STUDIA INFORMATICA 2009
Volume 30 Number 2B (84)

Dariusz Rafał AUGUSTYN, Ireneusz STANEK
Politechnika Śląska, Instytut Informatyki

ZASTOSOWANIE MODUŁU NUNIT W TESTOWANIU JEDNOSTKOWYM APLIKACJI BAZODANOWYCH WYTWORZONYCH W TECHNOLOGII .NET

Streszczenie. Artykuł dotyczy metod testowania jednostkowego funkcjonalności bazodanowych z wykorzystaniem silnika testującego NUnit. Testy jednostkowe funkcjonalności bazodanowych, z powodu trwałych modyfikacji bazy, nie są bezstanowe i niezależne, ponadto wymagają przygotowania właściwej zawartości bazy danych przed każdym wywołaniem. Aby umożliwić realizację takich testów, zgodnie z paradygmatem testów jednostkowych, można zastosować podejście polegające na automatycznym przywracaniu bazy do określonego stanu na podstawie danych XML. Innym podejściem jest odtwarzanie stanu danych przez otoczenie testu transakcją COM+. Artykuł przedstawia wykorzystanie mechanizmu rozszerzania NUnit w celu definicji własnych atrybutów, ukrywających szczegóły implementacji odtwarzania danych.

Słowa kluczowe: automatyzacja procesu testowania, testy jednostkowe, testowanie warstwy danych, odtwarzanie stanu bazy danych, rozproszone transakcje, usługi COM+, rozszerzenia NUnit

NUNIT-BASED METHODS SUPPORTING UNIT TESTING OF .NET DATABASE APPLICATIONS

Summary. The paper presents unit testing methods of database functionality using NUnit module. Unit tests should be stateless and independent. Database should have expected content before running of unit test. To fulfill these conditions a mechanism of placing database into know state and database content rollbacking should be used. It could be obtained by an approach based on XML data for setting expected database content. Another described approach uses transaction processing supported by COM+ services for automating rollback of changes made by test. The NUnit extension mechanism used for rollback functionality encapsulation is shown. It simplifying implementation of database functionally unit tests by using specific user-defined attributes.

Keywords: automatic testing process, unit tests, data access layer testing, database state rollback, distributed transaction, COM+ Enterprise Services, NUnit extensions

1. Testy jednostkowe z wykorzystaniem NUnit, NMock

Testy jednostkowe (ang. unit tests) przeznaczone są do weryfikacji poprawności funkcjonowania prostych elementów oprogramowania – jednostek oprogramowania – np. pojedynczych metod. Testy takie są definiowane przez programistę. Tworzenie, rozwijanie i uruchamianie testów jednostkowych, dzięki opracowaniu metod i wprowadzeniu narzędzi wspomagających, stało się integralną częścią procesu wytwórczego.

Do testowania jednostkowego elementów aplikacji wytworzonych w technologii .NET można wykorzystać moduł programowy (silnik testujący) NUnit, koncepcyjnie bazujący na idei produktów klasy xUnit dla innych technologii. Zastosowanie NUnit jest obszernie opisywane w literaturze i na stronach WWW (np. [1, 4]). Przy specyfikacji testów dla NUnit wykorzystuje się mechanizm atrybutów, udostępniany w ramach infrastruktury .NET Framework [3]. Zbudowanie zestawu testującego sprowadza się do wytworzenia testującej biblioteki DLL, implementującej tzw. klasę testową. Klasa testowa, oznaczona atrybutem *TestFixture*, „agreguje" następujące metody:

- opcjonalna metoda, uruchamiana tylko raz, przed wszystkimi pojedynczymi testami, oznaczona atrybutem TestFixtureSetup,
- opcjonalna metoda, uruchamiana tylko raz, po wszystkich pojedynczych testach, oznaczona atrybutem TestFixtureTearDown,
- opcjonalna metoda, uruchamiania przed każdym pojedynczym testem, oznaczona atrybutem Setup,
- opcjonalna metoda, uruchamiania po każdym pojedynczym teście, oznaczona atrybutem TearDown,
- pojedynczy test – metoda testująca, implementująca test jednostkowy, oznaczona atrybutem Test (na ogół w klasie testowej występuje wiele pojedynczych metod testujących).

Funkcjonalność jest testowana jednostkowo w ramach metod oznaczonych atrybutem *Test* (inne, wymienione metody mają charakter pomocniczy).

Biblioteka testująca wczytywana jest przez silnik testujący NUnit.exe, który, wykorzystując tzw. mechanizm refleksji .NET [3], odnajduje metody oznaczone odpowiednimi atrybutami i uruchamia je. Spośród metod oznaczonych atrybutem *Test* mogą być uruchamiane wszystkie bądź tylko te wskazane bezpośrednio przez użytkownika konsoli NUnit.

