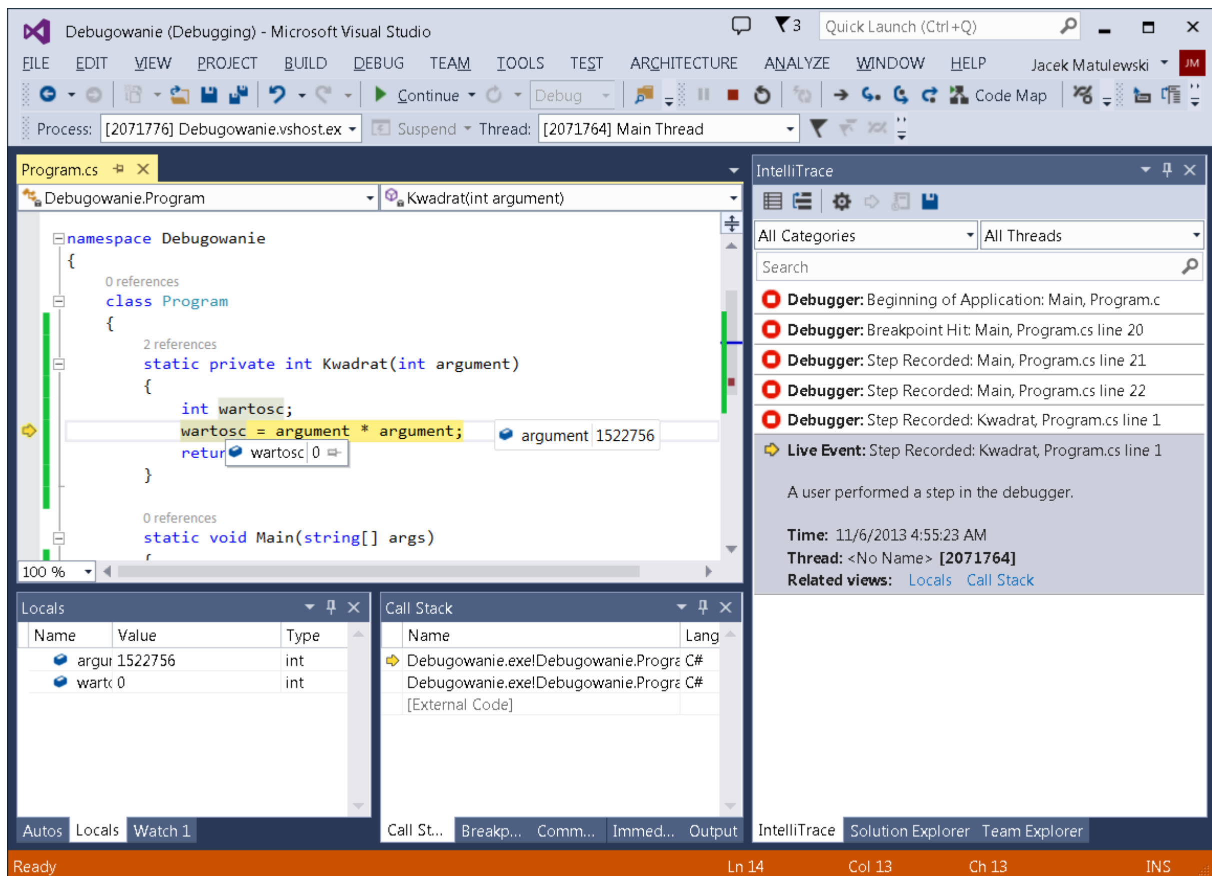
Możliwość obserwacji wartości zmiennych używanych w programie to bardzo ważne narzędzie debuggerów. Dzięki niemu możemy w każdej chwili sprawdzić, czy wartości zmiennych są zgodne z naszymi oczekiwaniami, a to pozwala ocenić, czy program działa prawidłowo. Wykonywanie aplikacji linia po linii z równoczesnym wpatrywaniem się w wartości zmiennych jest w praktyce najczęściej wykorzystywanym sposobem na odszukanie owego mistycznego „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 C#, np. `x*x`.



