Microsoft® Visual Studio® 2010

Mike Snell, Lars Powers

Poznaj środowisko Visual Studio 2010 Professional i naucz się tworzyć mistrzowskie aplikacje

- Jak wykorzystać technologię ASP.NET do tworzenia profesjonalnych witryn internetowych?
- Jak tworzyć aplikacje biznesowe oparte na pakiecie Office?
- Jak kompleksowo testować swój kod, zaprzęgając do pracy Visual Studio?

Microsoft® Visual Studio® 2010 KSIĘGA EKSPERTA

Helion

Microsoft Visual Studio 2010. Księga eksperta

Autorzy: Mike Snell, Lars Powers
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-3029-5
Tytuł oryginału: Microsoft Visual Studio 2010 Unleashed
Format: 172×245, stron: 1264

Poznaj środowisko Visual Studio 2010 Professional i naucz się tworzyć mistrzowskie aplikacje

- Jak wykorzystać technologię ASP.NET do tworzenia profesjonalnych witryn internetowych?
- Jak tworzyć aplikacje biznesowe oparte na pakiecie Office?
- Jak kompleksowo testować swój kod, zaprzęgając do pracy Visual Studio?

Visual Studio 2010 to najnowsza wersja środowiska programistycznego firmy Microsoft. Każdy programista, który zdecyduje się wykorzystać bogaty zestaw zgromadzonych tu narzędzi, osiągnie maksymalną produktywność w pracy i będzie mógł tworzyć kod działający w systemie Windows oraz w sieci. Dodatkowo będzie mieć do dyspozycji technologię Silverlight i możliwość budowania aplikacji w chmurze, z użyciem platformy Microsoftu – Azure. Nie koniec na tym. W Visual Studio 2010 pojawiły się kolejne innowacje. Znajdziesz tu nowy edytor kodu, oparty na platformie WPF, rozszerzenia środowiska IDE związane z platformą MEF oraz możliwość pisania skryptów za pomocą technologii Silverlight. Ponadto języki platformy .NET wzbogacono o obsługę programowania dynamicznego. Pojawił się nowy język F#, służący do programowania funkcyjnego, oraz mechanizmy usprawniające szybkie pisanie kodu wyższej jakości.

„Microsoft Visual Studio 2010. Księga eksperta" zawiera kompletne omówienie środowiska Visual Studio 2010, a skoncentrowanie się na wersji Professional pozwoliło autorom na stworzenie opisu bardziej szczegółowego niż kiedykolwiek wcześniej. Dzięki temu podręcznikowi nauczysz się w pełni wykorzystywać możliwości platformy .NET, w tym technologii WPF (pozwalającej na tworzenie bogatych klientów), technologii WCF (stworzonej do budowania dynamicznych rozwiązań opartych na usługach) czy też technologii WF (umożliwiającej rozwijanie ustrukturyzowanych programów na podstawie procesów biznesowych). Znajdziesz tu także omówienie nowych narzędzi Microsoftu, przeznaczonych do testowania, instrumentacji aplikacji i analizowania kodu.

- Języki platformy .NET
- Przeglądarki i eksploratory
- Tworzenie projektów WPF
- Szablony XML
- Testowanie, refaktoryzacja i diagnozowanie kodu
- Platforma MEF
- Tworzenie aplikacji ASP.NET
- Bogate i inteligentne interfejsy użytkownika
- Złożone aplikacje internetowe
- Silverlight 4.0
- Aplikacje biznesowe oparte na pakiecie Office
- Technologia Windows Azure i aplikacje działające w chmurze

Wykorzystaj zdobytą wiedzę i zostań mistrzem programowania!

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

Spis treści

Rozdział 9.
Testowanie kodu

W tym rozdziale:

- Podstawy testów jednostek
- Platforma testów jednostek
- Pisanie testów jednostek działających w ASP.NET
- Konfigurowanie atrybutów testów jednostek
- Tworzenie testów uporządkowanych
- Organizowanie testów

Programiści zawsze byli odpowiedzialni za testowanie kodu przed udostępnieniem go testerom lub użytkownikom. W przeszłości oznaczało to przejście przez każdy wiersz kodu w debugerze (włączając w to wszystkie warunki i błędy). W tym celu często trzeba było przygotować aplikacje testowe z operacjami potrzebnymi do wykonania kodu. Przejście przez cały kod pozwalało dojść do celu, ale nie zawsze programistom chciało się to robić (i bardzo trudno było to sprawdzić). W praktyce całą tę operację często pomijano w czasie wprowadzania zmian w kodzie i jego aktualizacji. Opisany proces utrudniał ponadto ustalenie, czy zmiany w kodzie wpływają na inne części systemu, a także powodował przesyłanie testerom i użytkownikom wersji o niższej jakości, co zwiększało liczbę usterek oraz wydłużało czas potrzebny na przesyłanie kodu tam i z powrotem między programistami a testerami.

Ten mechanizm (i efekty jego stosowania) wskazywał na potrzebę automatyzacji testowania jednostek. W efekcie powstały różne przeznaczone do tego platformy. Pierwszą taką platformą dla .NET była NUnit (*nunit.org*) — projekt o otwartym dostępie do kodu źródłowego działający dla większości języków .NET, który umożliwia pisanie kodu służącego do testowania innego kodu. Podobna platforma została wbudowana w Visual Studio 2005 i jest dostępna także w obecnej wersji.

Platforma testów jednostek w Visual Studio umożliwia kompilowanie testów wraz z aplikacją. Można też zdecydować się na zastosowanie programowania sterowanego testami i przygotować testy przed rozpoczęciem pisania kodu. W każdym przypadku ustrukturyzowane podejście do testów jednostek może doprowadzić do utworzenia pełnego zestawu testów współpracującego z aplikacją.

Pełny zestaw testów może składać się na test regresji dla większości komponentów, a nawet dla całego systemu. Daje to większą pewność przy wykonywaniu operacji, które wcześniej były bardzo ryzykowne, na przykład przy wprowadzaniu poprawek w ostatniej chwili, dokonywaniu refaktoryzacji czy dodawaniu elementów tuż przed zakończeniem pracy. Przy wykonywaniu tych działań można wykorzystać pełny zestaw testów jednostek do sprawdzenia, jakie elementy (i czy w ogóle) zostały zepsute w wyniku wprowadzenia zmian.

W tym rozdziale opisano wiele narzędzi, technologii i technik przeprowadzania testów przez programistów. Dzięki tym informacjom można zacząć czerpać liczne korzyści ze zautomatyzowanych testów dostępnych programistom. Prowadzą one do zmniejszenia liczby błędów, powstawania bardziej zrozumiałego kodu i większej pewności przy wprowadzaniu zmian w kodzie.

Podstawy testów jednostek

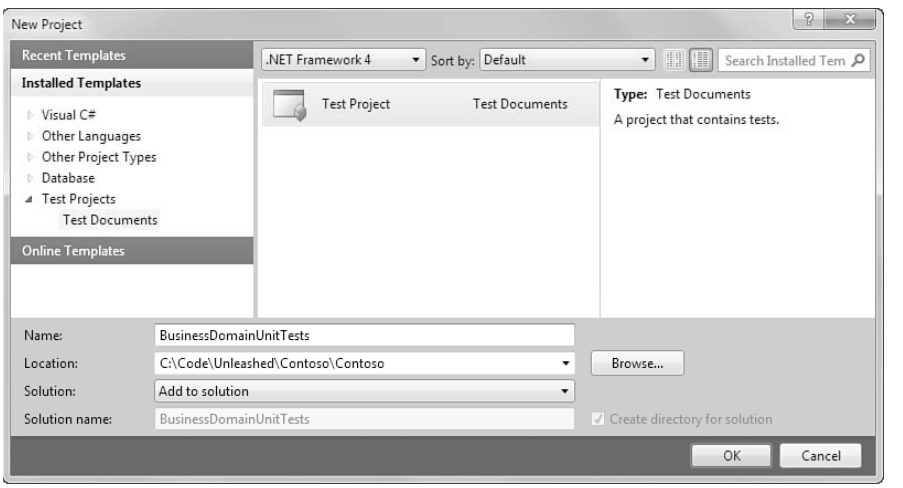
Testy jednostek w Visual Studio polegają na tworzeniu testów sprawdzających kod warstw aplikacji. Warstwy obejmują liczne klasy z obszaru logiki biznesowej i obsługi danych. Do sprawdzania interfejsu użytkownika służą zwykle inne metody automatycznego testowania. Są one przeznaczone głównie dla testerów. Potrzebne do tego funkcje i produkty znajdują się w wersji Premium środowiska Visual Studio, dlatego nie opisujemy ich w tym miejscu. W zamian koncentrujemy się na tworzeniu kodu do testowania innego kodu. Oczywiście, zakładamy przy tym, że programista w czasie rozwijania aplikacji stosuje prawidłowe techniki budowania architektury warstwowej. W tym podrozdziale omawiamy podstawy pisania testów jednostek. W dalszych punktach szczegółowo opisujemy przedstawione tu zagadnienia.

Tworzenie projektu testów

Testy programistów trzeba umieszczać w projektach testów. Projekt zawiera odpowiednie referencje do platformy testów jednostek i trzeba skonfigurować go tak, aby można go było uruchomić za pomocą narzędzi testowych wbudowanych w Visual Studio. Istnieją dwa główne sposoby tworzenia projektów testów jednostek. Można przygotować projekt testów w środowisku IDE lub automatycznie wygenerować testy jednostek dla istniejącego kodu w nowym projekcie testów. Zaczniemy od pierwszej z tych możliwości.

Szablon projektu testów

Nowy projekt testów w Visual Studio można utworzyć w oknie dialogowym *New Project* (*File/New/Project*). W oknie dialogowym należy przejść do węzła *Test Projects* w drzewie *Installed Templates*. Węzeł pokazano na rysunku 9.1. Warto zauważyć, że projekt testów można umieścić w nowym lub istniejącym rozwiązaniu. Zwykle testy projektów dodaje się do gotowych rozwiązań, ponieważ w testach trzeba umieścić referencje do projektów z rozwiązania.



Rysunek 9.1. W oknie dialogowym New Project można utworzyć nowy projekt testów dla rozwiązania

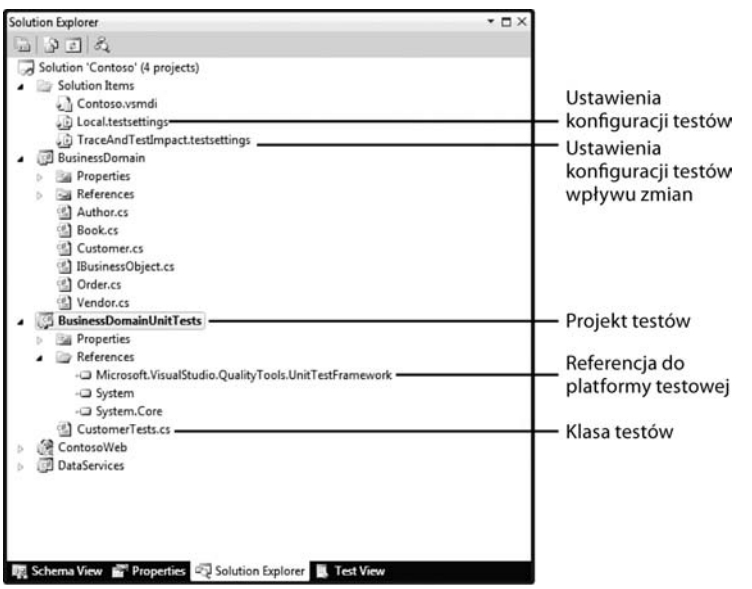
Dobre praktyki tworzenia projektów i klas testów

Należy ustalić reguły określające liczbę projektów testów tworzonych dla rozwiązania. Zwykle programiści tworzą jeden projekt testów dla każdego sprawdzanego projektu. Na przykład jeśli w jednym projekcie znajdują się obiekty biznesowe, a w innym usługi związane z danymi, można utworzyć dwa projekty testowe (po jednym dla każdego projektu głównego). Nie jest to jednak wymóg, a jedynie sposób na łatwe zrozumienie organizacji kodu.

Podobne zasady dotyczą klas testów. Należy utworzyć jedną klasę testów dla każdej klasy ze sprawdzanego docelowego projektu. Jeśli programista tworzy projekt testów dla kodu logiki biznesowej obejmującego obiekty `Customer` i `Order`, powinien przygotować klasy `CustomerTest` i `OrderTest`. Nie jest to konieczne, ale prowadzi do dobrego, logicznego uporządkowania kodu.

Projekt testów

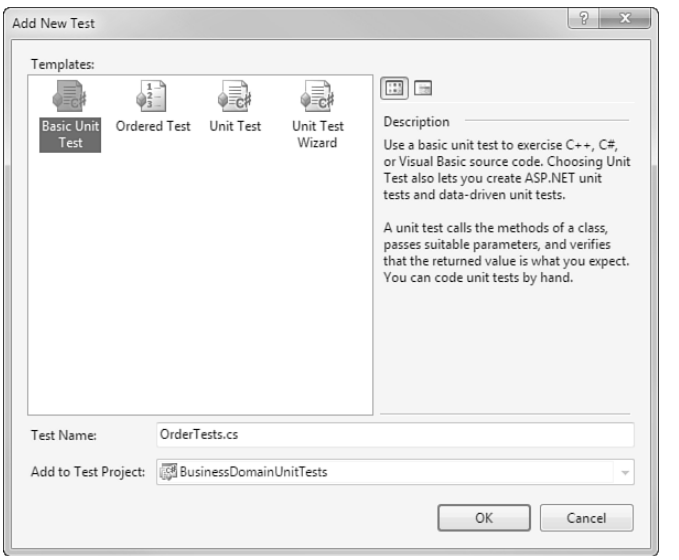
W czasie tworzenia nowego projektu testów Visual Studio dodaje referencje do platformy testów jednostek (`Microsoft.VisualStudio.QualityTools.UnitTestFramework`). Ponadto dodaje do rozwiązania dwa pliki (w katalogu *Solution Items*). Są to pliki z konfiguracją (ustawieniami) testów. Ich nazwy to *Local.testsettings* i *TraceAndTestImpact.testsettings*. W projekcie testów tworzona jest też klasa testów używana do pisania testów jednostek. Na rysunku 9.2 pokazano każdy z tych elementów w oknie *Solution Explorer*. Poszczególne pliki omawiamy szczegółowo w dalszych punktach.



Rysunek 9.2. Projekt testów i powiązane elementy w oknie Solution Explorer

Do projektu można dodać nowe pliki testów. W tym celu należy kliknąć prawym przyciskiem myszy projekt testów i wybrać opcję *Add/New Test*. Ponadto można użyć menu *Project* i otworzyć okno dialogowe *Add New Test* widoczne na rysunku 9.3. W tym miejscu dostępne są następujące opcje:

- ***Basic Unit Test.*** Ten szablon tworzy prostą klasę testów z pustą metodą testową. Używa się go zwykle przy tworzeniu testów od podstaw (bez korzystania z funkcji automatycznego generowania testów).
- ***Database Unit Test.*** Ten szablon służy do tworzenia testów jednostek wykonywanych na procedurach składowanych z bazy danych (do stosowania tej funkcji potrzebna jest wersja Premium środowiska Visual Studio).
- ***Ordered Test.*** Ten szablon umożliwia utworzenie sekwencyjnej listy wspólnie wykonywanych testów (zobacz podrozdział „Tworzenie testów uporządkowanych" w dalszej części rozdziału).
- ***Unit Test.*** Ten szablon tworzy pustą klasę testów jednostek korzystającą z obiektu do śledzenia kontekstu testów. Tego typu klasy testów jednostek są generowane automatycznie (zobacz punkt „Generowanie testów na podstawie istniejącego kodu").
- ***Unit Test Wizard.*** Ten szablon uruchamia kreator Unit Test Wizard służący do automatycznego generowania testów na podstawie istniejącego kodu rozwiązania (zobacz punkt „Generowanie testów na podstawie istniejącego kodu").