Wielowarstwowe aplikacje mogą być efektywnie testowane z wykorzystaniem mechanizmu tzw. obiektów atrap (ang. mock object) [2, 4]. Pewne obiekty, z których korzystają złożone metody przetwarzające (podlegające testowaniu jednostkowym), mogą być „trudne" w obsłudze (niekompletna implementacja klasy obiektu, duży czas utworzenia obiektu, nieefektywne działanie, niedostępność zdalnego obiektu w trakcie działania testu, nadmierne wykorzystanie zasobów pamięciowych czy dyskowych itp.). Wówczas taki obiekt można dla celów testowych zastąpić zamiennikiem – atrapą. Taki obiekt zastępczy ma taki sam interfejs programowy jak obiekt oryginalny. Wykorzystanie obiektów zastępczych sprowadza się do zarejestrowania oczekiwań w stosunku do obiektów atrap (sekwencja użycia własności i wywołań metod obiektu) i uruchomienia testów jednostkowych złożonych metod przetwarzających, działających na obiektach zastępczych (nie na obiektach oryginalnych). Mechanizm ten, wspierający testowanie aplikacji wielowarstwowych z użyciem NUnit, realizowany jest w ramach modułu NMock [2].

2. Testowanie jednostkowe aplikacji bazodanowych z wykorzystaniem NDbUnit

Testy jednostkowe powinny być tworzone jako niezależne, bezstanowe metody (niegenerujące tzw. efektów ubocznych, wpływających na działanie innych metod testowych). Tej cechy nie mają testy jednostkowe funkcjonalności bazodanowych, które, zmieniając zwartość bazy danych, są stanowe z natury. Najistotniejszym problemem związanym z testowaniem aplikacji bazodanowej jest zapewnienie działania takiego testu na określonej zawartości bazy danych i przywracania po teście odpowiedniej zawartości. Zatem proces automatyzacji takich testów wymaga również automatyzacji przygotowania stanu danych przed wykonaniem testu i odtworzenia stanu bazy danych po realizacji testu.

Zadanie to może być realizowane z wykorzystaniem modułu programowego NDbUnit [7, 8, 9], odtwarzającego pożądane stany bazy danych na podstawie zdefiniowanych przez użytkownika zbiorów danych XML. W podejściu bazującym na NDbUnit przygotowania wymagają następujące elementy:

a) opis struktury bazy danych w postaci pliku tzw. schematu XSD [6],
b) opis zawartości bazy danych w postaci pliku XML (zgodnych ze schematem z pkt. a).

Wykorzystanie NDbUnit zostanie zilustrowane na trywialnym przykładzie testowania funkcjonalności, dotyczącej pojedynczej, dwukolumnowej tablicy. Struktura *Person* tablicy relacyjnej bazy danych SZBD MS SQLServer zdefiniowana za pomocą komendy DDL:

```
CREATE TABLE [Person](
[PersonID] [int] NOT NULL,
```

```
[Name] [nvarchar](50) NOT NULL,
CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED
```

może zostać wyrażona w postaci schematu XSD (plik TestDataSet.xsd):

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="TestDataSet" targetNamespace="http://tempuri.org/TestDataSet.xsd"
xmlns:mstns="http://tempuri.org/TestDataSet.xsd"
xmlns="http://tempuri.org/TestDataSet.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-
com:xml-msdata" attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="TestDataSet" msdata:IsDataSet="true"
msdata:UseCurrentLocale="true">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Person">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="PersonID" type="xs:int" />
              <xs:element name="Name">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:maxLength value="50" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1" msdata:PrimaryKey="true">
      <xs:selector xpath=".//mstns:Person" />
      <xs:field xpath="mstns:PersonID" />
    </xs:unique>
  </xs:element>
</xs:schema>
```

Przykładowa zawartość tablicy *Person* jest utrwalona w pliku XML - TestDataSet.xml:

```
<?xml version="1.0" standalone="yes"?>
<TestDataSet xmlns="http://tempuri.org/TestDataSet.xsd">
  <Person>
    <PersonID>1</PersonID><Name>CharleMagne</Name>
  </Person>
  <Person>
    <PersonID>3</PersonID><Name>Cezar</Name>
  </Person>
  <Person>
    <PersonID>4</PersonID><Name>Hannibal</Name>
  </Person>
</TestDataSet>
```

Inna, przykładowa zawartość tablicy *Person* utrwalona w pliku XML – UpdateTestData-Set.xml (wykorzystana również w testach omawianych poniżej):

```
<Person>
    <PersonID>4</PersonID><Name>Hannibal</Name>
</Person>
```

Zawartości plików XSD, XML mogą zostać wygenerowane automatycznie przez utworzenie tzw. typowanego obiektu data set[1][5], zasilenie go danymi z bazy metodą *Fill* oraz utrwalenie tego obiektu (zapis do plików) przez wywołania metod *WriteXmlSchema* i *WriteXml*.

Przykładowa klasa *Person*, zdefiniowana w języku C#, stanowi namiastkę warstwy danych (DAL – ang. Data Access Layer), podlegającej testom jednostkowym i wymagającej odpowiedniego przygotowania/odtwarzania stanu testowej bazy danych. Kod źródłowy prostej metody *Update*, zapisującej w bazie danych (wykorzystującej klasyczny interfejs dostępu do baz danych - ADO.NET [5]), został pokazany poniżej:

```
01  class Person : IPerson
02  { public int PersonID;
03    public String Name;
04    // aktualizacja wiersza w tablicy Person,
05    // na podstawie składowych PersonID, Name
06    public bool Update()
07    {  SqlConnection connection = new SqlConnection(/*par. połączenia*/);
08               connection.Open();
09               SqlTransaction transaction = connection.BeginTransaction();
10               SqlCommand command = new SqlCommand("", connection,
transaction);
11               command.CommandText =
12       "UPDATE [Person] SET Name = @pName WHERE PersonID =@pPersonID";
13               command.Parameters.Add("@pPersonID", SqlDbType.Int);
14       command.Parameters.Add("@pName", SqlDbType.Text);
15       command.Parameters["@pPersonID"].Value = PersonID;
16       command.Parameters["@pName"].Value = Name;
17       int numRowsUpdated=-1;
18               try  {   numRowsUpdated += command.ExecuteNonQuery();
19                        transaction.Commit();
20               }
21               catch (Exception e)
22               {    transaction.Rollback();
23               }
24               finally
25               {   connection.Close();
26               }
27      return (numRowsUpdated>0);
28    }
29  }
```

W metodzie *Update* następuje jawne zatwierdzenie transakcji (linia 19).

Poniżej przedstawiono kod źródłowy klasy testowej *DBTestClass*, realizujący w metodzie *DBOperationTest1* sprawdzanie poprawności funkcjonowania zapisu do bazy danych, przez wywołanie metody *Person.Update*.

```
01 [TestFixture]
02 public class DBTestClass
03 {// Obiekt sterujący zmianą zawartości bazy danych
04  private SqlDbUnitTest hTest;
05  // Parametry połączenia (tzw. connection string)
06  String ConnectStr = @"Data Source=.;Initial Catalog=.;
07                        Integrated Security=True";
```

[1] Wykorzystując mechanizm „przeciągnij i upuść” w ramach środowiska Visual Studio 2008, wystarczy przenieść wskazaną tablicę bazy danych z okna *Server Builder* na okno *DataSet Designer*.

```
08  [TestFixtureSetUp]
09  protected void OneTimeSetUp()
10  { // Przekazanie parametrów połączenia
11    hTest = new SqlDbUnitTest(ConnectStr);
12    // Inicjalizacja obiektu opisem struktury danych
13    hTest.ReadXmlSchema(@"c:\TestDataSet.xsd");
14    // Przygotowanie (deklaracja)zestawu danych do zmiany zawartości b.d.
15    hTest.ReadXml(@"c:\TestDataSet.xml");
16    // Wykonanie operacji wstawienia lub aktualizacji wierszy
17    // (na podstawie wartości klucza),
18    // niezbędnych do realizacji grupy testów
19    hTest.PerformDbOperation(DbOperationFlag.Refresh);
20    // Przygotowanie innego, ograniczonego zestawu danych,
21    // na potrzeby odtwarzania przed każdym pojedynczym testem
22    hTest.ReadXml(@"c:\UpdateTestDataSet.xml");
23  }

24  [SetUp]
25  protected void AnyTimeSetup()
26  { // Każdorazowa (przed każdym testem) aktualizacja testowej bazy
27    // danych, w oparciu o ograniczony zestaw danych,
28    // zdefiniowany w UpdateTestDataSet.xml
29    hTest.PerformDbOperation(DbOperationFlag.Update);
30  }

31  [Test]
32  public void DatabaseOperationTest1()
33  { // Test podmiany wartości pola Name w rekordzie o wartości klucza 4
34    Person person = new Person();
35    person.PersonID = 4; person.Name = "Scypion";
36    // Zapis do bazy danych; zatwierdzona transakcja
37    person.Update();
38    String expected = person.Name, result;
39    // Pobranie do result z bazy danych wartości pola Name
40    // …
41    // Sprawdzenie wartości odczytanej z bazy danych
42    Assert.AreEqual(expected, result,
43        "Nieskuteczny zapis do bazy danych");
44  }