Rysunek 2.3. Podglądanie wartości zmiennych w edytorze

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. Okienko, w którym pokazywana jest wartość zmiennej można przypiąć do okna edytora (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 VC# widoczne są dwa podokna: z lewej okno Watch, z prawej — Call Stack

Przejdźmy w edytorze kodu do linii metody `Main`, w której zdefiniowana jest zmienna `y`, zaznaczmy tę zmienną i prawym przyciskiem myszy rozwińmy menu kontekstowe. Z niego wybierzmy polecenie *Add Watch*. Zmienna 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 kodu. Możemy się przekonać, wykonując ponownymi naciśnięciami klawisza *F10* kolejne polecenia metody `Main`, że po pierwszym uruchomieniu metody `Kwadrat` zmienna `y` uzyskuje wartość 1522756, czyli jej wartość jest jeszcze poprawna. Niestety, przy drugim uruchomieniu metody `Kwadrat` jej wartość staje się ujemna. Z tego wynika, że to wywołanie metody `Kwadrat`, a nie np. konwersja wyniku do łańcucha, jest przyczyną niepoprawnego wyniku, który widzimy w konsoli. Aby sprawdzić działanie metody `Kwadrat`, wchodzimy do jej „środka”, używając klawisza *F11* (najlepiej podczas jej drugiego wywołania). W tej metodzie 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`. Jego wartość okaże się nieprawidłowa.

Przyczyna błędu jest już chyba jasna — podczas mnożenia został przekroczony, i to znacznie, zakres możliwych wartości zmiennej `int` (1 522 756 ∗ 1 522 756 = 2 318 785 835 536 > 2 147 483 647). Innymi słowy, 32 bity, jakie oferuje typ `int`, nie wystarczają do zapisania wyniku, stąd jego zła interpretacja.

Czytelnik zastanawia się zapewne, czemu maszyna wirtualna nie wykryła przekroczenia zakresu w mnożeniu i nie zgłosiła błędu. Nie robi tego, bo to bardzo obniżyłoby wydajność wykonywanych przez nią operacji arytmetycznych. Ale można ją do tego zmusić używając słowa kluczowego `checked` (o tym poniżej).

Zwróćmy także uwagę na okno *Call Stack* (z ang. stos wywołań), które widoczne jest z prawej strony na dole okna Visual Studio. Wymienione są w nim wszystkie metody, począwszy od metody `Main`, a skończywszy na metodzie, której polecenie jest obecnie wykonywane. W naszym programie stos wywołań składa się tylko z metody `Main` i metody `Kwadrat`, ale w bardziej rozbudowanych programach tych metod może być sporo. Zawartość tego okna pomaga wówczas w zorientowaniu się, co się aktualnie dzieje w aplikacji, szczególnie po jej zatrzymaniu w wyniku działania *breakpointu*.

Stan wyjątkowy

Zgłaszanie wyjątków

Ostatnią rzeczą, o której chciałbym wspomnieć w kontekście tropienia błędów, jest obsługa wyjątków zgłaszanych przez aplikację i reakcja debuggera w tej sytuacji. Dodajmy do metody `Kwadrat` polecenie zgłaszające wyjątek w przypadku wykrycia błędnej wartości zmiennej `wartosc`.

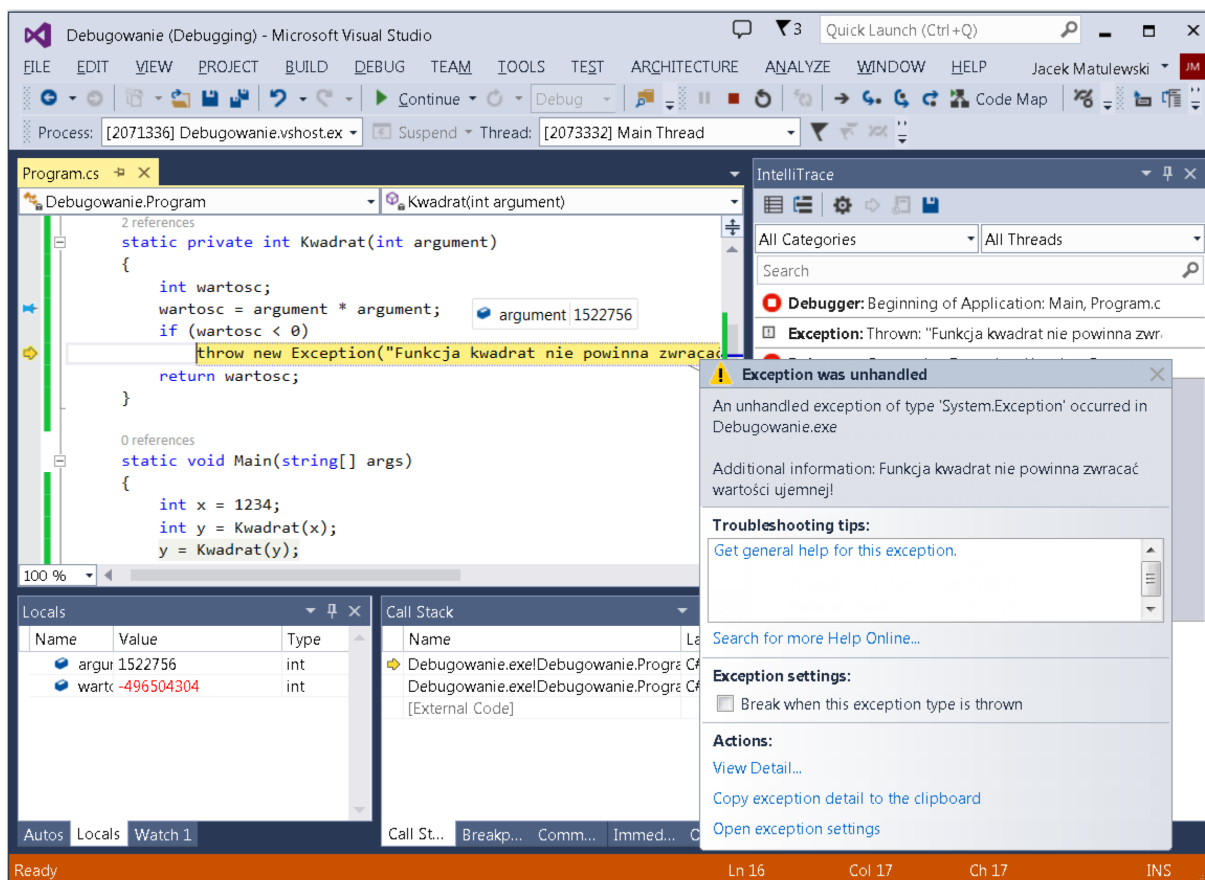
1. Do metody `Kwadrat` dodaj polecenie wyróżnione w listingu 2.3.