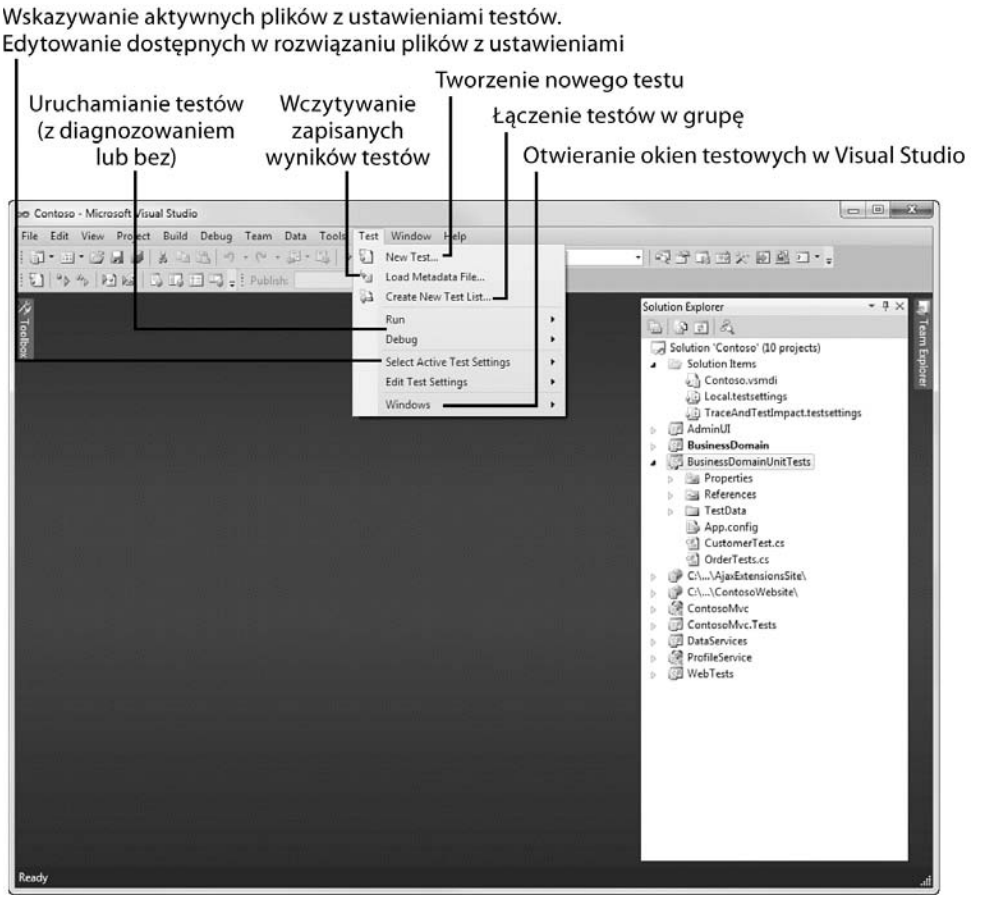


Rysunek 9.3. Za pomocą okna dialogowego Add New Test można dodawać nowe testy do projektu testów

Menu Test

Menu *Test* można wykorzystać do uzyskania dostępu do wszystkich funkcji środowiska Visual Studio związanych z testami przeprowadzanymi przez programistów. Menu pozwala między innymi dodawać klasy testów jednostek do projektu testów (*Test/New Test*). Można go ponadto

używać do wczytywania informacji o wcześniejszych testach (*Load Metadata File*), uruchamiania testów (z diagnozowaniem lub bez), wybierania ustawień stosowanych w następnym przebiegu testów, a także edytowania takich ustawień. Można też uzyskać dostęp do wielu okien służących do wyświetlania i organizowania testów w aplikacji. Na rysunku 9.4 pokazano przykład ilustrujący wiele funkcji omawianego menu.



Rysunek 9.4. Menu Test środowiska Visual Studio pozwala uzyskać dostęp do funkcji środowiska związanych z testami przeprowadzanymi przez programistów

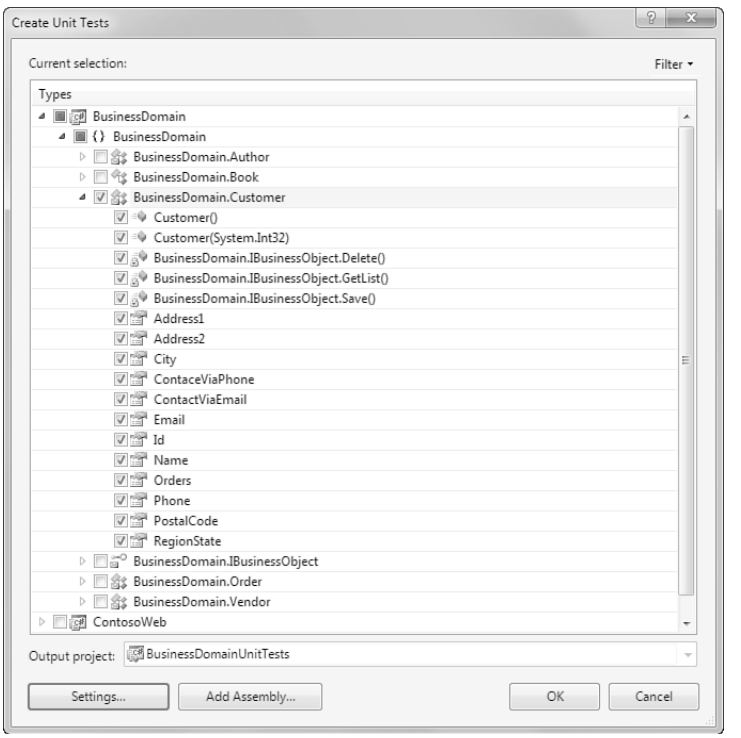
Generowanie testów na podstawie istniejącego kodu

Visual Studio umożliwia też automatyzację procesu tworzenia testów jednostek. Można kliknąć prawym przyciskiem myszy istniejącą klasę i wybrać z menu podręcznego opcję *Create Unit Tests*. Inna możliwość to wybranie szablonu elementu *Unit Test Wizard*. Oba rozwiązania powodują utworzenie przez Visual Studio zestawu testów jednostek na podstawie kodu klasy.

Wygenerowany kod to dobry punkt wyjścia do rozwijania testów jednostek. Visual Studio analizuje metody i właściwości klasy, na podstawie której programista chce wygenerować testy, oraz tworzy prawdziwe, możliwe do uruchomienia testy. Oczywiście, trzeba dokończyć każdy test przez dodanie odpowiednich wartości zmiennych i asercji, jednak kod wygenerowany przez narzędzia to dobry punkt wyjścia.

Przyjrzyjmy się przykładowi. Załóżmy, że programista używa obiektu `Customer` zawierającego standardowe właściwości, takie jak `Name`, `Address`, `Phone` i `Email`. Może także zawierać inne metody, takie jak `Save`, `Update` czy `Delete`. Środowiska Visual Studio można użyć do wygenerowania nowych testów jednostek dla klasy. W tym celu należy albo wybrać opcję *Unit Test Wizard* w oknie dialogowym *Add New Test*, albo kliknąć prawym przyciskiem myszy plik klasy i wybrać opcję *Create Unit Tests*.

W obu scenariuszach Visual Studio wyświetli okno dialogowe *Create Unit Tests* przedstawione na rysunku 9.5. Widoczne są w nim typy dostępne w rozwiązaniu. W powyższym oknie dialogowym można wybrać składowe, dla których trzeba wygenerować testy. Można też użyć ustawienia *Output project* do określenia projektu testów, w którym mają znaleźć się wygenerowane testy. Polega to na utworzeniu nowego projektu testów (w języku C# lub Visual Basic) albo na umieszczeniu kodu w istniejącym projekcie testów.

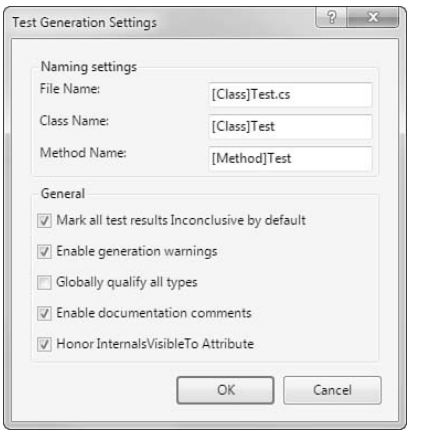


Rysunek 9.5. Można zezwolić środowisku Visual Studio na wygenerowanie testów jednostek na podstawie istniejącego kodu

Przycisk *Add Assembly* w dolnej części okna dialogowego umożliwia wybranie podzespołu (pliku *.dll*) spoza projektu. Testy można wygenerować dla dowolnego podzespołu .NET.

Zarządzanie ustawieniami procesu generowania testów

Przycisk *Settings* pozwala otworzyć okno dialogowe, które służy do określania różnych ustawień związanych z generowaniem testów jednostek. Rysunek 9.6 przedstawia przykładowe informacje w tym oknie. Warto zauważyć, że można użyć tekstu makr (`[File]`, `[Class]` i `[Method]`) do poinformowania Visual Studio o tym, jak ma tworzyć nazwy generowanych plików, klas i metod testów.



Rysunek 9.6. Można kontrolować sposób generowania testów jednostek przez Visual Studio

Warto zwrócić uwagę na dodatkowe ustawienia w grupie *General*. Umożliwiają one ustawienie każdego generowanego testu jako niejednoznacznego, umieszczenie komentarzy w wygenerowanym kodzie i tak dalej.

Uwaga

Kreator Create Unit Tests umożliwia wygenerowanie testów jednostek dla wszystkich projektów w rozwiązaniu. Zaznaczenie klasy i wygenerowanie testów powoduje dodanie w projekcie testów referencji do projektu z testowanym kodem.

Wygenerowany kod testów

Visual Studio generuje testy dla wszystkich metod i właściwości obiektu zaznaczonych w oknie dialogowym *Create Unit Tests*. Zastanówmy się na przykład nad właściwością `Name`. Listing 9.1 przedstawia kod wygenerowany jako test jednostek tej właściwości (kod w języku Visual Basic wygląda podobnie). Test tworzy nowy obiekt typu `Customer`, a następnie próbuje ustawić wartość właściwości `Name` docelowej zmiennej. Potem określany jest oczekiwany efekt. Tu właściwość `Name` jest typu `string`, dlatego kod testu tworzy zmienną o nazwie `expected` tego typu. Definiuje też zmienną typu `string` o nazwie `actual`. Następnie przypisuje do właściwości `Customer.Name` wartość zmiennej `exptected` i — po ustawieniu właściwości — wczytuje wartość zmiennej `actual`. Na zakończenie za pomocą asercji `Assert.AreEqual` sprawdza,

czy właściwość została poprawnie ustawiona. Jest to prawidłowy test właściwości. Programista musi jedynie podać poprawną wartość dla zmiennej `exptected` (punkt `TODO`) i usunąć wywołanie `Assert.Inconclusive`.

Listing 9.1. *Przykładowy automatycznie wygenerowany test jednostek*

```
[TestMethod()]
public void NameTest() {
   Customer target = new Customer(); // TODO: Initialize to an appropriate value
   string expected = string.Empty; // TODO: Initialize to an appropriate value
   string actual;
   target.Name = expected;
   actual = target.Name;
   Assert.AreEqual(expected, actual);
   Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Testowanie metod jest trochę bardziej pracochłonne. Jeśli metoda zwraca wartość, trzeba ją sprawdzić, aby się upewnić, że jest zgodna z oczekiwaniami. Jeżeli metoda nie zwraca wartości, trzeba napisać dodatkowy kod do sprawdzenia, czy działa prawidłowo. Na listingu 9.2 pokazano automatycznie wygenerowany test metody, która nie zwraca wartości. Oczywiście, jeśli wywołanie metody spowoduje błąd, test zakończy się niepowodzeniem. Można jednak napisać dodatkowe testy i asercje.

Listing 9.2. *Przykładowy automatycznie wygenerowany test jednostki*

```
[TestMethod()]
public void SaveTest() {
   Customer target = new Customer(); // TODO: Initialize to an appropriate value
   target.Save();
   Assert.Inconclusive("A method that does not return a value cannot be
   ↪verified.");
}
```

Uwaga

W czasie generowania kodu do testowania konstruktorów Visual Studio dodaje tekst `ConstructorTest` do nazwy klasy. Jeśli istnieje kilka konstruktorów, Visual Studio doda do nazw liczby, na przykład `CustomerConstructorTest1`.

Pisanie testów jednostek

Możliwe, że Czytelnicy zaczynają się domyślać, jak wykorzystać automatycznie wygenerowane testy jednostek. Przyjrzyjmy się pisaniu testów jednostek, aby lepiej je zrozumieć. Warto pamiętać, że test jednostek to po prostu kod testu, który ma wywoływać kod aplikacji. W takim kodzie testu znajdują się założenia dotyczące tego, czy określone warunki są spełnione, czy

nie po wykonaniu kodu aplikacji. Test może powieść się lub nie w zależności od wyników tych założeń. Jeśli programista oczekuje, że warunek będzie spełniony, a tak się nie stanie, wtedy test kończy się niepowodzeniem.

Kiedy projekt i klasa testu są już gotowe, trzeba wykonać trzy operacje, aby utworzyć typowy test jednostki:

1. Dodać atrybut `TestMethod` do kodu testu.
2. Uruchomić badany kod, przekazując znane wartości w taki sposób, jak parametry.
3. Zastosować asercje do oceny wyników testów.

Przyjrzyjmy się bliżej praktycznemu przykładowi. Załóżmy, że programista korzysta z usługi sieciowej zwracającej profil użytkownika z bazy danych. Ta usługa sieciowa przyjmuje jako parametr identyfikator klienta. Sygnatura funkcji może wyglądać tak:

```
public Customer GetCustomerProfile(int customerId)
```

Można napisać prosty test wywołujący tę usługę i przekazujący znany identyfikator z bazy danych. Dobrym sposobem na testy jednostek baz danych jest sprawdzanie znanego stanu bazy. W teście można następnie sprawdzić nie tylko to, czy zwracane dane działają, ale także czy są poprawne. Na listingu 9.3 znajduje się przykładowy test.

Listing 9.3. *Przykładowy test jednostek*

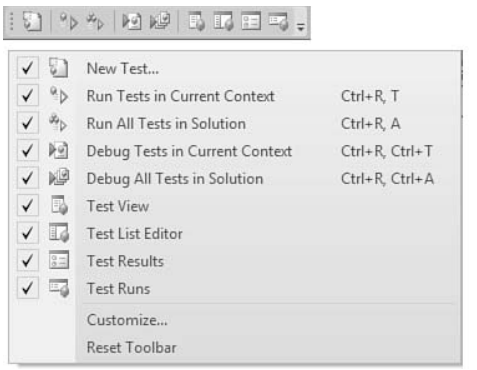
```
[TestMethod()]
public void GetCustomerProfileTest() {
   CustomerProfile cutProfileService = new CustomerProfile();
   int customerId = 1234;
   Customer customer = cutProfileService.GetCustomerProfile(customerId);
   Assert.AreEqual(customer.Id, 1234);
}
```

Warto zauważyć, że powyższy kod jest podobny do dowolnego kodu w języku C# (i Visual Basic). Aby oznaczyć metodę jako test, należy dodać atrybut `TestMethod`. W kodzie trzeba utworzyć egzemplarz testowanego obiektu i wywołać sprawdzaną metodę. Jeśli wywołanie się nie powiedzie (lub zostanie zgłoszony wyjątek), test zakończy się niepowodzeniem. Następnie w teście znajduje się asercja, która pozwala sprawdzić, czy zwrócony obiekt odpowiada oczekiwanemu. Jeśli ta asercja okaże się fałszywa (przekazane do niej wartości nie są sobie równe), test zakończy się niepowodzeniem. Jeśli asercja będzie prawdziwa, test się powiedzie. Można dodać kilka innych asercji, aby uzupełnić powyższy test (przez sprawdzenie innych znanych danych na temat klienta). Ponadto można utworzyć pewne dodatkowe testy sprawdzanej metody, na przykład określające, co się stanie przy próbie pobrania danych nieistniejącego klienta.

Powyższy kod powinien pozwolić zrozumieć podstawy działania testów jednostek. W dalszej części rozdziału zajmiemy się nimi nieco bardziej szczegółowo.

Uruchamianie testów jednostek

Testy jednostek można uruchamiać w kilku miejscach środowiska Visual Studio. Dwa z nich to pasek narzędzi *Test Tools* i menu *Test*. Obie możliwości związane są z podobnym zestawem opcji pokazanych na rysunku 9.7. W górnej części rysunku znajduje się pasek narzędzi *Test Tools*. W dolnej części pokazano zestaw przycisków opcji, które można dodać do paska lub usunąć z niego. Lista ta pomaga zrozumieć znaczenie ikon paska narzędzi i poznać klawisze skrótu służące do uruchamiania testów.



Rysunek 9.7. Menu Test umożliwia uruchamianie testów z debugerem lub bez niego

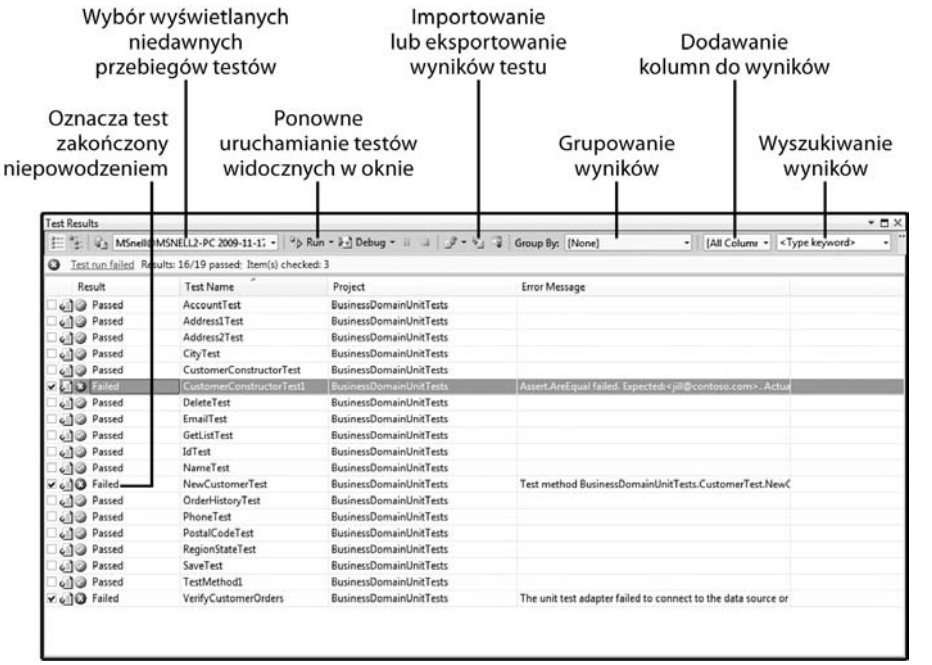
Testy można uruchamiać z debugerem lub bez. Pierwsza z tych opcji umożliwia włączenie debugera, jeśli test zakończy się niepowodzeniem. Ta właściwość jest przydatna do używania testów do rozwiązywania problemów z kodem. Częściej stosuje się jednak drugie rozwiązanie, które pozwala po prostu uruchomić zestaw testów jednostek i sprawdzić wyniki. Testy można uruchamiać wtedy w bieżącym kontekście lub w kontekście całego rozwiązania. Bieżący kontekst pozwala uruchomić dowolne wybrane teksty, przeprowadzić testy od lokalizacji kursora lub włączyć wszystkie testy danej klasy (jeżeli wyświetlany jest kod klasy, a nie testu). Czasem trudno określić, co jest bieżącym kontekstem dla środowiska Visual Studio. Wtedy można albo uruchomić wszystkie testy z rozwiązania, albo przeprowadzić testy z poziomu okna *Test View* (zobacz podrozdział „Organizowanie testów").

Uwaga

Przy uruchamianiu projektu testów wszystkie projekty, do których dodano referencje, są rekompilowane razem z projektem testów.

Przeglądanie wyników testów

Po uruchomieniu testów wyniki można zobaczyć w oknie *Test Results* (zwykle zadokowane jest w dolnej części środowiska IDE). Okno umożliwia sprawdzenie, które testy zakończyły się powodzeniem, a które nie. Rysunek 9.8 przedstawia liczne funkcje dostępne w tym oknie. Warto

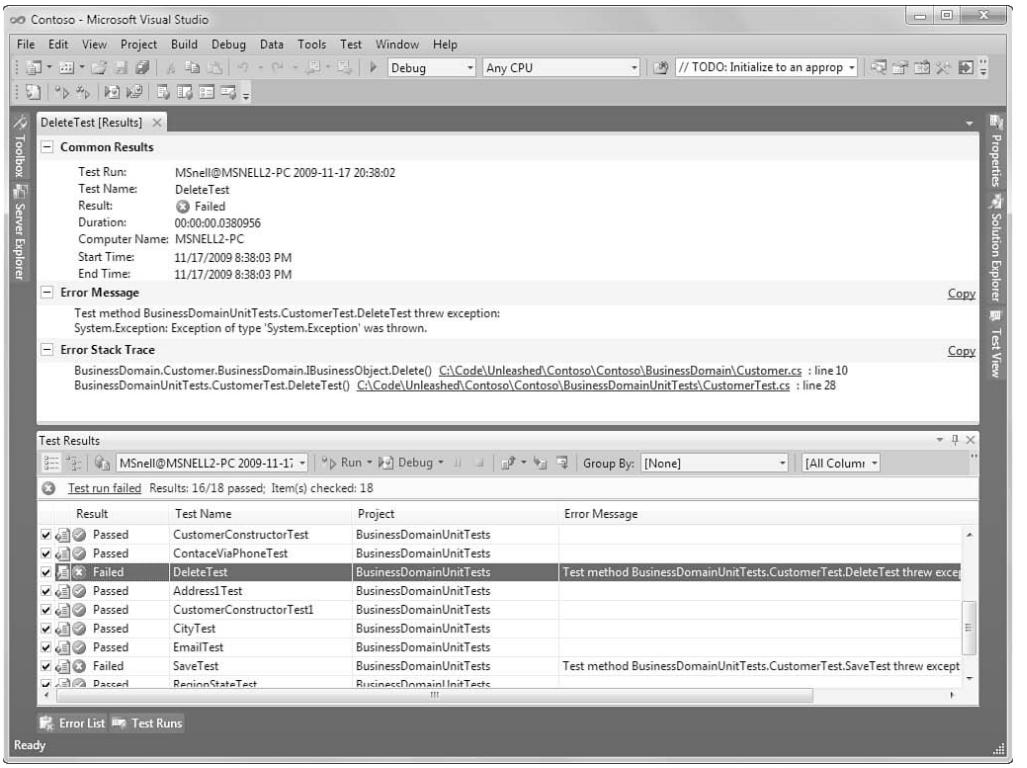


Rysunek 9.8. Okno Test Results umożliwia wyświetlenie wyników przebiegów testów i ponowne uruchomienie wybranych testów

zwrócić uwagę na to, że dany przebieg testu kończy się niepowodzeniem, jeśli nie powiedzie się przynajmniej jeden test. W tym przypadku powiodło się 16 z 18 testów. Jednak niepowodzenie dwóch testów spowodowało, że cały test zakończył się porażką.

Okno *Test Results* udostępnia pasek narzędzi z wieloma opcjami i kontrolkami. Jedna z opcji umożliwia wyświetlenie wyników wszystkich testów przeprowadzonych w czasie sesji środowiska Visual Studio. Pozwala to na wyświetlenie jednocześnie wyników wszystkich testów. Inna opcja umożliwia łatwe ponowne uruchomienie testów z danego przebiegu. Jeszcze inna służy do eksportowania wyników do pliku i udostępniania danych innym osobom (które mogą zaimportować je za pomocą tego samego paska narzędzi). Można też dodać nowe kolumny do wyników testu, aby wyświetlić więcej danych i lepiej je uporządkować (dostępnych jest ponad 20 kolumn).

Dwukrotne kliknięcie testu zakończonego powodzeniem powoduje wyświetlenie szczegółowych informacji na jego temat. Strona ze szczegółami obejmuje nazwę testu, wyniki, czas trwania testu, a także godzinę jego rozpoczęcia i zakończenia. Po dwukrotnym kliknięciu testu zakończonego niepowodzeniem programista zobaczy wiersz kodu sprawdzanej klasy, w którym wystąpił błąd. Aby zobaczyć szczegółowe informacje o takim teście, należy kliknąć go na liście wyników prawym przyciskiem myszy i wybrać opcję *View Test Results Details*. Na rysunku 9.9 przedstawiono stronę ze szczegółowymi danymi na temat nieudanego testu. Warto zauważyć, że obejmują one komunikat o błędzie, ślad stosu i odnośnik do kodu, który spowodował błąd.



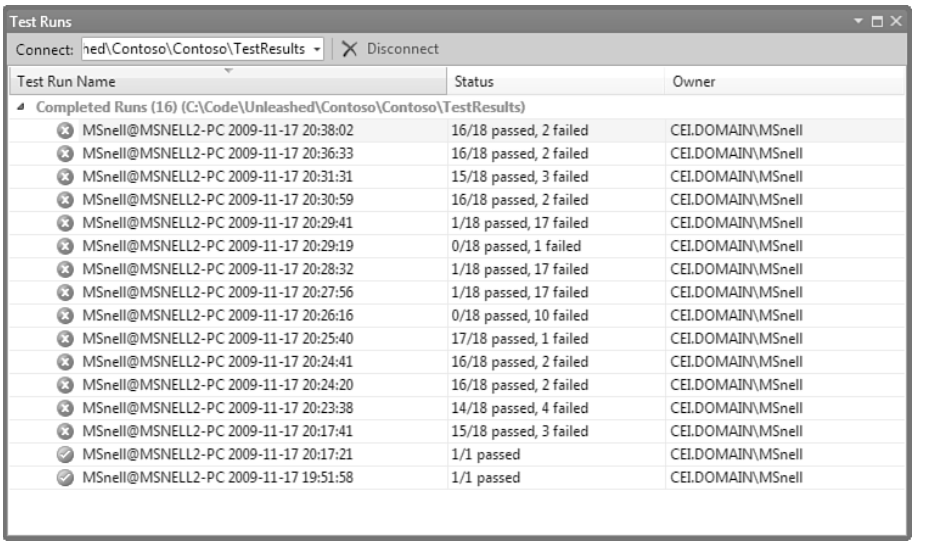
Rysunek 9.9. Można wyświetlić szczegółowe informacje o wynikach dowolnego testu (udanego lub nieudanego)

Wyświetlanie przebiegów testów

Visual Studio udostępnia też okno dialogowe *Test Runs*, służące do przeglądania przebiegów testów z bieżącej sesji (i zaimportowanych). Okno można otworzyć za pomocą paska narzędzi *Test Tools* lub z menu *Test/Windows/Test Runs*. Na rysunku 9.10 pokazano przykładową zawartość omawianego okna dialogowego. Można dwukrotnie kliknąć dany przebieg, aby wczytać szczegółowe informacje o wynikach do okna *Test Results*.

Konfigurowanie opcji i ustawień testów

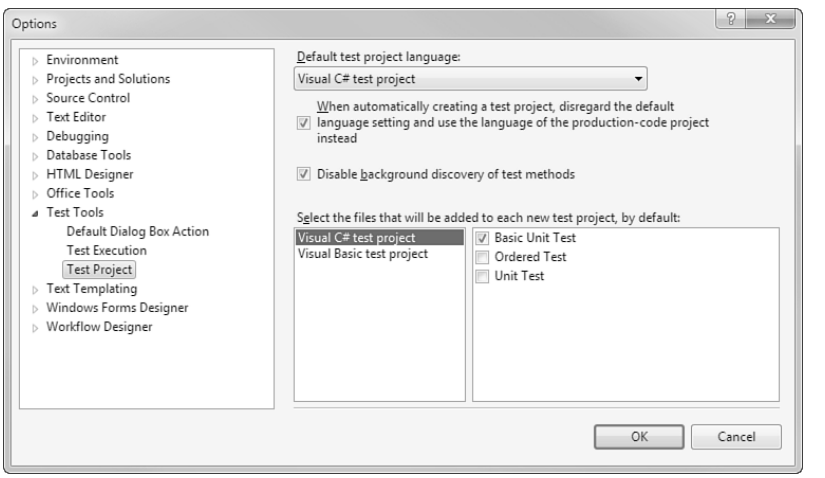
Visual Studio umożliwia kontrolowanie i dostosowywanie środowiska testowego. Służy do tego okno dialogowe *Options*. Ponadto w plikach konfiguracyjnych można kontrolować ustawienia testów specyficznych dla rozwiązania. W tym punkcie opisujemy, jak stosować obie techniki konfigurowania do lepszego zarządzania środowiskiem testowym.



Rysunek 9.10. Okno dialogowe Test Runs umożliwia wyświetlenie wyników wszystkich przebiegów testów z bieżącej sesji środowiska Visual Studio

Opcje projektów testów

Można kontrolować sposób generowania projektów testów przez Visual Studio. Należy to zrobić w węźle *Test Tools/Test Project* w oknie dialogowym *Options* (*Tools/Options*). Na rysunku 9.11 pokazano przykładową zawartość tego okna. Można tu określić język domyślny (Visual Basic lub C#) dla projektu testów stosowany przez Visual Studio przy tworzeniu lub generowaniu nowego projektu tego typu. Warto zauważyć, że można zmienić domyślny język ustawiony w środowisku IDE dla projektów testów.

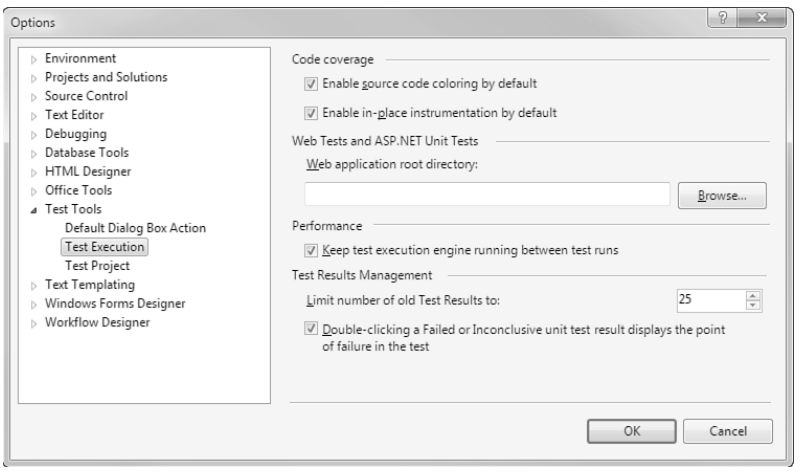


Rysunek 9.11. Okno dialogowe Options umożliwia dostosowanie domyślnych ustawień szablonów projektów testów w Visual Studio

Opcje w węźle *Test Project* pozwalają też określić, które pliki należy domyślnie umieścić w projekcie testów w czasie jego tworzenia. To, jakie opcje są dostępne, zależy od języka. Można określić pliki podstawowych testów jednostek, testów jednostek baz danych, testów uporządkowanych lub testów jednostek (uruchamiany jest wtedy kreator Unit Test Wizard).

Zarządzanie wynikami testów

Visual Studio domyślnie przechowuje wyniki ostatnich 25 przebiegów testów. Po przekroczeniu tego limitu środowisko przed zapisaniem wyników nowego przebiegu usunie informacje o najstarszym. W oknie dialogowym *Options* (*Tools/Options*) można zwiększyć (lub zmniejszyć) tę wartość. Należy wybrać węzeł podrzędny *Test Execution* w węźle *Test Tools*, aby wyświetlić opcje widoczne na rysunku 9.12. Warto zwrócić uwagę na opcję *Limit Number of Old Test Runs To* (czyli ogranicz liczbę dawnych przebiegów testów do).



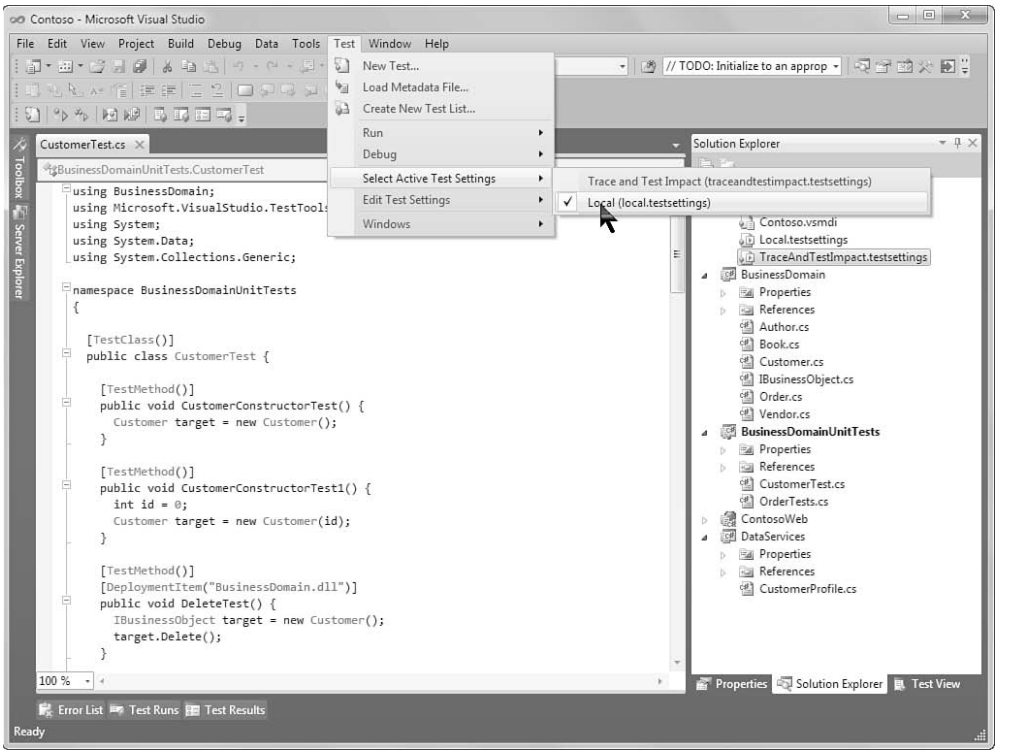
Rysunek 9.12. W oknie dialogowym Options można określić liczbę przechowywanych przebiegów testów

Kontrolowanie ustawień testów

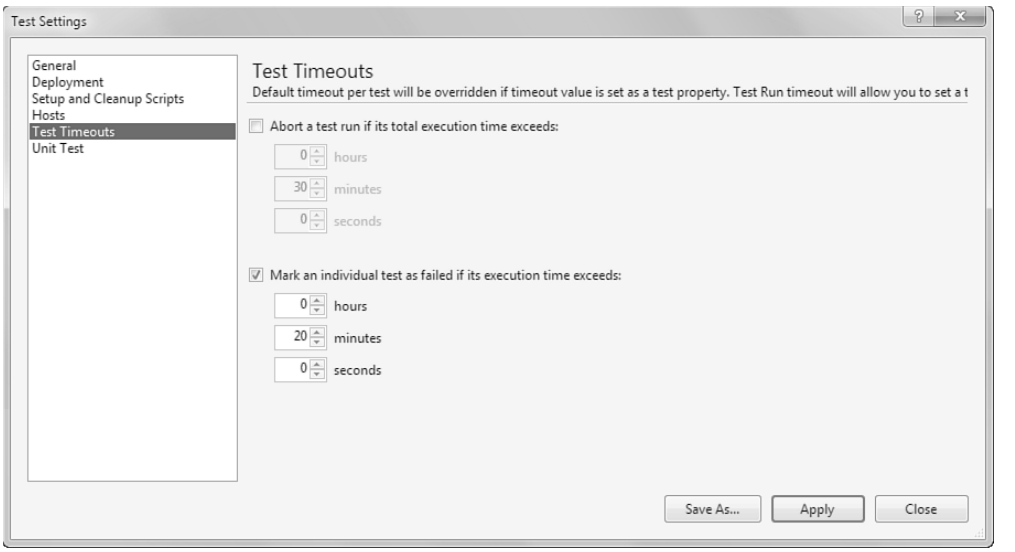
Visual Studio udostępnia plik ustawień umożliwiający kontrolowanie wykonywania testów. W większości sytuacji można pozostawić domyślną zawartość pliku ustawień. Jednak nieraz warto zmodyfikować niektóre opcje, dotyczące na przykład uruchamiania skryptów instalacyjnych w ramach testów, limitu czasu i tak dalej.

Pliki z ustawieniami testów można znaleźć na poziomie rozwiązania w oknie *Solution Explorer* (zobacz rysunek 9.2). Każdy plik ustawień testów ma rozszerzenie *.testsettings*. W dowolnym momencie za pomocą opcji *Test/Select Active Test Settings* można określić, który plik *.testsettings* ma być aktywny. Na rysunku 9.13 pokazano, jak wybrać aktywny plik ustawień.

Pliki *.testsettings* w Visual Studio to zwykłe pliki XML z konfiguracją. Kod w języku XML można bezpośrednio edytować. Visual Studio udostępnia jednak okno dialogowe do łatwiejszego modyfikowania ustawień. Okno dialogowe *Test Settings* można uruchomić przez dwukrotne kliknięcie pliku *.testsettings*. Na rysunku 9.14 pokazano to okno z wybranym węzłem *Test Timeouts*.



Rysunek 9.13. Za pomocą menu Test można wybrać aktywny plik z ustawieniami testów



Rysunek 9.14. Okno dialogowe Test Settings umożliwia zarządzanie ustawieniami związanymi z przeprowadzaniem testów (w tym limitem czasu)

Zaznaczenie węzła po lewej stronie okna dialogowego *Test Settings* pozwala skonfigurować różne ustawienia. Poniżej znajduje się przegląd opcji, którymi można zarządzać w każdym węźle okna *Test Settings*:

- ***General.*** Zawiera opcje do określania nazwy pliku ustawień, udostępniania opisu, generowania nazw dla poszczególnych przebiegów testów (na rysunku 9.10 pokazano nazwy takich przebiegów).
- ***Deployment.*** Umożliwia określenie dodatkowych plików i katalogów instalowanych w ramach przeprowadzania testów. Domyślnie testowane podzespoły są automatycznie instalowane w katalogu *bin* testu. Pozwala to przed przeprowadzeniem testów zainstalować nowe elementy (na przykład bazy danych oparte na plikach).
- ***Setup and Cleanup Scripts.*** Umożliwia określenie skryptu uruchamianego przed uruchomieniem testów lub po ich zakończeniu (albo w obu sytuacjach). Różni się to od uruchamiania kodu przed testami klas. Tu można zdefiniować kod wykonywany przed dowolną grupą testów (lub po niej).
- ***Hosts.*** Umożliwia określenie hosta, na którym można przeprowadzić testy. Zwykle jest to host domyślny (środowisko Visual Studio). Ponadto można ustawić testy tak, aby działały na serwerze środowiska ASP.NET.
- ***Test Timeouts.*** Umożliwia określenie czasu oczekiwania środowiska Visual Studio (lub innego hosta) przed porzuceniem długo działających testów. Ponadto można oznaczyć zbyt długo trwające testy jako nieudane. Na rysunku 9.12 przedstawiono przykładowe ustawienia w tym oknie dialogowym.
- ***Unit Test.*** Udostępnia zaawansowane opcje wczytywania dodatkowych podzespołów potrzebnych do przeprowadzenia testów.

Platforma testów jednostek

Platforma testów jednostek to część środowiska Visual Studio, a nie platformy .NET. System testowania jednostek w Visual Studio obejmuje: zestaw klas platformy, narzędzia i host, gdzie wykonywane są testy. Przestrzeń nazw zawierająca klasy platformy testów jednostek to `Microsoft.VisualStudio.TestTools.UnitTesting`. Dla programistów najważniejsze w tej przestrzeni nazw są klasy atrybutów i klasa `Assert`. W tym podrozdziale omawiamy podstawowe scenariusze korzystania z klas atrybutów i asercji (a także innych klas).

Klasa TestContext

Platforma testów jednostek obejmuje klasę służącą do przechowywania informacji związanych z wykonywaniem testów. Jest to klasa `TestContext`. Właściwości tej klasy służą do pobierania danych o działających testach. Informacje obejmują: ścieżkę do katalogu z testem, adres URL uruchamianego testu (jeśli są to testy jednostek ASP.NET) i informacje o wiązaniu danych (na

przykład połączenie z danymi lub obecny wiersz danych w wykonywanym teście). Wiadomości na temat kontekstu testu są przechowywane we właściwościach omawianej klasy. Kluczowe właściwości i ich zastosowania opisano w tabeli 9.1.

Tabela 9.1. *Kluczowe właściwości klasy TestContext*

Właściwość	Opis
`CurrentTestOutcome`	Umożliwia określenie wyniku ostatniego przeprowadzonego testu. Jest to przydatne w metodach oznaczonych jako `TestCleanup` (zobacz punkt „Klasy atrybutów testów").
`DataConnection`	Umożliwia wyświetlenie łańcucha połączenia używanego w testach jednostek sterowanych danymi (zobacz punkt „Tworzenie testów jednostek zależnych od danych").
`DataRow`	Umożliwia dostęp do kolumn w wierszu z danymi w czasie pracy z testami jednostek zależnych od danych (zobacz punkt „Tworzenie testów jednostek zależnych od danych").
`RequestedPage`	Zapewnia dostęp do obiektu `Page` reprezentującego żądaną stronę w testach jednostek ASP.NET (zobacz podrozdział „Pisanie testów jednostek działających w ASP.NET").
`TestDeploymentDir`	Umożliwia wyświetlenie ścieżki do katalogu, w którym testy są instalowane i uruchamiane. Jest to przydatne, jeśli trzeba zapisywać lub wczytywać dane z pliku z tego katalogu.
`TestLogsDir`	Umożliwia wyświetlenie ścieżki do katalogu, w którym zapisywane są wyniki testów.
`TestName`	Umożliwia określenie nazwy obecnie przeprowadzanego testu.

Klasa `TestContext` domyślnie jest niedostępna w kodzie. Aby uzyskać do niej dostęp, najpierw trzeba zdefiniować właściwość o tej samej nazwie i typie w klasie testu (potrzebna właściwość jest automatycznie tworzona w czasie generowania testów jednostek na podstawie istniejącego kodu). Platforma testów jednostek w momencie uruchamiania testów automatycznie tworzy egzemplarz klasy `TestContext`, a następnie sprawdza, czy w kodzie źródłowym znajduje się właściwość o tej nazwie. Jeśli ją znajdzie, przypisuje egzemplarz klasy `TestContext` do właściwości. Od tego momentu można korzystać z właściwości do pobierania informacji o kontekście przeprowadzanego testu. Poniższy kod pokazuje, jak zdefiniować właściwość `TestContext` w teście jednostek w języku C#:

```
public TestContext TestContext { get; set; }
```

Środowisko Visual Studio w czasie tworzenia obiektu typu `TestContext` generuje kompletną właściwość. Poniżej pokazano kod właściwości wygenerowany przez Visual Studio:

```
private TestContext testContextInstance;

public TestContext TestContext {
  get {
    return testContextInstance;
  }
  set {
    testContextInstance = value;
  }
}
```

Uwaga

Niektóre klasy atrybutów wymagają zdefiniowania parametru typu `TestContext` dla metod opatrzonych danym atrybutem. Dotyczy to na przykład atrybutu `ClassInitialize` (zobacz dalej). W takim scenariuszu platforma testów jednostek automatycznie przekazuje egzemplarz klasy `TestContext` do metody w momencie jej uruchamiania.

Klasy atrybutów testów

Testy jednostek są uruchamiane przez Visual Studio za pomocą hosta do przeprowadzania testów jednostek. Host musi zbadać kod, znaleźć w nim testy jednostek i uruchomić je w odpowiedni sposób. Wykorzystuje do tego atrybuty. Przypomnijmy, że atrybuty służą do udostępniania metadanych na temat kodu. Inny kod (na przykład host testów jednostek) może wykorzystać mechanizm refleksji do określenia różnych informacji na temat uruchamianego kodu.

W dotychczasowych krótkich przykładach pokazaliśmy, że testy jednostek oznaczane są za pomocą klas atrybutów zdefiniowanych w przestrzeni nazw testów. Przykładowo, klasa testu ma atrybut `TestClass`, a metodę testową należy oznaczyć atrybutem `TestMethod`. W tabeli 9.2 pokazano listę najczęściej stosowanych klas atrybutów z przestrzeni nazw testów jednostek.

Tabela 9.2. *Klasy atrybutów testów dostępne w Visual Studio*

Test	Opis
`AssemblyCleanup`	Służy do definiowania metod, które należy uruchomić po wykonaniu przez platformę wszystkich testów z danego podzespołu. Jest to przydatne, jeśli trzeba uporządkować zasoby po wykonaniu wszystkich testów. Warto zauważyć, że taki atrybut można przypisać tylko do jednej metody w podzespole.
`AssemblyInitialize`	Służy do definiowania metod, które należy uruchomić przed uruchomieniem przez platformę testów z danego podzespołu. Jest to przydatne, jeśli trzeba zainicjować zasoby dla wszystkich testów z podzespołu. Warto zauważyć, że taki atrybut można przypisać tylko do jednej metody w podzespole.
`ClassCleanup`	Służy do określania metody, którą należy uruchomić jeden raz po wykonaniu wszystkich testów z danej klasy testów. Jest to przydatne, jeśli trzeba zresetować stan systemu (na przykład w bazie danych) po zakończeniu testów.
`ClassInitialize`	Służy do określania metody, którą host powinien uruchomić jeden raz przed uruchomieniem testów z danej klasy testów. Jest to przydatne, jeśli trzeba zresetować bazę danych lub wykonać kod przygotowujący środowisko testowe. Metoda musi przyjmować parametr typu `TestContext`.
`DataSource`	Służy do udostępniania informacji o połączeniu w testach jednostek zależnych od danych („Tworzenie testów jednostek zależnych od danych").

Tabela 9.2. *Klasy atrybutów testów dostępne w Visual Studio — ciąg dalszy*

Test	Opis
`DeploymentItem`	Służy do określania dodatkowych plików (*.dll*, *.txt* i innych), które trzeba zainstalować w katalogu, gdzie przeprowadzane są testy.
`ExpectedException`	Służy do określania, że zgodnie z oczekiwaniami kod metody testowej powinien zgłosić wyjątek podanego typu. Działanie metod testowych tego rodzaju jest uważane za prawidłowe, jeśli w czasie wykonywania testu pojawi się wyjątek odpowiedniego typu (w przeciwnym razie test kończy się niepowodzeniem). Mechanizm ten jest przydatny do testowania oczekiwanych warunków wystąpienia błędów w kodzie (zobacz punkt „Testowanie wyjątków").
`HostType`	Używany do zastępowania domyślnego hosta testów (środowiska Visual Studio). Zwykle nie trzeba stosować tego atrybutu. Jeśli jednak programista pisze testy uruchamiane w procesie innego hosta (na przykład w środowisku ASP.NET), może wykorzystać ten atrybut (wraz z atrybutami `UrlToTest` i `AspNetDevelopmentServerHost`).
`Ignore`	Dodawany do metody testowej w celu określenia, że daną metodę należy pominąć, jeśli stosowany jest host do przeprowadzania testów.
`TestClass`	Służy do określania, że dana klasa jest klasą testów zawierającą jeden lub kilka testów jednostek (metod testowych).
`TestCleanup`	Służy do wskazywania metody, którą należy uruchomić jednokrotnie po wykonaniu każdej metody testowej. Wskazaną metodę można wykorzystać do uruchomienia operacji porządkujących po wykonaniu każdej metody testowej w danej klasie testów. Porządkowanie na poziomie klasy można przeprowadzić za pomocą atrybutu `ClassCleanup`.
`TestInitialize`	Służy do określania, że daną metodę należy uruchomić jednokrotnie przed wykonaniem każdej metody testowej. Jest to przydatne, jeśli trzeba zresetować stan systemu przed wykonaniem każdej metody testowej w danej klasie testów. Jeśli programista chce tylko zainicjować stan dla wszystkich metod z danej klasy testów, powinien użyć atrybutu `ClassInitialize`.
`TestProperty`	Służy do definiowania właściwości (pary nazwa – wartość) metody testowej. Informacje zapisane we właściwości można wczytać w kodzie metody testowej.
`TestMethod`	Służy do oznaczania metod w klasie testów jako testów jednostek. Metody testowe nie mogą zwracać wartości (zwracają typ `void`). Powodzenie lub niepowodzenie zależy od warunków błędów oraz asercji. Metody testowe nie mogą przyjmować parametrów, ponieważ host nie potrafi przekazywać parametrów do metod. Istnieją jednak sposoby na zasymulowanie przekazywania parametrów. Więcej informacji na ten temat zawiera punkt „Tworzenie testów jednostek zależnych od danych".
`Timeout`	Służy do określania limitu czasu (w milisekundach) dla danej metody testowej. Jeśli test będzie trwał dłużej, środowisko zatrzyma go i uzna za zakończony niepowodzeniem.

W tabeli 9.2 pokazano, że istnieje wiele klas atrybutów. Zapewniają one kontrolę nad testami jednostek. Jeśli programista pisze klasę testów dla obiektu `Customer`, może zdefiniować klasę `CustomerTest`. Na listingu 9.4 pokazano szkielet kilku testów z tej klasy. Zademonstrowano też liczne atrybuty stosowane w typowych klasach testów. Warto zwrócić uwagę na metody do inicjowania klasy testów i porządkowania zasobów po przeprowadzeniu testu.

Listing 9.4. *Przykładowa klasa testów*

```
using System;
using BusinessDomain;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Data;
using System.Collections.Generic;

namespace BusinessDomainUnitTests {

  [TestClass()]
  public class CustomerTest {

    public TestContext TestContext { get; set; }

    [ClassInitialize()]
    public static void InitTests(TestContext testContext) {
      // Wywołanie kodu resetującego testową bazę danych (Utilities.ResetTestDb();).

    }

    [ClassCleanup()]
    public static void CleanupPostTests() {
      // Wywołanie kodu resetującego testową bazę danych (Utilities.ResetTestDb();).

    }

    [TestMethod()]
    public void CustomerConstructorTest() {
      // Testuje konstruktor domyślny.

    }

    [TestMethod()]
    public void CustomerConstructorTest1() {
      // Testuje konstruktor przyjmujący identyfikator klienta.