45  [Test]
46  public void DatabaseOperationTest2(){ /* inny test */ }

47  [TestFixtureTearDown]
48  protected void OneTimeDown()
49  { // Ustawienie pełnego zestawu danych
50    hTest.ReadXml(@"c:\TestDataSet.xml");
51    // Usunięcie wszystkich danych testowych
52    hTest.PerformDbOperation(DbOperationFlag.Delete);
53  }
```

Metoda *ReadXMLSchema* (linia 13) pozwala na zainicjowanie struktury danych obiektu *hTest*, obsługującego mechanizm odtwarzania NDbUnit. Definicja zawarta w tym schemacie pozwala m.in. automatycznie generować (w sposób ukryty dla użytkownika) odpowiednie komendy SQL odpowiedzialne za wstawianie, kasowanie czy aktualizację bazy danych. Metoda *ReadXML* (linie 15, 22) pozwala na określenie konkretnych danych, służących później do wstawiania, kasowania, aktualizacji. Metoda *PerformDbOperation* (linie 19, 29, 52) realizuje określone operacje SQL związane ze zmianą zawartości bazy danych. Rodzaj operacji wynika z podanej wartości stałej (*Insert*, *InsertIdentity*, *Delete*, *DeleteAll*, *Update*, *Refresh*, *CleanInsert*, *CleanInsertIdentity* [7, 8]) typu wyliczeniowego (*DbOperationFlag*), przekazanej w parametrze metody.

Działanie przykładowej klasy testowej *DBTestClass* zostanie zilustrowanie przez śledzenie stanu zawartości tablicy *Person* w trakcie realizacji zestawu testów. Stany tablicy *Person* opisane są w tabeli 1 i rozróżniane przez wartości kolumny Wersja tej tabeli. Kolumny *PersonID*, *Name* odpowiadają wprost kolumnom tablicy *Person*. Początkowo tablica *Person* zawiera tylko jeden wiersz (Wersja A w tabeli 1).

Tabela 1
Stany zawartości tablicy *Person* (*PessonId*, *Name*) w chwilach oznaczonych określonymi wartościami kolumny *Wersja*

Wersja	**PersonId**	**Name**
A	2	Assar
B	1	CharleMagne
B	2	Assar
B	3	Attyla
B	4	*Hannibal*
C	1	CharleMagne
C	2	Assar
C	3	Attyla
C	4	*Scypion*

W celu uzyskania inicjalnej, określonej zawartości bazy danych (zawartość TestDataSet.xml) dla całego zestawu testów realizowana jest operacja odświeżania, tzn. wprowadza się dodatkowe, nowe wiersze lub aktualizuje się istniejące, jeśli już w bazie istnieją wiersze o zadanym kluczu, jednocześnie zachowując stan innych istniejących wierszy. W programie pokazanym powyżej, w metodzie *OneTimeSetup* następuje jednorazowe przygotowanie „pełnego" zestawu danych testowych dla grupy testów (metody *DatabaseOperationTest1*, *DatabaseOperationTest2*). Po wykonaniu *PerformDbOperation* (*DbOperationFlag.Refresh*) (linia 19) liczba wierszy tablicy *Person* wynosi 4, a stan tablicy odpowiada Wersji B tabeli 1.

W opisywanym scenariuszu testowym zakłada się, że zawartość TestDataSet.xml wprowadzona do bazy danych przed realizacją zestawu testów nie jest całkowicie wystarczająca dla zapewnienia poprawności/niezależności wykonań pojedynczych testów, które, zmieniając zwartość bazy, mogłyby wpływać na siebie nawzajem. W opisywanym przykładzie zakłada się, że każdy pojedynczy test jeszcze wymaga z osobna dodatkowego przygotowania zawartości bazy w pewnym ograniczonym zakresie – baza danych powinna przed uruchomieniem testu zawierać dane pochodzące z UpdateTestDataSet.xml. W związku z tym obiekt *hTest* sterujący odtwarzaniem zainicjowany został wartościami z UpdateTestDataSet.xml (linia 22). W metodzie *AnyTimeSetup*, wykonywanej przed każdym pojedynczym testem, następuje odpowiednia aktualizacja bazy danych przez wywołanie *PerformDbOperation*(*DbOperationFlag.Update*) w linii 29. Stąd też, nawet jeśli w metodzie testowej *DatabaseOperationTest1* po wykonaniu modyfikacji pola *Name* przez metodę *Update*, stan bazy danych jest opisy-

wany jako Wersja C tabeli 1, to przed wykonaniem następnej metody testowej *DatabaseOperationTest1* w linii 46 stan danych znowu jest przywrócony i odpowiada Wersji B tabeli 1.

Po zakończeniu całego zestawu testów dane testowe są usuwane z bazy. W przykładzie, w metodzie *OneTimeDown,* następuje jednorazowe usunięcie tylko tych danych, które zostały początkowo wygenerowane na potrzeby testów (kasowanie wierszy wg wartości klucza na podstawie zawartości TestDataSet.xml). Po wykonaniu *PerformDbOperation*(*DbOperationFlag.Delete*) (linia 53) liczba wierszy tablicy *Person* wynosi 1, a stan tablicy odpowiada Wersji A tabeli 1.

Można zaproponować inny wariant sposobu ustalania/odtwarzania danych, w którym dane testowe są w całości aktualizowane/kasowane przed/po każdym teście z osobna. Wystarczy usunąć linię 22 i zdefiniować metody wywoływane przed każdym/po teście w następujący sposób:

```
01  [SetUp]
02  protected void AnyTimeSetup()
03  { // aktualizacja przed każdym testem bazy danych
04    // w oparciu o pełny zestaw danych TestDataSet.xml
05    hTest.PerformDbOperation(DbOperationFlag.Refresh);
06  }

07  [TearDown]
09  protected void AnyTimeDown()
10  {  hTest.PerformDbOperation(DbOperationFlag.Delete);
11  }
```

3. Mechanizmy rozszerzania funkcjonalności narzędzia testującego – Nunit Extensions

Dzięki mechanizmowi rozszerzania moduły uruchamiające testy (silniki testujące) można rozbudowywać o własne specyficzne funkcjonalności. Mechanizm NUnit Extensions m.in. pozwala na włączanie tzw. dodatków (ang. NUnit Addins) – modułów programowych utworzonych przez użytkownika lub pochodzących od niezależnych producentów, pozwalających na przechwytywanie zdarzeń lub zmianę zachowania silnika NUnit [10].

Rozszerzanie NUnit może odbywać się przez podłączanie (ang. hook) własnych funkcjonalności do miejsc rozszerzenia (ang. Extension points), operujących na różnych poziomach definicji zestawu testów [10, 11]:

- Suite Builders – pozwala rozszerzać/zmieniać funkcjonowanie NUnit na „najwyższym poziomie abstrakcji" opisu zestawu testowego, przez własną redefinicję zachowania wszystkich elementów klasy testującej, w szczególności pozwala na zdefiniowanie własnych atrybutów samej klasy testującej (np. zamiast TestFixture), na które NUnit będzie uwrażliwiony. Przykład: zdefiniowanie atrybutu klasy (zamiast TextFixture), który spo-

woduje, że wszystkie metody klasy są metodami testującymi (bez konieczności wskazywania ich atrybutem Test [11]).

- Test Case Builders – pozwala na definicję/redefinicję pojedynczego testu w ramach istniejącego zestawu testów, w szczególności pozwala dynamicznie tworzyć nowe testy na etapie wykonania zestawu (np. tworzenie i uruchamianie ciągu testów na podstawie zawartości pliku czy bazy danych). Przykłady: rozszerzenia RowTest [12], SqlServerDataSource [11].
- Test Decorators – pozwala na wzbogacenie funkcjonalności istniejących elementów zestawu testowego. Na przykład metoda testowa oznaczona standardowym atrybutem Test może być „dekorowana" dodatkową funkcjonalnością, np. w wyniku dodania dodatkowego, własnego atrybutu. Przykład zastosowania w celu odtwarzania stanu bazy danych pokazany będzie w następnym rozdziale.
- Event Listeners – pozwala na uwrażliwienie NUnit na wystąpienie określonego zdarzenia: RunStarted, RunFinished, SuiteStarted, SuiteFinished, TestStarted, TestOutput, TestFinished, UnhandledException. Rozszerzenie takiego typu może być wykorzystane w celu rozbudowy funkcjonalności raportowania/monitorowania procesu testowania.

Rozszerzenia (AddIns) implementowane są w postaci biblioteki DLL i instalowane są wraz ze standardowymi modułami silnika testowego[2].

4. Testowanie jednostkowe aplikacji bazodanowych z wykorzystaniem funkcjonalności COM+ Enterprise Services

COM+ Enterprise Services jest rozwinięciem technologii Component Object Model i Microsoft Transaction Server (MTS). W szczególności, technologia ta pozwala na realizację rozproszonych transakcji (ang. distributed transaction), obejmujących modyfikacje zróżnicowanych zasobów (np. kolejki komunikatów, bazy danych), zlokalizowanych w różnych węzłach sieci komputerowej. Model zatwierdzania/wycofywania transakcji rozproszonej może być z powodzeniem zastosowany do odtwarzania stanu bazy danych w ramach procesu testowania, dotyczącego pojedynczej, lokalnej bazy danych.

W bardzo dużym uproszczeniu można stwierdzić, że zatwierdzanie transakcji rozproszonej (w ramach tzw. protokołu dwufazowego) zakłada, że w tzw. pierwszej fazie, „wstępnie" zatwierdzane są transakcje lokalne w węzłach systemu rozproszonego. Następnie, jeśli się wszystkie lokalne zatwierdzenia powiodą (albo nie powiodą), poszczególne węzły zgłaszają

[2] W przypadku narzędzia NUnit wystarczy umieszczenie bibliotek rozszerzających w podkatalogu *\bin\addin*, znajdującym się w katalogu głównym NUnit.

gotowość do zatwierdzenia (albo wycofania) transakcji do koordynatora transakcji rozproszonej. W ramach drugiej fazy, po odebraniu zgłoszeń od wszystkich uczestników, koordynator rozgłasza komunikat żądania zatwierdzenia (bądź wycofania, jeśli chociaż w jednym węźle nie powiodło się lokalne zatwierdzenie) do węzłów uczestniczących. Węzły uczestniczące, po odebraniu komunikatu od koordynatora, dokonują ostatecznego zatwierdzenia lub wycofania transakcji przeprowadzanej na lokalnym zasobie.

Przedstawiony wcześniej model pozwala na wyjaśnienie wykorzystania ww. mechanizmu do automatyzacji wycofywania zmian dokonanych w ramach testu. W takim rozwiązaniu zakłada się, że potencjalne zmiany będą się odbywać w ramach transakcji przetwarzanej dwufazowo, nawet jeśli transakcją będzie objęty tylko jeden zasób lokalny (jedna testowa baza danych). Nawet jeśli w ramach realizacji operacji w teście nastąpi poprawne zatwierdzenie transakcji przez standardową komendę *Commit*, dostępną z poziomu ADO.NET, to, jeśli operacje te objęte zostały dwufazowym przetwarzaniem transakcyjnym, komenda *SetAbort*, dostępna z poziomu COM+ Enterprise Services wycofa „wstępnie" zatwierdzoną transakcję. Rozwiązanie takie zostało zaproponowane przez R. Osherove [13, 14] i stanowi oficjalny moduł rozszerzający XtUnit, dostępny na stronnie domowej NUnit [15].

W rozwiązaniu pokazanym poniżej (funkcjonalnie zgodnym z [15]) wykorzystano mechanizm wycofywania dwufazowej transakcji w ramach wywołania każdej pojedynczej metody testowej. Mechanizm rozszerzania NUnit, działający na poziomie tzw. dekorowania metod testowych, pozwolił na ukrycie szczegółów realizacji funkcjonalności, związanej z wycofywaniem transakcji rozproszonej. Wykorzystanie własnego atrybutu *RollBack* pozwala na wskazanie metod testowych, objętych wycofywaniem transakcji dwufazowej.

Poniżej pokazany kod źródłowy C# stanowi implementację rozszerzenia NUnit (dodatku do modułu NUnit), pozwalającego na wykorzystanie niestandardowego atrybutu *RollBack*.