Listing 2.3. Metoda kwadrat z instrukcją zgłaszającą wyjątek w przypadku ujemnego wyniku

```
static private int Kwadrat(int argument)
{
   int wartosc;
   wartosc = argument * argument;
   if (wartosc < 0)
      throw new Exception("Funkcja kwadrat nie powinna zwracać wartości ujemnej!");
   return wartosc;
}
```

2. Usuń z kodu wszystkie *breakpointy*.
3. Uruchom aplikację klawiszem *F5* (uruchamianie z debugowaniem).

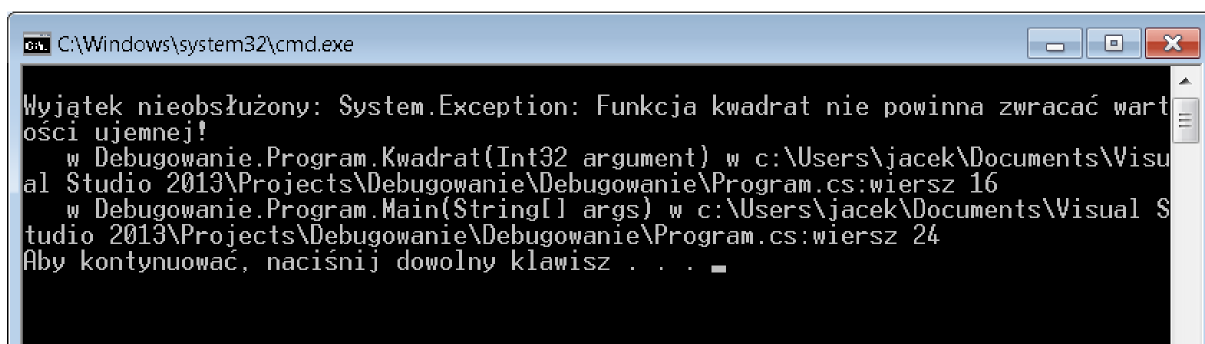
Po uruchomieniu, w oknie Visual Studio pojawi się komunikat debuggera informujący o tym, że aplikacja *Debugowanie* zgłosiła wyjątek typu `System.Exception` (rysunek 2.5). Przyjrzyjmy się temu komunikatowi. Przede wszystkim na pasku tytułu widoczne jest ostrzeżenie *Exception was unhandled*, co oznacza, że słowo kluczowe `throw` znajdowało się we fragmencie kodu, który nie był otoczony obsługą wyjątków. Nie znajdowało się zatem w sekcji `try` konstrukcji `try..catch`, ani w metodzie, która byłaby z tej sekcji wywołana. To ważna informacja, bo 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 `Exception`). Po kliknięciu *View Detail...* można obejrzeć stan przesyłanego obiektu w widocznym na dole okna odpowiedniku okna *Locals*.



Rysunek 2.5. Komunikat o wyjątku zgłoszony przez Visual C#

Jak widzimy, po zgłoszeniu przez debugowany program 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*.