    }

    [TestMethod()]
    [DeploymentItem("BusinessDomain.dll")]
    public void DeleteTest() {

    }

    [TestMethod()]
    [DeploymentItem("BusinessDomain.dll")]
```

```
        public void GetListTest() {

        }

        [TestMethod()]
        [DeploymentItem("BusinessDomain.dll")]
        [Timeout(5000)]

        public void SaveTest() {

        }

      }

    }
```

Następny krok polega na napisaniu kodu w każdej metodzie testowej. Kod powinien kierować wywołania do obiektu `Customer` i sprawdzać asercje związane z wynikami wywołań. Do tworzenia asercji służy opisana dalej klasa `Assert`.

Operacje wykonywane przed testami jednostek i po nich

Dobrą praktyką przy tworzeniu testów jednostek jest pisanie ich dla znanego stanu systemu. Dotyczy to bazy danych, plików i innych elementów składających się na system. W ten sposób programista może mieć pewność, że potrzebne elementy będą dostępne przy przeprowadzaniu testów. Oczywiście, same testy często naruszają stan. Testy usuwają dane, modyfikują je, dodają nowe rekordy i wykonują podobne operacje. Wtedy potrzebna jest możliwość przywrócenia stanu systemu przed wykonaniem testów (i [lub] po ich zakończeniu), aby zagwarantować stały stan na czas testów i umożliwić programistom uruchamianie ich za pomocą jednego kliknięcia (jest to następna dobra praktyka z obszaru testów jednostek).

Zwykle trzeba napisać kod utrzymujący stały stan systemu. Kod może kopiować znaną i poprawną testową bazę danych do katalogu testów (można wykorzystać do tego także atrybut `DeploymentItem`), resetować bazę danych za pomocą instrukcji w SQL-u, realizować plan generowania danych w celu utworzenia bazy, kopiować pliki lub sprawdzać inne instalowane elementy.

Kod resetujący system jest specyficzny dla środowiska. Jednak aby zagwarantować wywołanie potrzebnego kodu w czasie przeprowadzania testów, można zastosować kilka klas atrybutów: `ClassInitialize` i `ClassCleanup` lub `TestInitialize` i `TestCleanup`. Pierwsza para powoduje wykonanie kodu na początku (i na końcu) przebiegu testów dla całej klasy testów jednostek. Druga para pozwala uruchomić kod przed wykonaniem każdego testu w danej klasie testów i po ich przeprowadzeniu.

Zwykle operacje inicjujące i porządkujące wywoływane są na poziomie klasy. Jeśli programista używa klasy `Utilities` z metodą resetującą bazę danych, może zagwarantować wywołanie metody, oznaczając ją atrybutem `ClassInitialize`. Warto zauważyć, że metoda przyjmuje

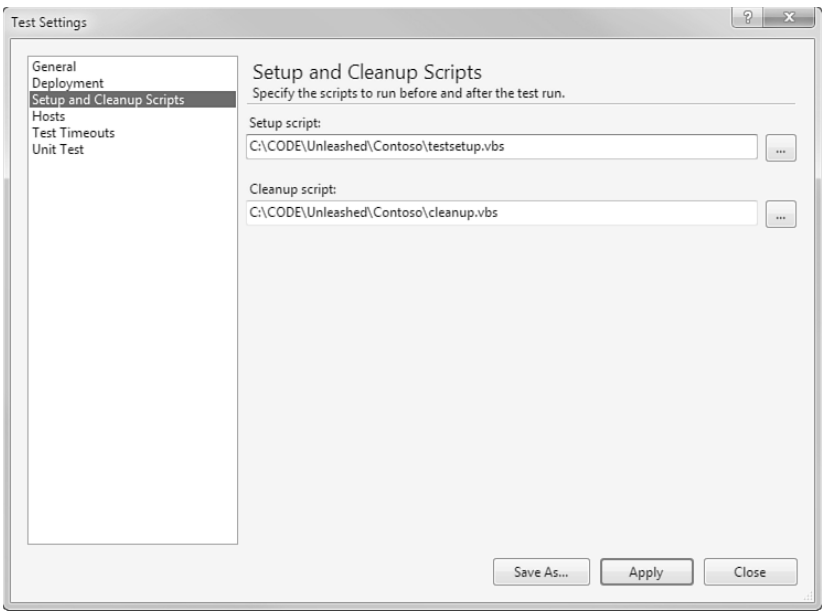
obiekt `TestContext` przekazywany przez platformę testów jednostek. Dobrą praktyką jest ponowne resetowanie systemu po wykonaniu testów jednostek. W poniższym kodzie pokazano dwie metody testów. Metody przygotowują testy i porządkują po nich.

```
[ClassInitialize()]
public static void InitTests(TestContext testContext) {
  Utilities.ResetTestDb();
}

[ClassCleanup()]
public static void CleanupPostTests() {
  Utilities.ResetTestDb();
}
```

Kontrolowanie operacji inicjujących i porządkujących na poziomie wykonywania testów

Zdarza się, że operacje inicjujące i porządkujące trzeba uruchomić na poziomie wyższym niż klasa testów. Czasem przed uruchomieniem klas testów trzeba wywołać skrypty. Wtedy można wykorzystać plik *.testsettings* (zobacz wcześniejszy punkt „Kontrolowanie ustawień testów") do zdefiniowania skryptu wywoływanego przed uruchomieniem testów i po ich zakończeniu.

Na rysunku 9.15 pokazano węzeł *Setup and Cleanup Scripts* w oknie dialogowym *Test Settings*. W tym miejscu można zdefiniować skrypty inicjujące i porządkujące wywoływane przy uruchamianiu dowolnych testów z rozwiązania.



Rysunek 9.15. Definiowanie skryptów inicjujących i porządkujących wywoływanych przy wykonywaniu dowolnych testów z rozwiązania

Klasy asercji

Przestrzeń nazw `UnitTesting` zawiera także typ statyczny `Assert`. Ten typ udostępnia liczne metody umożliwiające sprawdzenie, czy wynik testów jest zgodny z oczekiwaniami. Należy wywołać statyczne metody i sprawdzić wartość warunku (`true` lub `false`). Jeśli warunek logiczny nie jest spełniony, działanie asercji kończy się niepowodzeniem. Same asercje nie zwracają wyników. Zamiast tego w czasie wykonywania programu automatycznie powiadamiają platformę testów jednostek o sukcesie lub niepowodzeniu.

Można na przykład napisać test jednostki wczytujący znany rekord z bazy danych. Następnie można dodać asercję dotyczącą tego rekordu, aby udowodnić, że program potrafi pobrać dane z bazy, prawidłowo wywołać odpowiednie zbiory danych i dotrzeć do konkretnego obiektu. Poniżej pokazano prostą asercję do sprawdzania, czy dwie zmienne mają tę samą wartość. Jeśli wartości są sobie równe (`AreEqual`), działanie asercji kończy się powodzeniem. Jeżeli wartości są różne, asercja zgłasza niepowodzenie, a platforma testów jednostek oznacza test jako nieudany.

```
Assert.AreEqual(cust.Id, customerId);
```

`AreEqual` to tylko jedna z wielu metod asercji dostępnych w klasie `Assert`. Większość metod asercji dotyczy jednego zagadnienia — porównywania dwóch wartości. W tabeli 9.3 przedstawiono bardziej kompletną listę metod.

Tabela 9.3. *Asercje testów*

Test	Opis
`AreEqual` i `AreNotEqual`	Służy do sprawdzania, czy dwie wartości są sobie równe, czy nie.
`AreSame` i `AreNotSame`	Pozwala sprawdzić, czy dwa obiekty są tym samym obiektem, czy nie.
`Fail`	Metoda `Assert.Fail` oznacza warunek jako niespełniony bez jego sprawdzania. Może to być przydatne przy wartościowaniu warunku za pomocą operacji logicznych i dojściu do miejsca w kodzie, gdzie operacje reprezentują test zakończony niepowodzeniem.
`Inconclusive`	Służy do oznaczania, że test nie zakończył się ani niepowodzeniem, ani powodzeniem (wynik jest niejednoznaczny).
`IsInstanceOfType` i `IsNotInstanceOfType`	Pozwala sprawdzić, czy obiekt jest określonego typu, czy nie.
`IsNull` i `IsNotNull`	Pozwala określić, czy obiekt zawiera referencję `null`, czy nie.
`IsTrue` i `IsFalse`	Służy do sprawdzania, czy warunek jest spełniony, czy nie.

Każda metoda asercji ma kilka przeciążonych wersji. Umożliwiają one porównywanie różnych typów danych — łańcuchów znaków, liczb, obiektów i kolekcji ogólnych. Ponadto dostępne są wersje, które pozwalają po prostu zastosować asercję lub dodatkowo wyświetlić komunikat w sytuacji, kiedy asercja nie jest spełniona. Na przykład poniżej zapisano tę samą asercję, co wcześniej, jednak ta wersja wyświetla komunikat o błędzie, jeśli warunek nie jest spełniony.

```
Assert.AreEqual(cust.Id, customerId, "Identyfikatory klienta są różne.");
```

Przyjrzyjmy się kilku przykładowym metodom, których szkielet pokazano na wcześniejszym listingu (zobacz listing 9.4). Należy uzupełnić te metody przez napisanie kodu korzystającego z klasy `Customer`, a następnie poczynić asercje na temat wyników. Listing 9.5 zawiera przykładowy test konstruktora klasy `Customer`. Warto zauważyć, że testujemy znany rekord danych przez wczytanie egzemplarza klasy `Customer` na podstawie identyfikatora rekordu. Następnie sprawdzamy wyniki w klasie `Assert`.

Listing 9.5. *Przykładowy test jednostek ilustrujący korzystanie z klasy Assert*

```
[TestMethod()]
public void CustomerConstructorTest1() {

  // Testuje konstruktor przyjmujący identyfikator klienta.

  int customerId = 1; // Testowanie danych klienta.
  Customer cust = new Customer(customerId);

  // Sprawdzanie, czy dla testowego klienta zwrócono poprawne dane.
  // Sprawdzanie operacji ustawiania i wczytywania wartości właściwości.
  Assert.AreEqual(cust.Id, customerId,
    "Identyfikatory klienta są różne.");

  Assert.AreEqual(cust.Email, "jdevlop@contoso.com");
  Assert.AreEqual(cust.Name, "Jan Programista");
  Assert.AreEqual(cust.City, "Brzeg");
  Assert.AreEqual(cust.PostalCode, "OOOOO");
  Assert.AreEqual(cust.RegionState, "DL");
  Assert.AreEqual(cust.Address1, "ul. Microsoftu 100");
  Assert.IsTrue(cust.Address2 == "");
  Assert.IsTrue(cust.Phone == "");
}
```

Klasa `Assert` zawiera też wersje metod `AreEqual` i `AreNotEqual` dla ogólnych typów danych. Wersje te umożliwiają sprawdzanie, czy dwa typy ogólne są takie same. Programista musi określić typ ogólny za pomocą standardowej składni, `<T>` [lub `(of T)` w języku Visual Basic], i przekazać dwie porównywane zmienne. Poniżej pokazano przykładowy kod:

```
Assert.AreEqual<Customer>(cust1, cust2);
```

Sprawdzanie kolekcji obiektów

Przestrzeń nazw `UnitTesting` obejmuje też klasę asercji `CollectionAssert`. Przy jej użyciu można sprawdzić zawartość klas kolekcji. Można na przykład wywołać metodę `Contains`, aby ustalić, czy dana kolekcja zawiera konkretny element (lub go nie zawiera). Metoda `AllItems ↪AreInstancesOfType` służy do sprawdzania, czy kolekcja zawiera egzemplarze tego samego typu. Można też sprawdzić, czy dwie kolekcje są takie same (`AreEqual` i `AreNotEqual`) lub czy zawierają identyczne elementy, ale niekoniecznie w tej samej kolejności (`AreEquivalent` i `AreNotEquivalent`).

Sprawdzanie łańcuchów znaków

Klasa `StringAssert` zawiera metody służące do sprawdzania łańcuchów znaków i ich fragmentów. Metoda `Contains` pozwala ustalić, czy łańcuch obejmuje określony podłańcuch. Metoda `StartsWith` umożliwia określenie, czy łańcuch rozpoczyna się od podanych znaków, a metoda `EndsWith` sprawdza końcową część łańcucha. Metody `Matches` i `DoesNotMatch` umożliwiają ustalenie, czy łańcuch znaków pasuje do zdefiniowanego wyrażenia.

Testowanie wyjątków

Testy jednostek należy pisać w taki sposób, aby ustalić, czy kod działa w oczekiwany sposób zarówno w korzystnych, jak i niekorzystnych warunkach. Korzystne warunki można sprawdzić w opisany wcześniej sposób za pomocą metod klasy `Assert`. Jednak często trzeba określić, czy kod zgłasza odpowiedni wyjątek w odpowiedzi na specyficzne wywołanie. Można wtedy zastosować atrybut `ExpectedException` do przetestowania określonych warunków wystąpienia błędów.

Atrybut `ExpectedException` należy dodać do metody testowej. Atrybut przyjmuje jako parametr typ oczekiwanego wyjątku. Jeśli wywołanie metody testowej spowoduje zgłoszenie wyjątku odpowiedniego typu, test zakończy się powodzeniem. Jeżeli wyjątek nie wystąpi lub będzie miał niewłaściwy typ, test zakończy się niepowodzeniem.

Załóżmy, że programista chce sprawdzić, co stanie się przy próbie utworzenia nowego klienta przy użyciu niepełnych danych. Może w tym celu napisać kod zgłaszający niestandardowy wyjątek typu `InvalidCustomerException` i dodać do metody testowej następujący atrybut:

```
[TestMethod()]
[ExpectedException(typeof(InvalidCustomerException),
  "Nie zgłoszono wyjątku InvalidCustomerException.")]
public void NewCustomerTest() {

  // Tworzenie nieprawidłowego nowego egzemplarza klasy Customer.
}
```

Warto zauważyć, że jeśli powyższy kod nie zgłosi wyjątku, jako wynik testu przekazany zostanie komunikat o błędzie (`"Nie zgłoszono wyjątku InvalidCustomerException."`). Komunikat to opcjonalny parametr atrybutu `ExpectedException`.

Można łączyć asercje z atrybutem `ExpectedException`. Aby metoda testowa w takich warunkach przeszła test, asercje muszą być spełnione, a metoda musi zgłosić wyjątek.

Uwaga

Zgłoszony wyjątek musi mieć dokładnie taki sam typ, jak oczekiwany. Wyjątek nie może na przykład dziedziczyć po oczekiwanym. Wtedy test zakończy się niepowodzeniem.

Tworzenie testów jednostek zależnych od danych

Często lepiej jest napisać jeden test jednostek i uruchomić go wielokrotnie dla różnych wartości parametrów, niż przygotowywać wiele podobnych testów. Cechą dobrych testów jednostek jest pokrycie różnych warunków. Oczywiście, testy jednostek nie przyjmują parametrów. Utrudnia to przekazywanie danych do testów. Dlatego trzeba powiązać testy jednostek z danymi. Następnie można nakazać platformie testów jednostek uruchomienie testu dla każdego wiersza danych. Można też wykorzystać powiązane dane testowe w samym teście jednostek i wywołać kod na wiele sposobów.

Łączenie się z danymi

Do wiązania testów jednostek z danymi testowymi służy klasa atrybutu `DataSource` z platformy testów jednostek. Klasa umożliwia bezpośrednie podanie łańcucha połączenia z danymi testowymi lub określenie nazwy łańcucha połączenia zapisanego w pliku konfiguracji projektu. Obie techniki pozwalają nawiązać połączenie z bazą danych (na przykład SQL Server lub Access), plikiem *.csv* (zwykle tworzonym w Notatniku lub Excelu) lub plikiem *.xml*.

Klasa atrybutu `DataSource` ma trzy wersje. Pierwsza przyjmuje jeden parametr, `dataSource` ↪`SettingName`. Wymaga to przekazania nazwy ustawień źródła danych zdefiniowanych w pliku konfiguracji. Druga wersja przyjmuje parametry `connectionString` i `tableName`. Należy wtedy przekazać poprawny łańcuch połączenia ze źródłem danych i określić nazwę tabeli, którą system ma powiązać z testem jednostek. Ostatnia wersja przyjmuje parametry `providerInvariantName`, `connectionString`, `tableName` i `dataAccessMethod`. Pierwszy służy do określania typu dostawcy, na przykład dostawcy plików CSV, bazy SQL Server i tak dalej. Drugi, łańcuch połączenia, zależy od wybranego dostawcy i określa sposób dostępu do danych. Trzeci parametr to nazwa tabeli (lub pliku) z danymi. Czwarty, metoda dostępu do danych, określa sposób powiązania danych z testem jednostek — sekwencyjny lub swobodny.

Załóżmy, że programista pisze test jednostek do sprawdzenia informacji o zamówieniach i koncie klienta. Może wczytać dane klienta, pobrać złożone zamówienia, określić przyznany kredyt i sprawdzić obecny stan rachunku. Wszystkie te operacje dla danego klienta można wykonać w jednym teście jednostek. Test można powiązać z wierszami danych testowych klienta, aby sprawdzić poprawność kodu w różnych scenariuszach. Przyjmijmy, że dane testowe znajdują się w pliku *.csv* w aplikacji testowej. Wtedy atrybut `DataSource` można dodać do metody testowej w następujący sposób:

```
[TestMethod(),
DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
  "|DataDirectory|\\OrderTestData.csv", "OrderTestData#csv",
  DataAccessMethod.Sequential),
DeploymentItem("BusinessDomainUnitTests\\OrderTestData.csv")]
public void VerifyCustomerOrders() {

  // Kod sprawdzający informacje o zamówieniach i koncie klienta.