```
01[NUnitAddin(Description = "Rollback AddIn for database tests")]
02 public class RollbackAddin : IAddin, ITestDecorator
03 { // Rejestracja rozszerzenia na „poziomie" TestDecorators
04   public bool Install(IExtensionHost host)
05   {IExtensionPoint decorators=host.GetExtensionPoint("TestDecorators");
06       if (decorators == null)      return false;
07           decorators.Install(this); return true;
08   }
09
10   // Implementacja metod interfesju ITestDecorator,
11   // podstawienie własnej, opakowanej wersji metody testującej
12   public Test Decorate(Test testToExecute,
13                        System.Reflection.MemberInfo member)
14   { // Sprawdzenie czy element jest odpowiednikiem
15     // metody testującej - NUnitTestMethod, (a nie np. TestFixture)
16     if (testToExecute is NUnitTestMethod)
17      { // Pobieranie wszystkich atrybutów RollBack
18        // dla danej metody testującej
19        Attribute[] attributes = Reflect.GetAttributes(member,
20                   "NUnitAddinAttributes.RollbackTestAttribute", false);
21        // Sprawdzenie czy metoda jest oznaczona
22        // chociaż jednym atrybutem [Rollback]
23        if (attributes.Length > 0)
```

```
24          { // Podstawienie dekorowanej wersji metody testującej
25            // (przysłonięcie domyslnej metody testującej)
26          testToExecute = new RollbackTestMethod
27                    (((NUnitTestMethod)testToExecute).Method);
28          }
29     return testToExecute;
30    }
31  }
32  // Definicja klasy dekorowanej metody testującej
33  public class RollbackTestMethod : NUnitTestMethod
34  { // Przesłonięcie standarowej metody uruchamiającej pojedynczy test
35      public override void Run(TestCaseResult testOriginal)
36      { // Początkowe dekorowanie metody testowej;
37        // dodanie tzw. funkcjonalności before behavior;
38        // rozpoczęcie transakcji COM+ (metoda Enter)
39        ServiceConfig config = new ServiceConfig();
40        config.Transaction = TransactionOption.RequiresNew;
41        ServiceDomain.Enter(config);

42        // Uruchomienie oryginalnej metody testującej
43        base.Run(testOriginal);

44        // Ponowne, końcowe dekorowanie metody testowej;
45        // dodanie tzw. funkcjonalności after behavior;
46        // wycofanie transakcji COM+ (metoda SetAbort)
47        if (ContextUtil.IsInTransaction)
48                          ContextUtil.SetAbort();
49        ServiceDomain.Leave();
50      }
51    }