Jeżeli aplikacja zostanie uruchomiona poza debuggerem przez naciśnięcie kombinacji klawiszy *Ctrl+F5* lub samodzielne uruchomienie pliku *.exe* poza środowiskiem Visual Studio i aplikacja nie przechwytuje zgłoszonych w niej wyjątków, wyjątek taki jest przechwytywany przez platformę .NET. O takiej sytuacji użytkownik powiadamiany jest komunikatem widocznym na rysunku 2.6. W aplikacji konsolowej sytuacja taka oznacza zakończenie działania programu. Warto jednak wiedzieć, że w aplikacji Windows Forms, którym poświęcona jest druga część tej książki, po zgłoszeniu nieobsługiwanego wyjątku istnieje możliwość kontynuowania jej działania. Oczywiście, wyjątek pojawi się znowu, jeżeli ponownie wejdziemy do metody zawierającej błąd, 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 na dysku dokumentu.



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

Przechwytywanie wyjątków w konstrukcji try..catch

Teraz spróbujmy wykryć wystąpienie wyjątku z poziomu programu. W tym celu w metodzie `Main` otoczymy wywołanie metody `Kwadrat` konstrukcją przechwytywania wyjątków.

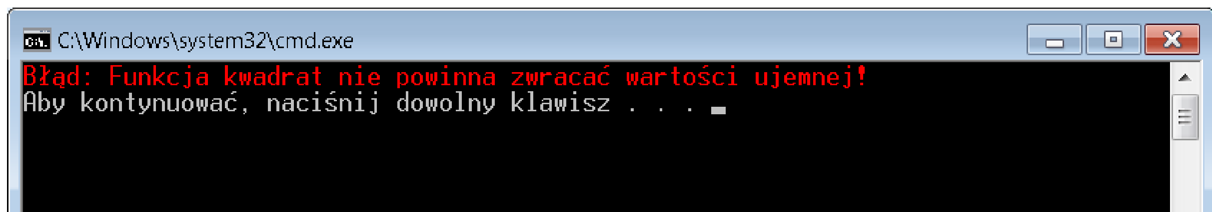
1. Naciśnij *Shift+F5*, aby zakończyć działanie debuggera.
2. Do metody `Main` dodaj obsługę wyjątku zgodnie ze wzorem na listingu 2.4.

Listing 2.4. Wszystkie dotychczasowe polecenia metody Main umieściliśmy w jednej sekcji try

```
static void Main(string[] args)
{
    try
    {
        int x = 1234;
        int y = Kwadrat(x);
        y = Kwadrat(y);
        string sy = y.ToString();
        Console.WriteLine(sy);
    }
    catch (Exception ex)
    {
        ConsoleColor bieżącyKolor = Console.ForegroundColor;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.Error.WriteLine("Błąd: " + ex.Message);
        Console.ForegroundColor = bieżącyKolor;
    }
}
```

3. Skompiluj i uruchom aplikację.

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



Rysunek 2.7. Komunikat wyświetlany użytkownikowi po przechwyceniu wyjątku

Wszystkie polecenia metody `Main` 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.

Więcej informacji na temat wyjątków można znaleźć w następnym rozdziale.

Wymuszenie kontroli zakresu zmiennych

Wspomniałem wyżej, że możemy zmusić wirtualną maszynę platformy .NET do kontrolowania przekraczania zakresu w operacjach arytmetycznych i podczas konwersji typów. Można to uzyskać korzystając ze słowa kluczowego `checked` (więcej na jego temat w rozdziale 6.). Wówczas w razie wykrycia przekroczenia zakresu

zgłaszany jest wyjątek OverflowException. Należy być jednak świadomym, że kontrola taka znacznie obniża wydajność programu, dlatego domyślnie nie jest włączona.

Zmodyfikujmy ostatni raz metodę kwadrat zgodnie ze wzorem widocznym na listingu 2.5. Jeżeli teraz uruchomimy projekt, wyjątek zostanie zgłoszony już w linii, w której zmienna argument podnoszona jest do kwadratu (przy drugim wywołaniu zmiennej Kwadrat). Mechanizm przechwytywania wyjątków z metody Main zadziała jednak podobnie i w oknie konsoli zobaczymy komunikat „Nastąpiło przepełnienie w czasie wykonywania operacji arytmetycznej".

Listing 2.5. Kontrola przekroczenia zakresu zmiennej int przy mnożeniu

```
static private int Kwadrat(int argument)
{
    int wartosc;
    wartosc = checked(argument * argument);
    if (wartosc < 0)
        throw new Exception("Funkcja kwadrat nie powinna zwracać wartości ujemnej!");
    return wartosc;
}
```