}
```

Warto zauważyć, że w przykładowym kodzie pierwszy parametr atrybutu `DataSource` określa dostawcę plików *.csv*. Następny parametr to łańcuch połączenia z plikiem danych. Trzeci parametr (`OrderTestData#csv`) informuje, że nazwa tabeli nie istnieje (użyto nazwy pliku). Ostatni parametr, wartość wyliczeniowa `DataAccessMethod.Sequential`, określa, że każdy wiersz należy powiązać z testem jednostek w sposób sekwencyjny. Zwróćmy też uwagę na to, że dodano atrybut `DeploymentItem`, aby zagwarantować, że plik z danymi testowymi zostanie zainstalowany wraz z testem jednostek.

Nawiązywanie połączenia przy użyciu pliku konfiguracyjnego

W innym scenariuszu można przechowywać informacje o połączeniu w pliku konfiguracyjnym. Jest to przydatne, jeśli programista nie chce na trwałe zapisywać informacji w kodzie testów jednostek. Możliwe, że ta sama testowa baza danych jest wykorzystywana w wielu testach. W środowisku programistycznym i testowym połączenie z bazą może wyglądać inaczej. Wtedy umieszczenie informacji o połączeniu w pliku konfiguracyjnym ułatwia zarządzanie nimi.

Pierwszy krok w tym podejściu polega na utworzeniu pliku konfiguracyjnego. W tym celu wystarczy dodać plik *app.config* do projektu testów jednostek (należy kliknąć plik projektu prawym przyciskiem myszy i wybrać opcję *Add/New Item*). Jedyna trudność to prawidłowe określenie formatu pliku konfiguracyjnego. Trzeba dodać sekcję konfiguracyjną specyficzną dla narzędzi testowych środowiska Visual Studio. Warto zauważyć, że w sekcji należy wskazać wersję 10. platformy testów jednostek. Oto przykładowy kod:

```
<configSections>
  <section name="microsoft.visualstudio.testtools"

type="Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection,
Microsoft.VisualStudio.QualityTools.UnitTestFramework, Version=10.0.0.0,
↪Culture=
neutral, PublicKeyToken=b03f5f7f11d50a3a" />
</configSections>
```

Następnie należy dodać sekcję z łańcuchem połączenia. Trzeba określić nazwę łańcucha i podać informacje o połączeniu oraz dostawcy danych. Tekst łańcucha zależy oczywiście od dostawcy i rodzaju danych. Dla danych przechowywanych w plikach *.xlsx*, *.csv* i *.xml* oraz bazach Access, Oracle i SQL potrzebne są specyficzne elementy w łańcuchu połączenia. W poniższym przykładzie wykorzystano dostawcę `OleDb` do utworzenia łańcucha połączenia dla pliku *.csv*:

```
<connectionStrings>
  <add name="OrderTestDataCnn"
    connectionString="Provider=Microsoft.Jet.OLEDB.4.0; Data Source=
|DataDirectory|TestData\; Extended Properties='text;HDR=Yes;FMT=Delimited'"
    providerName="System.Data.OleDb" />
</connectionStrings>
```

Ostatni krok polega na utworzeniu sekcji źródła danych zagnieżdżonej w sekcji narzędzi testowych. Należy określić nazwę testowego źródła danych i wskazać łańcuch połączenia. Ponadto można podać metodę dostępu do danych (dostęp sekwencyjny lub swobodny) i nazwę tabeli (jeśli jest potrzebna). Ostatecznie plik konfiguracyjny powinien wyglądać tak jak na listingu 9.6.

Listing 9.6. *Przykładowy plik konfiguracyjny testu jednostek*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="microsoft.visualstudio.testtools"

type="Microsoft.VisualStudio.TestTools.UnitTesting.TestConfigurationSection,
Microsoft.VisualStudio.QualityTools.UnitTestFramework, Version=10.0.0.0,
↪Culture=
neutral, PublicKeyToken=b03f5f7f11d50a3a" />
  </configSections>
  <connectionStrings>
    <add name="OrderTestDataCnn"
      connectionString="Provider=Microsoft.Jet.OLEDB.4.0; Data Source=
|DataDirectory|TestData\; Extended Properties='text;HDR=Yes;FMT=Delimited'"
      providerName="System.Data.OleDb" />
  </connectionStrings>
  <microsoft.visualstudio.testtools>
    <dataSources>
      <add name="OrderTestDataSource"
        connectionString="OrderTestDataCnn" dataTableName="OrderTestData#csv"
        dataAccessMethod="Sequential"/>
    </dataSources>
  </microsoft.visualstudio.testtools>

</configuration>
```

Uwaga

Warto zauważyć, że w jednym pliku konfiguracyjnym można zdefiniować wiele łańcuchów połączeń (każdy z nich musi mieć inną nazwę).

Następnie można wykorzystać skonfigurowane testowe źródło danych w metodzie testowej. W tym celu wystarczy podać nazwę źródła danych w atrybucie `DataSource` metody testowej. Prowadzi to do zwiększenia przejrzystości kodu, ułatwia wielokrotne wykorzystanie łańcuchów połączeń i umożliwia centralne zarządzanie połączeniem, jeśli jest ono inne na poszczególnych komputerach. Poniżej pokazano, jak wykorzystać źródło danych:

```
[TestMethod(),
DataSource("OrderTestDataSource"]
public void VerifyCustomerOrders() {

  // Kod sprawdzający informacje o zamówieniach i koncie klienta.

}
```

Warto zauważyć, że w tym przykładzie dane testowe znajdują się w pliku *.csv* o nazwie *OrderTestData.csv*. Plik dodano do katalogu *TestData* w projekcie testów. Katalog i plik są następnie przenoszone do katalogu *bin*, w którym środowisko uruchamia testy. Dzieje się tak, ponieważ

łańcuch połączenia obejmuje fragment `|DataDirectory|TestData\`. Można też bezpośrednio wymusić instalowanie plików, dodając atrybut `DeploymentItem` do metody testowej.

Wskazówka

Łańcuchy połączeń dla testów jednostek można skonfigurować za pomocą okna dialogowego *Properties*. Zobacz na przykład punkt „Konfigurowanie atrybutów testów jednostek" w dalszej części rozdziału.

Stosowanie danych testowych w asercjach

Dostęp do danych testowych powiązanych z metodami testowymi można uzyskać za pomocą obiektu kontekstu testu. Przypomnijmy, że jeśli programista ustawi właściwość typu `TestContext` (zobacz punkt „Klasa TestContext"), obiekt kontekstu zostanie udostępniony przez platformę. Metoda `DataRow` klasy `TestContext` pozwala pobrać wiersz powiązany z wykonywanym testem. Do metody można przekazać nazwę pola lub indeks, aby pobrać dane z odpowiedniej kolumny. Zwykle należy ustawić zmienne dla wszystkich pól danych testowych. Następnie można wykorzystać zmienne w asercjach w testach jednostek.

Na listingu 9.7 pokazano przykład zastosowania danych testowych (układ danych testowych przedstawiono na rysunku 9.18 w dalszej części rozdziału). Warto zauważyć, że każdą kolumnę trzeba zrzutować lub przekształcić na odpowiedni typ danych. Następnie można napisać właściwe asercje.

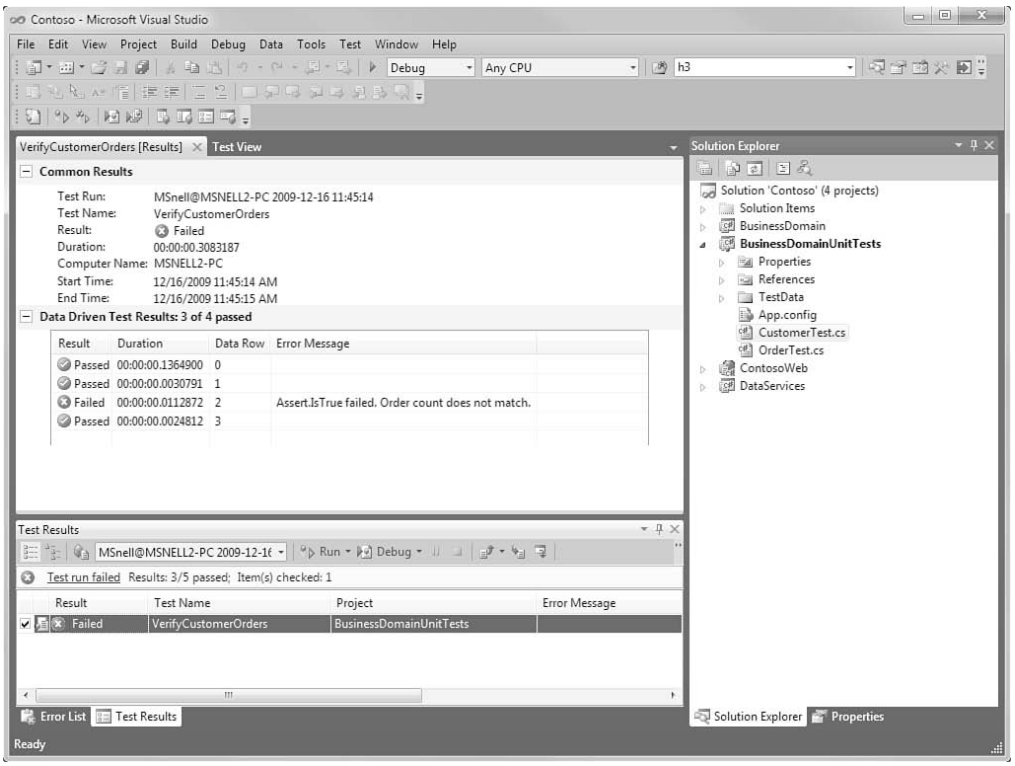
Listing 9.7. *Przykładowy test jednostek, w którym wykorzystano powiązane dane*

```
[TestMethod(),
DataSource("OrderTestDataSource")]
public void VerifyCustomerOrders() {

  // Pobieranie wartości z wierszy danych.
  string email = (string)TestContext.DataRow["email"];
  int numOrders = (int)TestContext.DataRow["num_orders_todate"];
  double creditLimit = Convert.ToDouble(TestContext.DataRow["credit_limit"]);
  double accountBalance =
   ↪Convert.ToDouble(TestContext.DataRow["account_balance"]);

  // Tworzenie obiektu Customer na podstawie adresu e-mail.
  Customer cust = new Customer(email);

  // Sprawdzanie informacji o zamówieniach i koncie klienta.
  Assert.IsTrue(cust.OrderHistory.Count == numOrders,
    "Liczba zamówień jest nieprawidłowa.");
  Assert.IsTrue(cust.Account.Balance == accountBalance,
    "Stan rachunku jest nieprawidłowy.");
  Assert.IsTrue(cust.Account.CreditLimit == creditLimit,
    "Limit debetu jest nieprawidłowy.");
```

```
        Assert.IsTrue(cust.Account.CreditLimit > cust.Account.Balance,
          "Limit debetu musi być mniejszy niż stan rachunku.");

    }
```

Uruchamianie testów zależnych od danych i wyświetlanie ich wyników

Testy jednostek zależne od danych można uruchamiać w taki sam sposób jak inne testy jednostek. Główna różnica polega na tym, że cały test jednostek kończy się niepowodzeniem, jeśli test zakończy się w ten sposób dla choćby jednego wiersza danych. Można dwukrotnie kliknąć wynik nieudanego testu, aby wyświetlić szczegółowe informacje na jego temat. Wyniki testów zależnych od danych pojawiają się w odrębnym okienku pod standardowymi wynikami testu jednostek. Przykładowe dane pokazano na rysunku 9.16.



Rysunek 9.16. Wyniki testów jednostek zależnych od danych wyświetlane są pod standardowymi wynikami całego testu jednostek

Warto zauważyć, że dla trzech z czterech wierszy testy jednostek zakończyły się powodzeniem. Można zobaczyć komunikat o błędzie (z asercji) dla wiersza, który spowodował błąd. Dwukrotnie kliknięcie tego wiersza spowoduje wyświetlenie szczegółowych informacji o niepowodzeniu, w tym śladu stosu błędów i odnośnika do wiersza kodu, gdzie wystąpił problem.

Pisanie testów jednostek działających w ASP.NET

Można wybrać, czy testy jednostek mają być przechowywane i uruchamiane w środowisku ASP.NET zamiast w domyślnym procesie hosta VSTest. Jest to przydatne przy pisaniu testów współdziałających z obiektami ASP.NET, takimi jak `Page` i `Session`. Czasem programista chce przetestować usługę sieciową lub inny kod aplikacji związany z interfejsem sieciowym. Można wtedy umieścić testy jednostek dla aplikacji ASP.NET na serwerze IIS lub na serwerze środowiska Visual Studio. Takie rozwiązanie gwarantuje, że testy będą miały pełny dostęp do zmiennych środowiskowych ASP.NET.

Definiowanie atrybutów środowiska ASP.NET

Aby określić, że test jednostek ma działać w środowisku ASP.NET, należy dodać odpowiednie atrybuty do metody testowej. Liczne atrybuty znajdują się w przestrzeni nazw `UnitTesting.Web`. Dlatego należy najpierw dodać odpowiednią instrukcję `using` (`imports` w języku Visual Basic) w górnej części pliku klasy testów, które mają działać w ASP.NET. Instrukcja powinna wyglądać tak:

```
using Microsoft.VisualStudio.TestTools.UnitTesting.Web;
```

Ponadto w aplikacji z testem jednostek należy dodać referencję do witryny. Umożliwi to dostęp do stron witryny i innych klas, które mogą znajdować się w katalogu *App_Code* oraz innych miejscach.

Przy tworzeniu testów jednostek w ASP.NET używane są trzy główne atrybuty: `UrlToTest`, `HostType` i `AspNetDevelopmentServerHost`. Atrybut `UrlToTest` umożliwia określenie strony, którą należy wywołać w czasie uruchamiania danego testu jednostek. Strona jest wywoływana przez platformę testów, a dla testu jednostek dostępny jest kontekst żądania sieciowego (poprzez obiekt `Page`). W kodzie testu jednostek można korzystać ze środowiska ASP.NET w taki sam sposób, jak w pliku z kodem ukrytym strony internetowej.

Atrybut `HostType` umożliwia zmianę typu hosta na ASP.NET. W tym celu należy przekazać do atrybutu łańcuch znaków `ASP.NET`.

Przy korzystaniu z hosta w postaci serwera IIS wystarczy ustawić atrybuty `UrlToTest` i `HostType`. Jeśli jednak programista używa serwera ASP.NET Development (współdziałającego z Visual Studio), musi ponadto dodać atrybut `AspNetDevelopmentServerHost`. Jako parametr atrybutu należy przekazać ścieżkę do aplikacji sieciowej. Trzeba też podać nazwę głównego katalogu aplikacji sieciowej.

Na listingu 9.8 pokazano przykład zastosowania wszystkich trzech wymienionych atrybutów do zdefiniowania testu jednostek ASP.NET działającego na lokalnym serwerze rozwojowym.

Listing 9.8. *Test jednostek ASP.NET*

```
[TestMethod()]
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("C:\\CODE\\Unleashed\\Contoso\\ContosoWeb", "/")]
[UrlToTest("http://localhost:55136/ShoppingCart.aspx")]
public void AddShoppingCartItemTest() {

  // Pobieranie żądanej strony.
  Page reqPage = TestContext.RequestedPage;
  Assert.IsTrue(reqPage.Title == "Koszyk zakupów", "Tytuł strony jest
  ↪niewłaściwy.");

  // Rzutowanie na typ strony.
  ShoppingCartPage actualPage = (ShoppingCartPage)reqPage;
  Assert.IsNotNull(actualPage.Cart, "Na stronie nie ma koszyka.");

  // Sprawdzanie poprawności kodu do obsługi koszyka zakupów.
  actualPage.Cart.AddItem("Produkt 1", 1, 12.75);
  actualPage.Cart.AddItem("Produkt 2", 2, 26.95);
  Assert.IsTrue(actualPage.Cart.Items.Count == 2, "Liczba elementów jest
  ↪niewłaściwa.");
  Assert.AreEqual(actualPage.Cart.CalculateCartTotal(), 66.65);

}
```

Warto zauważyć, że referencje do obiektów ASP.NET pochodzą z właściwości `RequestedPage` obiektu `TestContext`. Właściwości tej można użyć do zrzutowania danych bezpośrednio na typ żądanej strony (tu jest to typ `ShoppingCartPage`). Oczywiście, właściwość `RequestedPage` ma typ `System.Web.UI.Page`, dlatego zapewnia dostęp do obiektów `Server`, `Session`, `Request`, `Response` i podobnych.

Generowanie testów jednostek ASP.NET

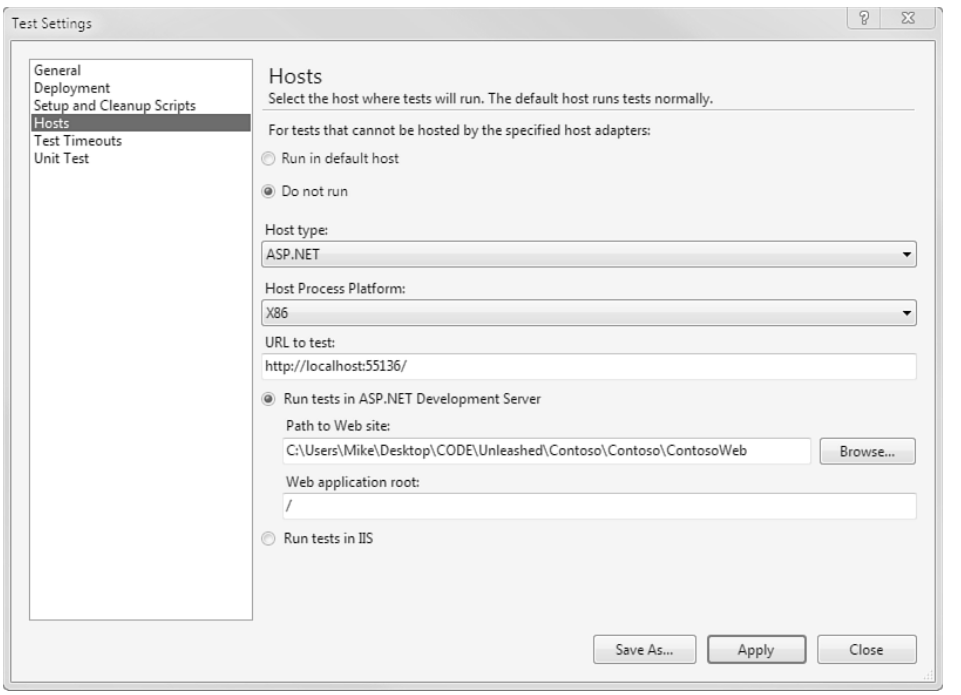
Za pomocą Visual Studio można automatycznie wygenerować testy jednostek ASP.NET. Często jest to zalecana metoda tworzenia testów, ponieważ przygotowuje projekt testów do pracy ze środowiskiem ASP.NET i automatycznie konfiguruje atrybuty.

Testy jednostek ASP.NET generuje się w taki sam sposób, jak wszystkie inne testy jednostek. Najpierw należy otworzyć kod (plik z kodem ukrytym strony lub inny plik klasy). Następnie trzeba kliknąć prawym przyciskiem myszy kod i wybrać opcję *Create Unit Tests*. Spowoduje to uruchomienie kreatora *Create Unit Tests* (zobacz rysunek 9.5). W tym miejscu można wybrać metody, dla których kreator ma wygenerować testy. Obsługiwane są zdarzenia związane ze stroną, kliknięcie przycisków i inne zdarzenia kontrolek. Można też napisać test jednostek wyszukujący kontrolki na stronie i sprawdzający ich zawartość.

Konfigurowanie hosta projektu testów

Programiści często tworzą dla witryny odrębny projekt testów jednostek. Projekt może dotyczyć testów powiązanych z funkcjami witryny. Można też utworzyć dodatkowe projekty testów do sprawdzania innych bibliotek kodu. Tego typu podział ułatwia pisanie i przeprowadzanie testów jednostek.

Ponadto można skonfigurować ustawienia projektu testów specyficznie pod kątem hosta. Dzięki temu nie trzeba definiować licznych atrybutów ASP.NET w każdym teście jednostek. Zamiast tego można skonfigurować plik *.testsettings* w taki sposób, aby współdziałał ze specyficznym hostem. W tym celu należy kliknąć dwukrotnie plik *.testsettings* (*Solution Items/Local.testsettings*), co spowoduje wyświetlenie okna dialogowego *Test Settings*, widocznego na rysunku 9.17.



Rysunek 9.17. Plik .testsettings można wykorzystać do skonfigurowania całego projektu testów, tak aby działał dla witryny ASP.NET

Wskazówka

Jeśli rozwiązanie zawiera wiele projektów testów jednostek, warto utworzyć kilka wersji pliku *.testsettings*. Następnie można łatwo przełączać się między konfiguracjami dla hostów ASP.NET i standardowego (VSTest).

W oknie dialogowym należy otworzyć widoczny po lewej stronie węzeł *Hosts*, a następnie ustawić wartość *ASP.NET* dla opcji *Host type*. Spowoduje to udostępnienie kilku dodatkowych opcji: *URL to Test*, *Path to Web Site* i *Web Application Root*. Każdą z tych opcji można ustawić na wartości domyślne dla całego projektu testów jednostek.

Konfigurowanie atrybutów testów jednostek

Pokazaliśmy, że istnieje wiele atrybutów, które można zastosować do testów jednostek. Oczywiście, możliwe jest skonfigurowanie każdego z nich w edytorze kodu. Jednak Visual Studio udostępnia pewną pomoc w oknie dialogowym *Properties*. Można wybrać dany test jednostek, wyświetlić okno *Properties*, a następnie skonfigurować atrybuty stosowane do metody testowej. Visual Studio wygeneruje kod atrybutów dla metody testowej. Przyjrzyjmy się przykładowi.

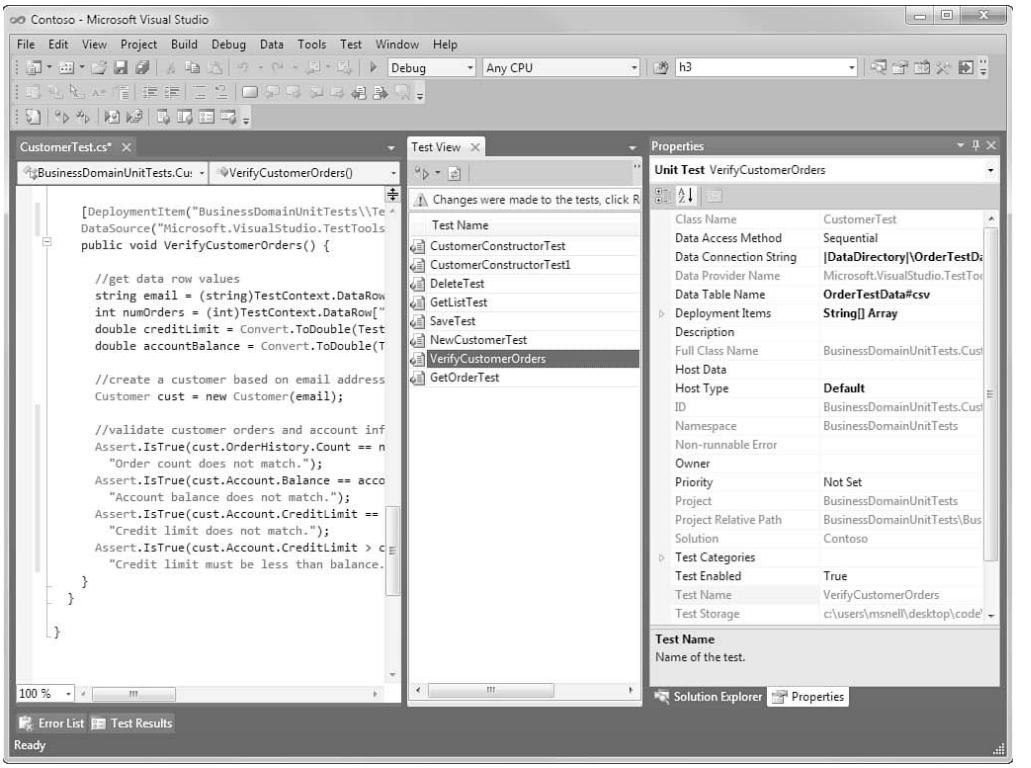
Definiowanie połączenia za pomocą okna dialogowego Properties

Pierwszym krokiem przy stosowaniu okna *Properties* do konfigurowania testów jednostek jest wybranie testu. Należy to zrobić w oknie *Test View* (*Test/Windows/Test View*). Następnie można kliknąć prawym przyciskiem myszy test na liście i wybrać opcję *Properties*, aby wyświetlić okno *Properties* dla danego testu. Na rysunku 9.18 pokazano przykładowe dane.

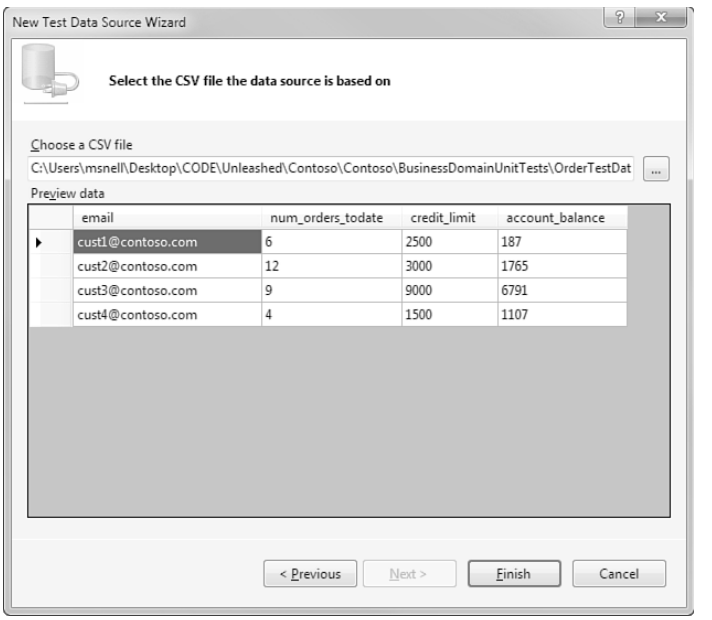
Warto zauważyć, że w oknie *Properties* znajdują się właściwości wielu atrybutów, które można zastosować do metody testowej. Zmiana wartości jednej z właściwości spowoduje wygenerowanie przez środowisko Visual Studio atrybutu i kodu w pliku z kodem (między kodem i omawianym oknem następuje dwustronna synchronizacja).

Przykładowo, właściwość *Data Connection String* ułatwia zdefiniowanie atrybutu `DataSource`. Kliknięcie przycisku z wielokropkiem przy właściwości powoduje uruchomienia kreatora, dzięki któremu można zdefiniować łańcuch połączenia z bazą danych SQL, plikiem CSV lub plikiem XML (oczywiście, można nawiązać połączenie także z innymi źródłami danych, ale wymaga to napisania własnego łańcucha połączenia). We wcześniejszym przykładzie używaliśmy pliku CSV. Po zaznaczeniu opcji związanej z tym formatem można przejść do pliku CSV, a nawet podejrzeć zapisane w nim dane. Na rysunku 9.19 pokazano przykładowy ekran kreatora. Po ustawieniu wszystkich opcji kreator zaktualizuje kod, dodając atrybuty `DataSource` i `DeploymentItem`. Poniżej znajduje się uzyskany kod konfiguracyjny:

```
[TestMethod(),
  DeploymentItem("BusinessDomainUnitTests\\TestData\\OrderTestData.csv"),
  DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
    "|DataDirectory|\\OrderTestData.csv", "OrderTestData#csv",
  DataAccessMethod.Sequential)]
public void VerifyCustomerOrders() { ...
```



Rysunek 9.18. Można zaznaczyć test w oknie Test View, a następnie skonfigurować jego atrybuty w oknie Properties



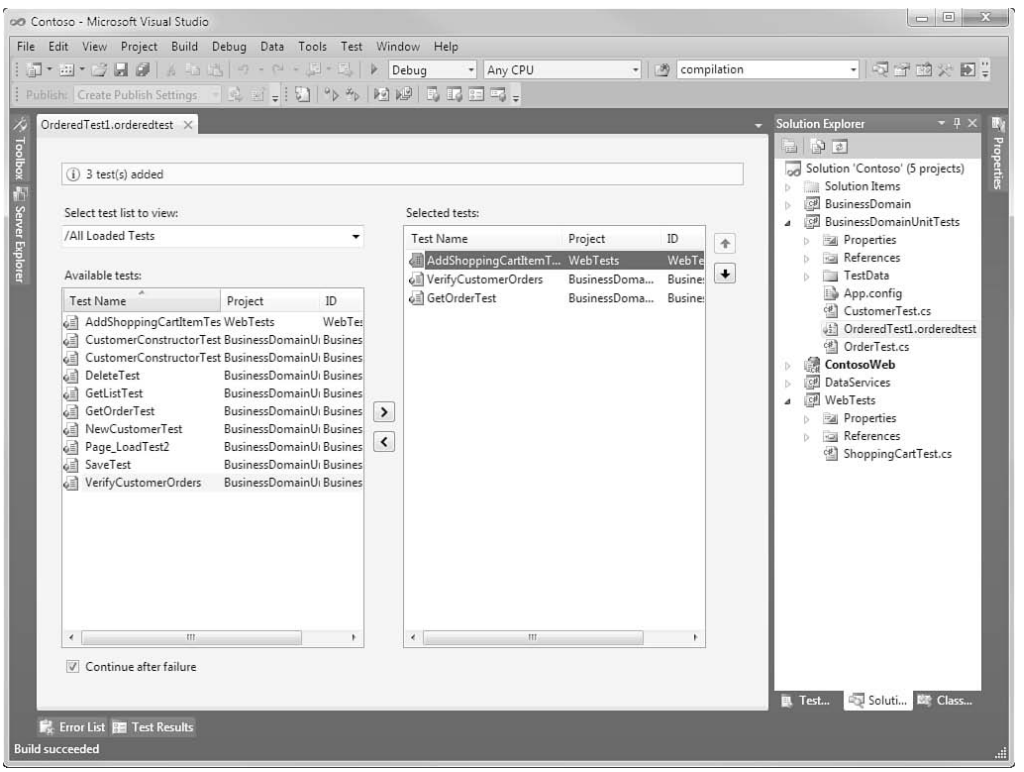
Rysunek 9.19. Testy jednostek można powiązać z testowymi danymi, w tym z prostym plikiem .csv

Tworzenie testów uporządkowanych

Visual Studio umożliwia łączenie grup testów jednostek, określanie kolejności ich wykonywania i traktowanie wyników w taki sposób, jakby uzyskano je dla pojedynczego testu. Może to być przydatne, jeśli programista chce napisać testy jednostek zależne od siebie. Można na przykład wstawić rekord w jednym teście i oczekiwać, że ten sam rekord będzie dostępny w dalszym teście. Oczywiście, jest to niezgodne z dobrymi praktykami testów jednostek — możliwe powinno być przeprowadzenie każdego testu niezależnie od pozostałych. Na szczęście można utworzyć test uporządkowany, który łączy poszczególne testy jednostek w nowy, niezależny test.

Test uporządkowany można dodać do projektu testów, klikając projekt prawym przyciskiem myszy i wybierając opcję *Add/Ordered Test*. Można też wybrać szablon *Ordered Test* w oknie dialogowym *Add New Test* (zobacz rysunek 9.3).

Test uporządkowany to po prostu plik XML oparty na szablonie *Ordered Test*. Nie trzeba jednak ręcznie edytować kodu w języku XML. Środowisko Visual Studio udostępnia jako pomoc okno projektowe dla testów uporządkowanych. Przykładową zawartość takiego okna pokazano na rysunku 9.20.



Rysunek 9.20. Do testu uporządkowanego można dodać istniejący test jednostek, aby utworzyć nowy test, w którym poszczególne testy uruchamiane są w określonej kolejności

Po lewej stronie okna dialogowego znajdują się wszystkie testy z rozwiązania. Można wyświetlić testy z wybranej listy (zobacz podrozdział „Organizowanie testów"). Po lewej stronie należy zaznaczyć poszczególne testy i użyć strzałki (>) w celu ich dodania do testu uporządkowanego. Skierowane w górę i w dół strzałki widoczne po prawej stronie służą do zmiany kolejności przeprowadzania testów.

Przy wykonywaniu testów uporządkowanych środowisko Visual Studio uruchamia każdy test jednostek w podanej kolejności. Jeśli którykolwiek test się nie powiedzie, cały test uporządkowany zakończy się niepowodzeniem. Oczywiście, można wyświetlić szczegółowe informacje na temat testu uporządkowanego, aby zobaczyć, które testy zakończyły się sukcesem, a które — porażką.