52    // Definicja własnego atrybutu [RollBack]
53    [AttributeUsage(AttributeTargets.Method, Inherited = false,
54                                  AllowMultiple = true,)]
55    public class RollbackTestAttribute : Attribute {}
```

Fragment programu obejmujący linie 04-08 pozwala zarejestrować klasę *RollbackAddin* jako rozszerzenie NUnit, dekorujące pojedyncze metody testujące. W liniach 10-31 następuje podstawienie obiektu dekorowanego testu (obiektu typu *RollbackTestMethod*), zamiast testu oryginalnego (obiektu typu *NUnitTestMethod*), jeśli metoda testująca posiada atrybut *RollBack*. Linie 32÷51 definiują klasę metod testujących, dekorowanych przez rozszerzenie funkcjonalności metody *Run*. Operacja wycofania potencjalnych zmian w bazie danych, zrealizowanych i ewentualnie zatwierdzonych „wstępnie" przez oryginalną metodę testującą (linia 43), odbywa się przez wycofanie transakcji dwufazowej komendą *SetAbort* w linii 48. Linie 52÷55 definiują „pustą" klasę atrybutu *RollBack* (w linii 54 zadeklarowano możliwość występowania *RollBack* z wraz innymi atrybutami, w szczególności atrybutem *Test*).

Po utworzeniu i instalacji rozszerzającej biblioteki DLL *RollBackAddIn*, której kod źródłowy został przedstawiony powyżej, możliwe jest wykorzystanie atrybutu *RollBack* w dowolnej bibliotece testującej (np. linie 04, 10 poniżej), w celu wycofywania zmian realizowanych w każdej pojedynczej metodzie testującej, np.:

```
01 [TestFixture]
02 public class DBTestClass {

03   [Test]
04   [RollBack]
05    public void DatabaseOperationTest1()
```

```
06    { // …
07      person.Update();
08      // …
09    }

10   [Test, RollBack]
11   public void DatabaseOperationTest2(){ /* inny test */ }