Organizowanie testów

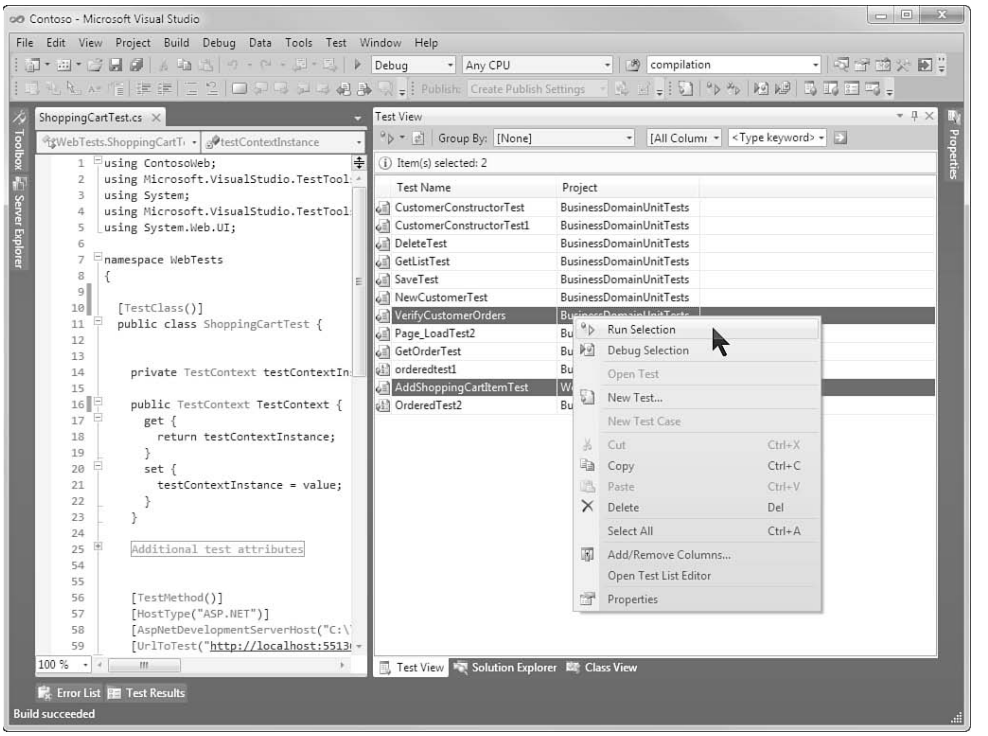
Większość projektów testów jednostek składa się z licznych testów — nieraz są ich setki. Ponadto często w jednym rozwiązaniu znajduje się wiele projektów testów jednostek. Zarządzanie wszystkimi testami i uruchamianie tylko tych potrzebnych w danym momencie bywa trudne. Visual Studio udostępnia pewną pomoc w tym zakresie — okna *Test View* i *Test List Editor*.

Okno Test View

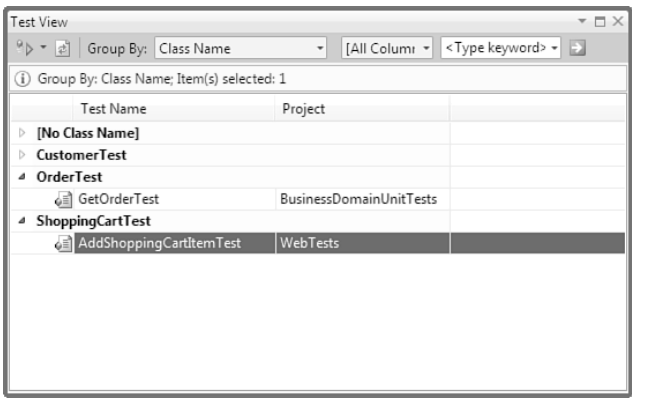
Okno *Test View* umożliwia wyświetlenie wszystkich testów w otwartym rozwiązaniu, zaznaczenie tych, którymi programista jest zainteresowany, i albo uruchomienie ich, albo zarządzanie właściwościami ich atrybutów.

Dostęp do okna *Test View* można uzyskać z menu *Test* (*Test/Windows/Test View*). Na rysunku 9.21 pokazano przykładową zawartość okna. Warto zauważyć, że można zaznaczyć jeden lub kilka testów, kliknąć prawym przyciskiem myszy i uruchomić wybrane testy w jednym przebiegu. Dzięki temu nie trzeba uruchamiać wszystkich testów jednostek, jeśli ważna jest tylko jedna metoda testowa (lub ich mała grupa).

W górnej części okna *Test View* znajduje się pasek narzędzi. Umożliwia on wyszukiwanie testów na podstawie słów kluczowych i modyfikowanie sposobów grupowania lub filtrowania testów. Sprawia to, że znalezienie testów jest łatwiejsze. Można na przykład zmienić wartość opcji *Group By* na *Class Name*, aby wyświetlić wszystkie testy dla określonej klasy. Przykładowe ustawienia pokazano na rysunku 9.22.



Rysunek 9.21. Okno Test View umożliwia łatwe zaznaczenie zbioru testów do uruchomienia

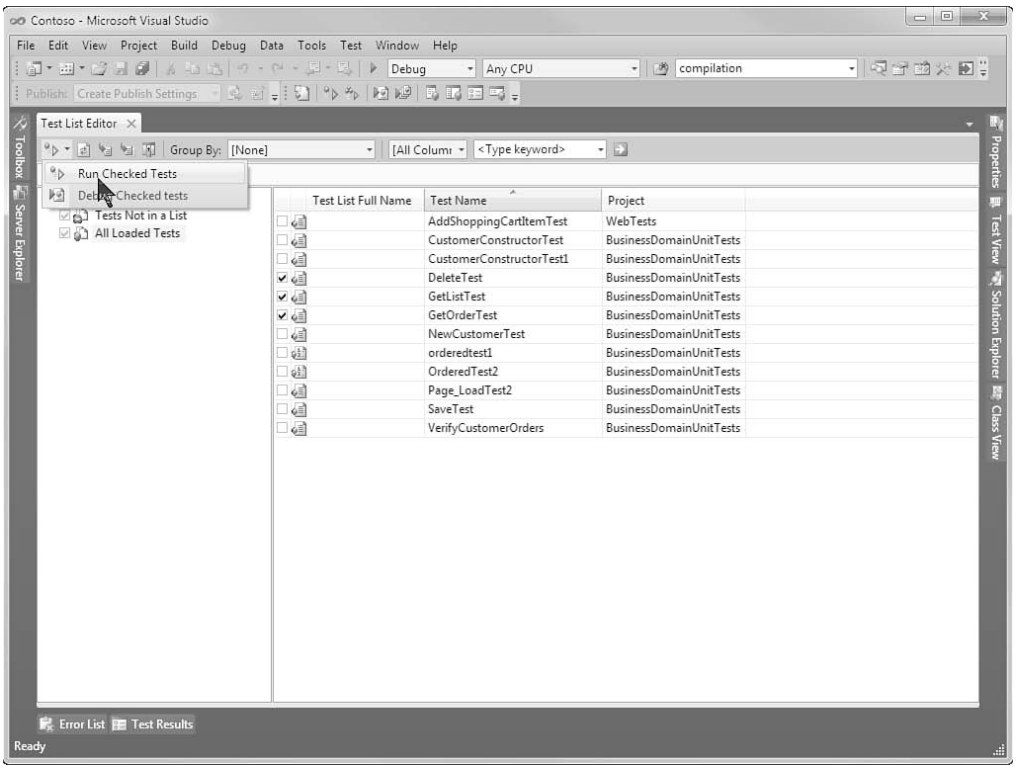


Rysunek 9.22. Można użyć opcji Group By, aby ułatwić sobie znalezienie testów jednostek

Okno Test List Editor

Visual Studio udostępnia też okno *Test List Editor*. Posiada ono podobne funkcje jak okno *Test View*. Umożliwia wyszukiwanie testów, grupowanie ich i uruchamianie zbiorów testów. Jednak w odróżnieniu od okna *Test View* pozwala także zapisywać zdefiniowane listy testów. W ten sposób można utworzyć i ponownie wykorzystać listy, kiedy zajdzie potrzeba uruchomienia zbioru testów.

Dostęp do okna *Test List Editor* można uzyskać z menu *Test* (*Test/Windows/Test List Editor*). Pojawi się okno podobne do tego z rysunku 9.23. Po lewej stronie widnieją listy testów. Domyślnie dostępne są opcje *Lists of Tests*, *Tests Not in a List* i *All Loaded Tests*. Jeśli programista wybierze listę z testami (na przykład *All Loaded Tests*), może albo uruchomić całą listę, albo użyć pól wyboru widocznych po prawej stronie okna do wybrania z listy testów do przeprowadzenia.

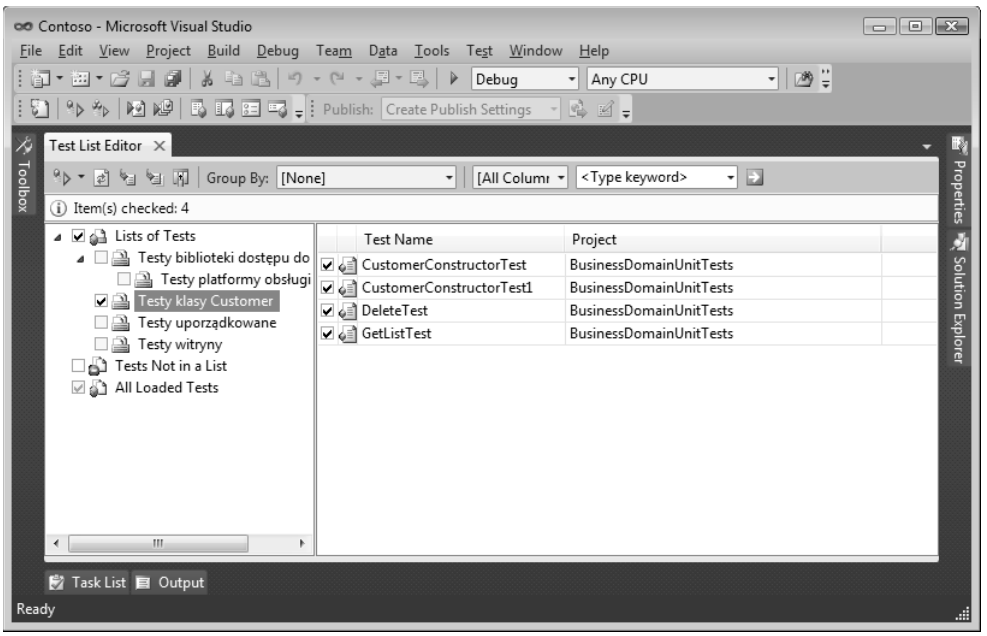


Rysunek 9.23. Można zaznaczyć testy w oknie Test List Editor i uruchomić je jako grupę

Okna *Test List Editor* można używać do tworzenia nowych list testów w celu zapisania testów regularnie wykonywanych w jednym zestawie. Aby utworzyć listę, należy kliknąć prawym przyciskiem myszy pozycję *Lists of Tests* w oknie *Test List Editor*, a następnie wybrać opcję *New Test List*. Pojawi się okno dialogowe *Create New Test List*. W oknie można podać nazwę listy testów i określić jej pozycję wśród innych list (zobacz rysunek 9.24).

Aby dodać test jednostek do nowej listy, można zastosować technikę przeciągania. Najpierw należy wybrać listę *All Loaded Tests* w celu wyświetlenia wszystkich wczytanych testów. Następnie trzeba zaznaczyć wszystkie testy, które mają znaleźć się na nowej liście. Wybrane testy wystarczy przeciągnąć na nazwę niestandardowej listy widoczną po lewej stronie okna *Test List Editor*. W ten sposób testy zostaną powiązane z nową listą. Na rysunku 9.25 pokazano kilka przykładowych niestandardowych list i testy z jednej z nich.

Rysunek 9.24. Można definiować nowe listy testów do wielokrotnego użytku



Rysunek 9.25. Można definiować niestandardowe listy testów i dodawać do nich testy jednostek

Po utworzeniu listy testów do przeprowadzenia można uruchomić je za pomocą paska narzędzia okna *Test List Editor*. W tym celu należy wybrać opcję *Run Checked Tests* (zobacz rysunek 9.23).

Podsumowanie

W tym rozdziale pokazaliśmy, jak używać platformy testów jednostek do definiowania projektów testów i jak za pomocą atrybutu `TestClass` tworzyć pliki klas testów. Każda metoda testowa powinna mieć atrybut `TestMethod`. Przy użyciu klasy atrybutu `DataSource` można tworzyć testy jednostek powiązane z danymi.

Visual Studio umożliwia generowanie testów jednostek na podstawie istniejącego kodu. Udostępnia też w narzędziu *Test View* okno dialogowe *Properties*, aby ułatwić generowanie kodu atrybutów testów.

Można tworzyć testy jednostek obsługiwane przez ASP.NET (na serwerze IIS lub serwerze rozwojowym środowiska Visual Studio). W testach jednostek ASP.NET wykorzystywany jest obiekt `TestContext` do uzyskania dostępu do informacji o środowisku ASP.NET, w tym o przetwarzanej stronie, sesji i zmiennych serwera.

Pisanie testów jednostek może prowadzić do zmniejszenia liczby problemów w kodzie produkcyjnym. Przygotowanie zestawu testów kodu ułatwia jego zrozumienie i zwiększa niezawodność. Ponadto programista może z większą pewnością wprowadzać zmiany, ponieważ potrafi określić, w jakim fragmencie kodu w wyniku modyfikacji pojawiły się usterki.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Y

Z

Ź

Ż

Microsoft Visual Studio 2010

Visual Studio 2010 to najnowsza wersja środowiska programistycznego firmy Microsoft. Każdy programista, który zdecyduje się wykorzystać bogaty zestaw zgromadzonych tu narzędzi, osiągnie maksymalną produktywność w pracy i będzie mógł tworzyć kod działający w systemie Windows oraz w sieci. Dodatkowo będzie mieć do dyspozycji technologię Silverlight i możliwość budowania aplikacji w chmurze z użyciem platformy Microsoftu — Azure. Na tym jednak nie koniec. W Visual Studio 2010 pojawiły się kolejne innowacje. Znajdziesz tu nowy edytor kodu, oparty na platformie WPF, rozszerzenia środowiska IDE związane z platformą MEF oraz możliwość pisania skryptów za pomocą technologii Silverlight. Ponadto języki platformy .NET wzbogacono o obsługę programowania dynamicznego. Pojawił się nowy język F#, służący do programowania funkcyjnego, oraz mechanizmy usprawniające szybkie pisanie kodu wyższej jakości.

„Microsoft Visual Studio 2010. Księga eksperta" zawiera kompletne omówienie środowiska Visual Studio 2010, a skoncentrowanie się na wersji Professional pozwoliło autorom na stworzenie opisu bardziej szczegółowego niż kiedykolwiek wcześniej. Dzięki temu podręcznikowi nauczysz się w pełni wykorzystywać możliwości platformy .NET, w tym technologii WPF (pozwalającej na tworzenie bogatych klientów), technologii WCF (stworzonej do budowania dynamicznych rozwiązań opartych na usługach) czy też technologii WF (umożliwiającej rozwijanie ustrukturyzowanych programów na podstawie procesów biznesowych). Znajdziesz tu także omówienie nowych narzędzi Microsoftu, przeznaczonych do testowania, instrumentacji aplikacji i analizowania kodu.

- Języki platformy .NET
- Przeglądarki i eksploratory
- Tworzenie projektów WPF
- Szablony XML
- Testowanie, refaktoryzacja i diagnozowanie kodu
- Platforma MEF
- Tworzenie aplikacji ASP.NET
- Bogate i inteligentne interfejsy użytkownika
- Złożone aplikacje internetowe
- Silverlight 4.0
- Aplikacje biznesowe oparte na pakiecie Office
- Technologia Windows Azure i aplikacje działające w chmurze

Wykorzystaj zdobytą wiedzę i zostań mistrzem programowania!

Mike Snell zawodowo pomaga zespołom budować oprogramowanie dla przedsiębiorstw i tworzyć aplikacje komercyjne. Kieruje działem Solutions w CEI (www.ceiamerica.com). Wraz z zespołem, którym zarządza, oferuje klientom usługi z zakresu konsultacji, mentoringu i rozwijania architektury. Mike jest też dyrektorem regionalnym Microsoftu (www.TheRegion.com).

Lars Powers jest głównym doradcą technicznym ds spraw ISV w zespole Developer and Platform Evangelism Microsoftu. Współpracuje z najważniejszymi partnerami tej korporacji z zakresu ISV i pomaga im tworzyć rozwiązania przy użyciu technologii nowej generacji. Przed związaniem się z Microsoftem Lars był niezależnym konsultantem do spraw szkoleń i poradnictwa na temat platformy .NET.

ISBN 978-83-246-3029-5