12   [Test]
13   public void NO_DatabaseOperationTest3(){
14            /* metoda nieobjęta odtwarzaniem COM+*/ }
15  }
```

Ewentualne zmiany bazy danych zrealizowane w *DatabaseOperationTest1*, *DatabaseOperationTest2* zostają zawsze wycofane (w szczególności te wykonane w ramach metody *Update*, niezależnie od tego, że w jej ciele wystąpiła komenda *Commit*).

5. Podsumowanie

Testowanie jednostkowe jest istotnym elementem procesu wytwórczego w większości stosowanych metodyk wytwarzania oprogramowania, wspieranym przez wiele narzędzi programowych (np. NUnit i pokrewne [1, 4]). Współczesne, wielowarstwowe systemy informatyczne wymagają niezależnego testowania poszczególnych warstw oprogramowania. Bazodanowe systemy informatyczne wymagają testowania funkcjonalności modyfikującej bazę danych, niezależnie od tego, jaka byłaby architektura takiego systemu. Na ogół zastosowane przy budowie systemu szablony architektury i wzorce projektowe (ang. *architecture frameworks and design patterns*) pozwalają na tworzenie testów poszczególnych warstw niezależnie, z wykorzystaniem obiektów atrap, np. z użyciem modułu NMock [2]. Jednak i tak pozostaje problem automatyzacji testowania ogólnie pojmowanej warstwy dostępu do danych. Ponieważ funkcjonalność bazodanowa z natury rzeczy jest stanowa, automatyzacja testów jednostkowych takiej funkcjonalności wymaga przygotowania odpowiednich danych testowych przed realizacją testu jednostkowego i odtworzenia stanu bazy po jego realizacji.

Przedstawione narzędzia i metody testowania pozwalają na realizację bazodanowych testów jednostkowych w różnych wariantach, wykorzystując rożne poziomy abstrakcji architektury systemu, np.:

- testowanie na „niskim” poziomie, funkcjonalnie prostej warstwy dostępu do danych (np. testowanie działania obiektów „encyjnych”, wprost zapisujących/odczytujących z tablic bazy danych) i niezależne testowanie „wyższych” warstw przetwarzających, zwanych warstwami logiki biznesowej, za pomocą atrap obiektów z warstwy dostępu do danych (czyli bez bazy danych),
- testowanie jedynie warstw logiki biznesowej, operującej na bazie danych (w abstrakcji od wewnętrznie używanej warstwy dostępu do danych).

Oba warianty są możliwe do realizacji, zakładając wykorzystanie modułów odtwarzających stan bazy danych, np. tych, opartych na omawianych rozwiązaniach typu NDbUnit czy XtUnit. Pierwszy wariant jest oczywiście korzystniejszy – pozwala m.in. na tworzenie/rozwijanie/uruchamianie testów niezależnie dla obu warstw.

Prezentowany artykuł omawia dwa podejścia do procesu odtwarzania stanu bazy danych. Jedno, wykorzystujące moduł NDbUnit, bazuje na definicji i zawartości plików XML [7, 8, 9]. Drugie, bazujące na module XtUnit, opiera się na mechanizmach dwufazowego zatwierdzania transakcji, dostępnych przez usługi COM+ Enterprise Services [13, 14, 15]. Przykłady zamieszczone w artykule dotyczą rozwiązania działającego z SZBD MS SQLServer, ale można je rozszerzyć na inne platformy bazodanowe w następujący sposób:

- dla rozwiązania bazującego na NDbUnit należy zastosować uniwersalny interfejs dostępu do danych OleDb i zastosować OleDbUnitTest jako klasę obiektu sterującego zamiast SqlDbUnitTest,
- dla rozwiązania typu XtUnit należy sprawdzić wymaganie, że zainstalowany SZBD wspiera przetwarzanie transakcji rozproszonych w technologii COM+.

W artykule pokazano przykład zastosowania mechanizmu NUnit Extensions, pozwalającego na ukrycie szczegółów realizacji funkcjonalności związanych z automatycznym odtwarzaniem transakcji COM+. Wygoda wykorzystania tego mechanizmu wynika z łatwości użycia zdefiniowanego w ramach tego mechanizmu, dodatkowego atrybutu *Rollback*.

W podobny sposób można stworzyć inne rozszerzenie NUnit, wprowadzające dodatkowe atrybuty, pozwalające na równie łatwą obsługę odtwarzania bazy danych przez pliki XML, bazujące na NDbUnit (enkapsulacja użycia elementów NDbUnit). Analogicznie (przez rozszerzenie NUnit na poziomie *Test Case Builders* oraz wprowadzenie odpowiednich, własnych atrybutów) można zrealizować ukryte operacje ustawiania/odtwarzania stanu danych przez wykonanie/odtworzenie kopii bazy danych (np. z użyciem komend Transact-SQL: backup/restore dla SZBD MS SQLServer). Jednak przedstawienie odpowiednich kodów źródłowych przekracza ilościowe ograniczenia dla tego opracowania.

Omówione mechanizmy pozwalają na skuteczne wykorzystanie silnika testowego NUnit do obsługi testów jednostkowych, obejmujących funkcjonalności bazodanowe.

BIBLIOGRAFIA

1. NUnit Home Page: http://www.nunit.org.
2. NMock Home Page: http://www.nmock.org.
3. Burton K. R.: .NET Common Language Runtime Unleashed, SAMS Publishing 2002.

4. Augustyn D. R.: Rozwój narzędzi programowych wspierających automatyzację testów jednostkowych dla technologii .NET. Bazy danych. Rozwój metod i technologii. Bezpieczeństwo, wybrane technologie i zastosowania. WKŁ, Warszawa 2008.
5. ADO.NET MSDN Page, http://msdn.microsoft.com/en-us/library/aa286484.aspx.
6. W3C XML Schema Page, http://www.w3.org/XML/Schema.
7. Glover A.: NDbUnit Home Page: http://www.ndbunit.org.
8. Balsara M.: Digital Group Infotech Page, http://newsletter.thedigitalgroup.com/apr08/technical.html
9. NDbUnit Google Code Page, http://code.google.com/p/ndbunit/source/browse.
10. NUnit Addins Page: http://www.nunit.org/index.php?p=nunitAddins&r=2.4.8.
11. Hall B.: Testing Times Ahead: Extending NUnit Page, http://www.simple-talk.com/dotnet/.net-tools/testing-times-ahead-extending-nunit/.
12. Schlapsi A.: Row Test Extension Page, http://bazaar.launchpad.net/~a-schlapsi/nunit-rowtests/trunk/files.
13. Osherove R.: Simplified Database Unit testing using Enterprise Services Page http://weblogs.asp.net/rosherove/articles/DbUnitTesting.aspx.
14. Osherove R.: NUnitX and the Rollback attribute Page, http://weblogs.asp.net/rosherove/archive/2004/07/12/180189.aspx.
15. NUnit – XtUnit Page, http://www.nunit.org/index.php?p=xtunit.

Recenzent: Prof. dr hab. inż. Jerzy Klamka

Wpłynęło do Redakcji 9 lutego 2009 r.

Abstract

The paper presents practical solutions of database functionality unit testing using NUnit module [1]. Because of persistent data modifications, database functionality tests are statefull and can influence one to another. Any database functionality test requires the expected content of database before test running. By definition a unit test should be stateless and independent from each other. To satisfy these conditions a mechanism of placing database into know state and database content rollbacking should be supported.

The paper describes the solution based on two known approaches. The first one based on NDbUnit [7, 8, 9] uses XML data file and XSD schema one for modifying database. Database modification operations can be integrated with NUnit test processing for ensuring

expected database content before and after running of any tests. The second approach based on distributed transaction processing supported by COM+ Enterprise Services [13, 14, 15] lets automatically rollback any data changes made by single unit tests. In this solution any unit test transaction defined on ADO.NET level are decorated by an always-aborted COM+ transaction.

The paper shows NUnit AddIns mechanism [10, 11] which lets extend NUnit by user-defined functionality. Especially it lets define own .NET attributes which are responsible for database rollback operations [14]. Such NUnit extension lets simplify unit tests definitions by hiding implementation details of database content changes made before/after test running.

Adresy

Dariusz Rafał AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, draugustyn@polsl.pl.
Ireneusz STANEK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska.