Tworzenie interfejsów do baz danych z wykorzystaniem technologii ADO.Net

Andrzej Ptasznik
Warszawska Wyższa Szkoła Informatyki
aptaszni@wwsi.edu.pl

Streszczenie
W ramach wykładu zostaną przedstawione podstawy technologii ADO.Net, zapewniającej dostęp do baz danych, podstawy wielowarstwowej architektury aplikacji korzystających z baz danych, a także podstawowe cechy technologii Data Access Application Block, która umożliwia uproszczenie obsługi baz danych z poziomu aplikacji. Wśród poruszanych tematów znajdują się elementy LINQ (ang. *Language Integrated Query*). Poszczególne elementy technologii zostaną natomiast omówione z wykorzystaniem przykładów napisanych w języku C#. W czasie wykładu zostaną zaprezentowane również przykładowe rozwiązania wykorzystujące omawiane technologie.

Spis treści

1. Wprowadzenie 121
2. Architektura aplikacji bazodanowych 121
3. Architektura wielowarstwowa 122
4. Planowanie aplikacji bazodanowej 123
5. Podstawy ADO.Net 124
6. Typowe scenariusze dostępu do baz danych 125
7. Implementacje komponentów dostępu do baz danych 125
8. Zastosowanie Data Access Application Block (DAAB) 128
9. LING to SQL 130
Podsumowanie 133
Literatura 133

1 WPROWADZENIE

We współczesnym świecie trudno wyobrazić sobie jakikolwiek system informatyczny, który nie korzystałby z jakiegoś źródła danych. W charakterze źródła danych występują zarówno proste pliki tekstowe, pliki XML, zasoby umieszczone w Internecie (RSS, Atom), jak i relacyjne bazy danych. Te ostatnie są szczególnie popularne ze względu na dalece bardziej zaawansowane możliwości manipulowania przechowywanymi danymi. Zależnie od technologii wykorzystanej do budowy aplikacji zmieniają się także sposoby uzyskiwania dostępu do baz danych. W ramach niniejszego artykułu zajmiemy się tym zagadnieniem w kontekście .NET Framework 3.5.

W kolejnych rozdziałach są opisane zagadnienia dotyczące planowania budowy aplikacji i doboru właściwej architektury do konkretnych wymagań. Wymienione są również typowe błędy popełniane na tym etapie tworzenia aplikacji, a także ich konsekwencje. W dalszej części zajmujemy się sposobami tworzenia kodu dostępu do danych z wykorzystaniem trzech różnych podejść. W efekcie możemy zobaczyć, jakie są charakterystyczne cechy poszczególnych rozwiązań, ich wady i zalety oraz w jakich projektach można je bez obaw stosować. Ze względu na ograniczony czas wykładu, naszym celem jest jedynie zasygnalizowanie problemów oraz wskazanie przykładowych rozwiązań. Pozwoli to uświadomić słuchaczom podstawowe zagadnienia dotyczące budowania kodu dostępu do baz danych oraz uczulić ich na najpospolitsze błędy popełniane przy okazji tworzenia aplikacji bazodanowych.

2 ARCHITEKTURA APLIKACJI BAZODANOWYCH

W ciągu kilkunastu ostatnich lat burzliwy rozwój systemów informatycznych sprawił, że stają się one bardziej skomplikowane i spełniają coraz to bardziej złożone i zaawansowane wymagania. Co za tym idzie ich tworzenie nastręcza większych trudności. W związku z tym coraz istotniejsze staje się podejście do procesu wytwarzania systemu informatycznego, które ma zapewnić minimalizowanie możliwości błędnego działania systemu, a także umożliwia rozwijanie i rozbudowywanie jego funkcjonalności wraz z pojawiającymi się i zmieniającymi wymaganiami.

Kolejnym zagadnieniem jest kwestia doboru architektury rozwiązania, adekwatnej do wymagań. Pomimo istnienia dużej liczby rozwiązań, szablonów, koncepcji oraz wzorców, nie ma architektury idealnej. Do rozwiązania konkretnego problemu można zastosować wiele podejść, natomiast każde z nich niesie ze sobą swoją specyfikę, która w znacznym stopniu może rzutować na to, jak sprawdzi się przy realizacji konkretnego przedsięwzięcia. Bardzo często w praktyce okazuje się, że zbyt pochopne przyjęcie architektury rozwiązania może powodować powstawanie na różnych etapach zaawansowania projektu nieprzewidzianych problemów, które zwykle powodują powstawanie opóźnień przy realizacji harmonogramu, lub wręcz wymuszają zmianę koncepcji projektu już w trakcie jego realizacji. Tego typu zdarzenia potrafią doprowadzić nawet do upadku projektu, co nie należy do rzadkości, jeśli weźmie się pod uwagę, że w opinii wielu specjalistów tylko ok. 20-30% projektów kończy się sukcesem.

Nie należy wierzyć w istnienie jednej, najlepszej i gwarantującej sukces architektury systemu, jak i samego procesu wytwórczego. Gdyby takowe istniały, to nie byłoby problemu z realizacją projektów. Warto wspomnieć o paradoksie Cobb'a:

> *We know why projects fail, we know how to prevent their failure*
> *– so why do they still fail?*
> *[Wiemy, dlaczego projekty upadają, wiemy jak zapobiec tym upadkom*
> *– więc dlaczego one ciągle upadają?]*[1]

[1] Za: http://stakeholdermenagment.wordpress.com/2011/03/18/cobbs-paradox/, tłumaczenie autora.

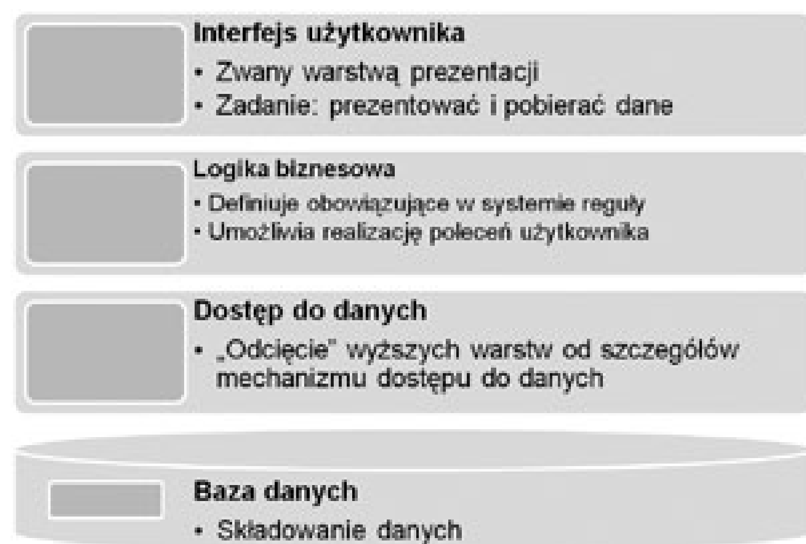
Zagadnienia związane z tym, jak podejść do realizacji konkretnego projektu, jaką zastosować metodykę, jaką architekturę rozwiązania są na tyle rozległe, że mogą stanowić temat na cały cykl publikacji, z których każda będzie przekonywać o tym, że ta konkretna metoda jest najlepsza.

Istotne także są względy ekonomiczne. Zwykle jednym z zasobów, którego jest wiecznie za mało w projekcie, jest czas. Nie zawsze da się poświęcić odpowiednią ilość czasu na dopracowanie szczegółów architektury, gdyż trzeba zaczynać realizację systemu, żeby udało się go ukończyć w rozsądnym terminie. W takiej sytuacji szuka się kompromisu i upraszcza niektóre aspekty architektury, zyskując na czasie, co jednak potrafi się okrutnie zemścić na dalszych etapach realizacji i rozwoju systemu.

W czasie tego wykładu skupimy uwagę jedynie na niewielkim wycinku architektury systemu, jakim jest kod dostępu do baz danych. Jest to jednak na tyle istotny wycinek, że z pewnością zasługuje na dokładniejsze omówienie. Dobrze zaprojektowany i zaimplementowany system jest w stanie zapewnić aplikacji wysoką stabilność i wydajność oraz odporność na zmiany wymagań (w postaci łatwości nanoszenia zmian). Dla odmiany, zaprojektowany źle – potrafi pogrzebać cały projekt.

3 ARCHITEKTURA WIELOWARSTWOWA

Typowa struktura aplikacji bazodanowej może zostać opisana modelem warstwowym, jak na rysunku 1.



Rysunek 1.
Typowa architektura aplikacji bazodanowej

Istotnym mechanizmem w ramach tej architektury jest sposób, w jaki warstwy komunikują się między sobą. Polega to na tym, że wyższa warstwa komunikuje się jedynie z leżącą bezpośrednio pod nią – czyli warstwa prezentacji korzysta z warstwy logiki biznesowej, a nie ma możliwości sięgania do warstw niższych, a nawet nie wie nic o ich istnieniu. Podobnie warstwa biznesowa korzysta z warstwy dostępu do danych nie wiedząc nic na temat samej bazy danych. Warstwa dostępu do danych korzysta z bazy danych i przekazuje jej polecenia do wykonania oraz odbiera wyniki.

Role poszczególnych warstw są ściśle określone i można je ogólnie opisać następująco:

- Warstwa prezentacji:
  - jej celem jest przedstawianie danych użytkownikowi oraz umożliwienie mu modyfikowania tych danych, a także sterowania zachowaniem aplikacji;
  - może zawierać tzw. logikę aplikacji (np. regułę, że jeżeli opcja x jest zaznaczona, to użytkownik nie widzi pól y i z).
- Warstwa logiki biznesowej:
  - zawiera kod odpowiedzialny za realizację poszczególnych operacji biznesowych (np. złożenie zamówienia, przyznanie rabatu itp.);
  - zawiera także reguły biznesowe (np. pracownik może przyznać rabat do 5%, a za zgodą menedżera do 20%, kwota na fakturze musi być większa od zera).
- Warstwa dostępu do danych:
  - zawiera kod realizujący operacje komunikowania się z bazą danych i przekazywania jej poleceń pobrania/dodania/modyfikacji/usuwania danych;
  - może zawierać także mechanizmy umożliwiające konfigurowanie dostępu do danych (wybór serwera, sposobu łączenia się z bazą itp.).
- Warstwa danych:
  - zwykle jest to serwer baz danych;
  - baza danych zawiera tabele;
  - baza może także zawierać widoki, procedury składowane, funkcje użytkownika, wyzwalacze, które służą do ukrycia szczegółów implementacyjnych bazy przed aplikacjami z niej korzystającymi, bądź do zaimplementowania części lub całości logiki biznesowej.

Zależnie od stopnia złożoności projektu, architektura może ulegać modyfikacjom polegającym na wchłanianiu wewnętrznych warstw przez skrajne. W takim przypadku warstwa logiki biznesowej bywa wplatana w kod warstwy prezentacji lub wprost w bazę danych (w postaci procedur składowanych, widoków, funkcji użytkownika i wyzwalaczy). Podobnie warstwa dostępu do danych może przestać być osobnym tworem i zostaje pofragmentowana i wkomponowana w kod warstwy prezentacji.

Takie zjawisko prowadzi jednak do powstawania aplikacji bardzo trudnych w utrzymaniu i rozwijaniu. Co z tego, że mamy jasno sformułowane wymaganie, skoro jego implementacja w aplikacji jest podzielona na kilka fragmentów osadzonych w różnych metodach obsługi zdarzeń, ewentualnie mamy do czynienia z powielaniem tego samego kodu w kilku miejscach. To wszystko razem powoduje, że bardzo trudno jest oszacować, ile czasu będzie potrzeba na realizację zadania, a także zapewnić stabilne i poprawne działanie aplikacji.

4 PLANOWANIE APLIKACJI BAZODANOWEJ

Przy planowaniu aplikacji bazodanowej, a w szczególności jej warstwy biznesowej i dostępu do danych, bardzo ważne jest wcześniejsze zebranie wymagań. Znakomicie ułatwia to proces definiowania operacji, które będą wykonywane na danych, co z kolei staje się podstawą do zdefiniowania interfejsu warstwy dostępu do danych. Interfejs ten powinien zawierać metody, które są niezbędne do wykonania wszystkich wynikających z wymagań operacji na danych. Interfejs ten będzie odtąd pełnił rolę kontraktu zawieranego pomiędzy warstwą biznesową a warstwą dostępu do danych i będzie definiował listę operacji, które będą udostępniane przez warstwę dostępu do danych, a z których będzie korzystała warstwa biznesowa.

Kolejnym krokiem jest zaprojektowanie encji biznesowych – klas, które będą nośnikiem danych. Będą zawierały wszystkie cechy informacyjne wynikające z wymagań. Dane pobierane z bazy będą przetwarzane do postaci encji biznesowych, a na nich z kolei będą operować wyższe warstwy. Przy tej okazji warto wspo-

mnieć o narzędziach, które w większym lub mniejszym stopniu automatyzują generowanie klas encji biznesowych oraz mechanizmu zasilania ich danymi z bazy oraz przenoszenia modyfikacji z encji do bazy. Jedno z takich narzędzi zostanie opisane w dalszej części rozdziału.

5 PODSTAWY ADO.NET

W systemie .NET Framework 3.5 biblioteką odpowiedzialną za udostępnianie klas, służących do organizowania dostępu do danych, jest ADO.NET. W jej ramach dostajemy do dyspozycji sporą liczbę klas, które umożliwiają realizowanie nawet bardzo złożonych mechanizmów współpracy z bazą danych. Dodatkowo, całość jest zaprojektowana w sposób ułatwiający rozszerzanie istniejących mechanizmów o nowe implementacje bez konieczności znacznych modyfikacji kodu aplikacji (model Dostawców – ang. *Providers*). Z grubsza polega on na zdefiniowaniu ogólnych interfejsów dla usług oferowanych przez dostawcę (np. połączenie z bazą powinno umożliwiać otwarcie połączenia oraz jego zamknięcie, a także zwracać informacje o jego aktualnym stanie – interfejs IDbConnection zawiera zdefiniowane metody Open(), Close() oraz właściwość State).

Każdy mechanizm służący do nawiązywania połączenia z konkretną bazą danych zawiera klasy implementujące interfejsy dostawcy. Standardowo .NET Framework zawiera dostawców:
- ODBC Data Provider;
- OLEDB Data Provider;
- SQLClient Data Provider.

Każdy z nich zawiera zestaw klas umożliwiających dostęp do różnych rodzajów źródeł danych:
- ODBC – dostęp do źródeł z wykorzystaniem ODBC (ang. *Open DataBase Connectivity*);
- OLEDB (ang. *Object Linking and Embedding, Database*) – następca ODBC, ma większe możliwości;
- SQLClient – dedykowany dostawca dla SQL Servera.

Jeżeli musimy się łączyć z inną bazą danych lub dowolnym innym źródłem – wystarczy uzyskać odpowiedniego dostawcę od producenta bazy (np. Oracle) lub wręcz napisać kod własnego dostawcy danych. Korzystać z niego będziemy dokładnie tak samo jak z innych.

.NET Framework 3.5 oferuje wiele interfejsów pomocnych przy komunikowaniu się z bazami danych. Klasy implementujące te interfejsy pełnią następujące role:
- IDbConnection – odpowiedzialna za zdefiniowanie i zarządzanie połączeniem z bazą danych;
- IDbCommand – odpowiedzialna za zbudowanie polecenia, które będzie wysłane do bazy danych za pośrednictwem połączenia;
- IDataReader – umożliwia odbieranie rezultatu wykonania polecenia przez bazę danych;
- IDbParameter – ułatwia definiowanie parametrów polecenia przekazywanego do bazy danych, lub odbierania wartości parametrów wyjściowych;
- IDataAdapter – umożliwia zdefiniowanie operacji CRUD (*Create, Read, Update, Delete*) dla określonej tabeli w bazie danych. W jej skład wchodzą cztery zestawy klas implementujących interfejsy IDbConnection oraz IDbCommand, które odpowiadają operacjom wykonywanym na danych;
- DataSet – Uniwersalny nośnik danych – umożliwia zdefiniowanie modelu danych (encje, relacje między nimi). Ma szerokie możliwości manipulowania zawartymi w sobie danymi oraz śledzenia zmian. Zwykle jest napełniany i obsługiwany za pomocą DataAdaptera. Ze względu na rozbudowaną funkcjonalność nie jest zbyt wydajny.

6 TYPOWE SCENARIUSZE DOSTĘPU DO BAZ DANYCH

W dalszych rozważaniach przyjmiemy założenie, że pracujemy z dostawcą SQLClient. Proces uzyskania dostępu do danych składa się z kilku etapów. Można je opisać w następujący sposób:
1. Utworzenie obiektu SqlCommand i przekazanie mu ciągu definiującego połączenie z bazą (ang. *connection string*).
2. Utworzenie obiektu SqlCommand, reprezentującego polecenie do wykonania. Może to być polecenie DML, DDL, DCL lub wywołanie procedury składowanej. Jeżeli to konieczne, należy do stworzonego obiektu dodać definicje parametrów wykonania polecenia.
3. Otwarcie połączenia – wykonanie metody Open() obiektu SqlCommand.
4. Wykonanie polecenia za pośrednictwem obiektu SqlConnection skojarzonego z SqlCommand. Istnieją trzy warianty wykonania polecenia:
   - ExecutenonQuery() – stosowane, gdy nie spodziewamy się zwrotnie zbioru rekordów (ang. *recordset*).
   - ExecuteScalar() – używane, gdy zwrotnie dostajemy zbiór rekordów zawierający jeden rekord z jedną kolumną. Upraszcza mechanizm „wyłuskania” zwróconej wartości.
   - ExecuteReader() – wykorzystane, gdy spodziewamy się zwrotnie zbioru (lub zbiorów) rekordów. Metoda ta zwraca obiekt SqlDataReader, który można traktować jako prosty kursor read and forward only, czyli umożliwiający iterowanie po kolejnych rekordach ze zbioru zwróconego przez polecenie. Umożliwia także na przejście do kolejnego zbioru rekordów (o ile polecenie spowodowało zwrócenie takowego).
5. Przetworzenie wyników polega zwykle na utworzeniu pętli działającej dla każdego rekordu zwróconego w zbiorze wynikowym.
6. Istotnym zagadnieniem jest zamknięcie połączenia z bazą, gdyż w przeciwnym przypadku można szybko doprowadzić do niepotrzebnego zużycia zasobów i utrzymywania niepotrzebnych już połączeń.

Drugim typowym scenariuszem jest korzystanie z obiektu DataSet i DataAdapter. Odbywa się to za pomocą dwóch metod klasy SqlDataAdapter – Fill() i Update(). Wewnętrznie jednak korzystają one z tego samego podstawowego scenariusza (stosującego SqlDataReader), który został opisany wcześniej.

7 IMPLEMENTACJE KOMPONENTÓW DOSTĘPU DO BAZ DANYCH

Opisany w poprzednim rozdziale scenariusz uzyskania dostępu do danych można zaimplementować w najprostszej postaci, jak pokazano na rysunku 2.

```
string connectionString = @"server=.\sql2008;database=schooldatabase;integrated security=true";
string sqlText = "SELECT * FROM Student ORDER BY LastName";

SqlConnection connection = new SqlConnection(connectionString);
SqlCommand command = new SqlCommand(sqlText, connection);
connection.Open();
SqlDataReader dr = command.ExecuteReader();
while (dr.NextResult())  //przejście do kolejnego recordsetu
{
    while (dr.Read())
    {
        // ...przetworzenie rekordu.....
    }
}
connection.Close();
```

Rysunek 2.

Należy zauważyć, że nie jest to dobry przykład, jak w praktyce wykonywać operacje na bazie danych! Ten kod nie uwzględnia możliwości wystąpienia błędu, a w takim przypadku połączenie z bazą może pozostać otwarte, mimo że nie będziemy już z niego korzystać.

Kwestia sensownego przechowywania zawartości łańcucha połączenia i treści polecenia również nie jest tu istotna. W prawdziwych projektach tego typu informacje są przechowywane w plikach konfiguracyjnych aplikacji, często w postaci zaszyfrowanej. Zaprezentowany kod będzie poprawnie działał, jeżeli nie wystąpią żadne nieprzewidziane sytuacje i nie nastąpi błąd, który wygeneruje tzw. wyjątek. Każdy wyjątek spowoduje, że połączenie z bazą nie zostanie automatycznie zamknięte i będzie niepotrzebnie zużywać zasoby serwera.

W celu bezpiecznego skorzystania z opisywanego scenariusza należy nieznacznie zmodyfikować jego kod, aby uwzględnić możliwość wystąpienia błędu i w sposób bezpieczny zwolnić zaalokowane zasoby, patrz rysunek 3.

```
string connectionString = @"server=.\sql2008;database=schooldatabase;integrated sec
string sqlText = "SELECT * FROM Student ORDER BY LastName";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    using (SqlCommand command = new SqlCommand(sqlText, connection))
    {
        connection.Open();
        SqlDataReader dr = command.ExecuteReader();

        while (dr.NextResult())  //przejście do kolejnego recordsetu
        {
            while (dr.Read())
            {
                // ...przetworzenie rekordu.....
            }
        }
    }
}
```

Rysunek 3.

Zastosowanie konstrukcji using gwarantuje, że w momencie wyjścia z bloku kodu:

```
using (Typ x = new Typ())
{
…
}
```

zostanie wywołana metoda Dispose() obiektu `x`, która zgodnie z przeznaczeniem powinna zawierać kod zwalniający zasoby używane przez obiekt `x`. W naszym przypadku będzie to zamknięcie połączenia z bazą. Metoda ta zostanie wywołana **ZAWSZE**. Nawet gdy wystąpi nieoczekiwany wyjątek. Wewnętrznie jest to realizowane poprzez zamianę bloku `using {…}` na blok `try{…} finally{…}`.

Wypadałoby jeszcze wspomnieć o kwestii zarządzania połączeniem z bazą danych. Tego typu zasoby są uznawane za dość kosztowne, więc należy zwrócić szczególną uwagę na sposób posługiwania się nimi. Mamy tu do czynienia z dwoma skrajnościami:

- Utworzenie połączenia, otwarcie go i trzymanie w tym stanie w trakcie działania aplikacji.
- Tworzenie połączenia za każdym razem, gdy chcemy przesłać polecenie do bazy, i niezwłoczne zamykanie go zaraz po odebraniu wyników.

Często uznaje się, że ciągłe otwieranie i zamykanie połączeń ma negatywny wpływ na wydajność ze względu na to, że utworzenie i nawiązanie połączenia z bazą jest dość kosztowne. Na szczęście SQL Server zawiera mechanizm (ang. *Connection Pooling*), który pozwala zapomnieć o problemach z wydajnością. Zamykane połączenie nie jest tak naprawdę niszczone tylko trafia do specjalnej puli, z której jest ponownie pobierane i wykorzystywane w razie potrzeby. Dzięki temu można przyjąć (dotyczy to szczególnie aplikacji typu WWW), że połączenie można śmiało tworzyć wyłącznie na czas wykonania polecenia. Do tego właśnie służy zaprezentowany kod.

Jak poradzić sobie w sytuacji, gdy trzeba zwrócić obiekt SqlDataReader do wykorzystania przez inne klasy? Co wtedy z połączeniem z bazą danych? W takim przypadku nie można zastosować konstrukcji using do zapewnienia bezpiecznego zamknięcia połączenia z bazą. Zastosowanie using powodowałoby, że SqlDataReader stałby się bezużyteczny, gdyż do pobierania kolejnych rekordów wymaga on otwartego połączenia z bazą. Jest to istotna cecha obiektu DataReader – potrzebuje on otwartego połączenia z bazą.

Aby móc bezpiecznie zamknąć połączenie po wykorzystaniu obiektu SqlDataReader w innej metodzie, należy skorzystać z parametru CommandBehavior.CloseConnection, który spowoduje automatyczne zamknięcie połączenia w momencie wywołania metody Close() lub Dispose() obiektu SqlDataReader.

```
string connectionString = @"server=.\sql2008;database=schooldatabase;integrated secu
string sqlText = "SELECT * FROM Student ORDER BY LastName";

SqlConnection connection = new SqlConnection(connectionString);
{
    using (SqlCommand command = new SqlCommand(sqlText, connection))
    {
        connection.Open();
        SqlDataReader dr = command.ExecuteReader(CommandBehavior.CloseConnection);
        return dr;
    }
}
```

Rysunek 4.

Często do polecenia wysyłanego do bazy trzeba wstawić wartości przekazane przez użytkownika. Można to zrealizować poprzez zbudowanie całego polecenia z fragmentów stałych przeplatanych wartościami podanymi przez użytkownika. Budowanie takiego polecenia wygląda na najprostszą metodę, lecz stanowi spore zagrożenie. Po pierwsze, przy bardziej złożonych poleceniach łatwo można popełnić błąd, który spowoduje, że serwer baz danych nie będzie mógł wykonać polecenia z powodu błędnej składni (niedomknięte apostrofy itp.). Także modyfikowanie zapytania stworzonego w ten sposób nastręcza wielu problemów.

Większym zagrożeniem są jednak ataki typu SQL Injection. Polegają one z grubsza na tym, że użytkownik aplikacji podaje specjalnie sformatowane wartości, które po wkomponowaniu w polecenie modyfikują jego działanie.

Aby ograniczyć ryzyko takich ataków, należy skorzystać z możliwości definiowania parametrów poleceń (SQLParameter). Polecenia tworzone w ten sposób mają postać szablonu z miejscami na wartości parametrów, do którego w następnej kolejności dodaje się kolekcje obiektów SQLParameter z przypisanymi do nich wartościami. Tak skonfigurowane polecenie wykonuje się analogicznie jak poprzednio (patrz rysunek 5).

```
string sqlText2 =
"INSERT INTO dbo.StudentNote (StudentID,SubjectID,NoteValue) VALUES ( @studentID,@subjectID,@noteValue)";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    using (SqlCommand command = new SqlCommand(sqlText2, connection))
    {
        command.Parameters.Add("@studentID", SqlDbType.Int).Value = studentId;
        command.Parameters.Add("@subjectID", SqlDbType.Int).Value = subjectID;
        command.Parameters.Add("@noteValue", SqlDbType.Decimal).Value = noteValue;

        connection.Open();
        command.ExecuteNonQuery();
    }
```

Rysunek 5.

Najbardziej eleganckim rozwiązaniem tego problemu jest korzystanie z procedur składowanych. Upraszcza to tworzony kod oraz uodparnia go na zmiany w bazie (np. zmianę postaci zapytania). W takich przypadkach kod pozostaje niezmienny do momentu, w którym zmienia się parametry wywołania procedury składowanej.

Można wtedy dodać dodatkową warstwę abstrakcji, odcinającą kod dostępu do danych od szczegółów implementacyjnych samej bazy. Specjalnym efektem jest otrzymanie aplikacji, w której można część modyfikacji wprowadzić bez dokonywania jakichkolwiek zmian w kodzie – zmieniając jedynie kod procedur składowanych w bazie. Z drugiej strony – rosnąca ilość logiki zaszytej w bazie danych w postaci skomplikowanego kodu SQL powoduje wzrost nakładów pracy, potrzebnych na jego utrzymanie i weryfikowanie poprawności działania po naniesieniu modyfikacji. Przykładowy kod wywołujący procedurę składowaną może mieć postać, jak na rysunku 6.

```
string procedureName = "pAddStudentNote";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    using (SqlCommand command = new SqlCommand(procedureName, connection))
    {
        command.CommandType = CommandType.StoredProcedure;

        command.Parameters.Add("@studentID", SqlDbType.Int).Value = studentId;
        command.Parameters.Add("@subjectID", SqlDbType.Int).Value = subjectID;
        command.Parameters.Add("@noteValue", SqlDbType.Decimal).Value = noteValue;

        connection.Open();
        command.ExecuteNonQuery();
    }
```

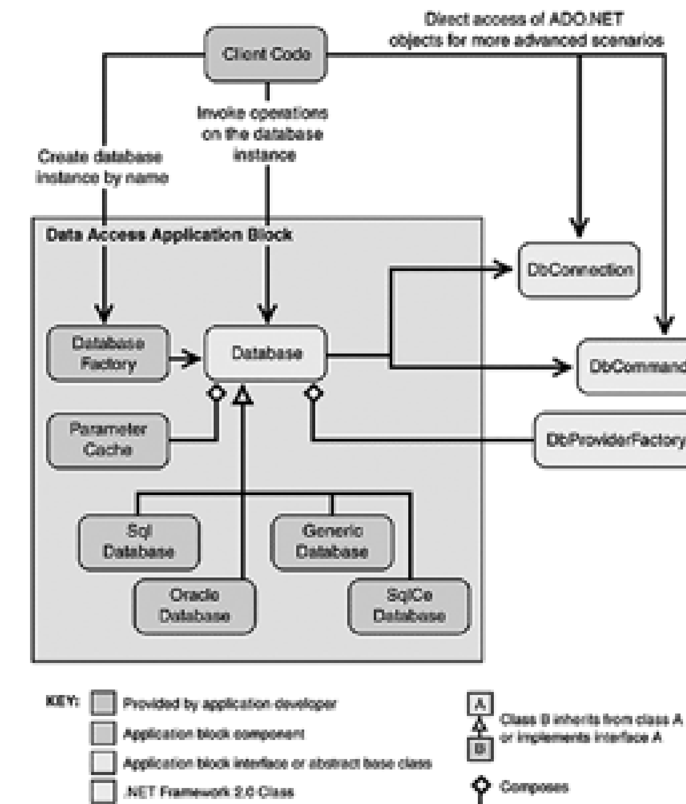
Rysunek 6.

W powyższym przykładzie zakładamy, że w bazie danych istnieje procedura składowana o nazwie pAddStudentNote, która wymaga podania wartości trzech parametrów @studentID, @subjectID oraz @noteValue. Istotne jest to, że z punktu widzenia aplikacji nieistotna jest wiedza o sposobie działania tej procedury i szczegółach jej operacji wykonywanych w bazie danych.

8 ZASTOSOWANIE DATA ACCESS APPLICATION BLOCK (DAAB)

Opisane w poprzednim rozdziale metody dostępu do danych dość dobrze ilustrują koncepcję, jaka przyświecała autorom ADO.NET, ale nie do końca nadają się do zastosowań praktycznych. Przede wszystkim ze względu na znaczną ilość kodu, który trzeba napisać, żeby zaimplementować pożądaną funkcjonalność.

Na szczęście istnieją inne rozwiązania, które choć wewnętrznie korzystają dokładnie z tych samych mechanizmów, to od strony programisty znacznie upraszczają pracę z bazą danych. Do tego typu rozwiązań należy Data Access Application Block z pakietu Enterprise Library. Jego ogólna koncepcja jest następująca (patrz rysunek 7):



Rysunek 7.
Schemat architektury DAAB [źródło: http://cdiban.wordpress.com/page/2/]

- kluczowym elementem jest obiekt typu Database, którego instancji nie tworzy sam programista, tylko specjalna klasa DatabaseFactory. To ona jest odpowiedzialna za odnalezienie w pliku konfiguracyjnym aplikacji informacji o sposobie łączenia się z bazą danych oraz o rodzaju bazy danych (SQL Server, Oracle itp.);
- obiekt typu Database udostępnia wiele metod służących do przekazywania poleceń do bazy danych i odbierania wyników;
- w razie potrzeby [bardziej wyrafinowane operacje na bazie: organizowanie transakcji, ustawianie parametrów połączenia (ang. *timeout*) i polecenia] można uzyskać dostęp do używanych wewnętrznie przez obiekt typu Database obiektów DbConnection, DbCommand;
- dodatkowo DAAB zawiera wbudowane mechanizmy powodujące wzrost wydajności (np.: cache dla obiektów SqlParameter wykorzystywanych przy wywoływaniu procedur i poleceń z parametrami).

Kod, który trzeba napisać, żeby wykonać polecenie bazodanowe przy użyciu DAAB, jest bardzo prosty i w sporej części przypadków składa się z jednego wiersza. Programista nie musi pilnować procesu nawiązywania połączenia z bazą oraz jego zamykania, a także obsługi błędów z tym związanych. Nie musi także definiować

parametrów poleceń – wystarczy przekazanie kolejnych wartości parametrów jako argumentów wywołania metody obiektu klasy Database. DAAB przed wykonaniem polecenia sam pobierze z bazy informacje na temat parametrów konkretnej procedury, stworzy odpowiednie obiekty SqlParameter i przypisze im wartości. Do tego utworzone obiekty zostaną umieszczone w pamięci cache i przy kolejnych wywołaniach tej procedury będą gotowe do użycia (wzrost wydajności). W tworzonym kodzie nie ma ŻADNYCH informacji o docelowej bazie danych i jej typie – zawiera on jedynie informacje na temat poleceń, które trzeba wykonać. Raz napisany i skompilowany kod może być wykorzystywany do komunikacji z dowolną bazą danych w dowolnej aplikacji. To właśnie aplikacja dostarcza informacji na temat typu i lokalizacji serwera baz danych, z którym będzie się komunikować. Przykładowy kod korzystający z DAAB jest przedstawiony na rysunku 8.

```
private Database database = DatabaseFactory.CreateDatabase("StudentNotesDBConnectionString");

public IDataReader GetStudents()
{
    return database.ExecuteReader(CommandType.Text, "SELECT * FROM Student ORDER BY LastName");
}

public void AddNote(int studentID, int subjectID, decimal noteValue)
{
    database.ExecuteNonQuery("pAddStudentNote", studentID, subjectID, noteValue);
}
```

Rysunek 8.

Na tym przykładzie łatwo zauważyć, jak bardzo zmniejsza się ilość tworzonego kodu. Dodatkowo programista koncentruje się na samym poleceniu i ewentualnych parametrach jego wywołania. Cała reszta jest załatwiana na poziomie konfigurowania aplikacji.

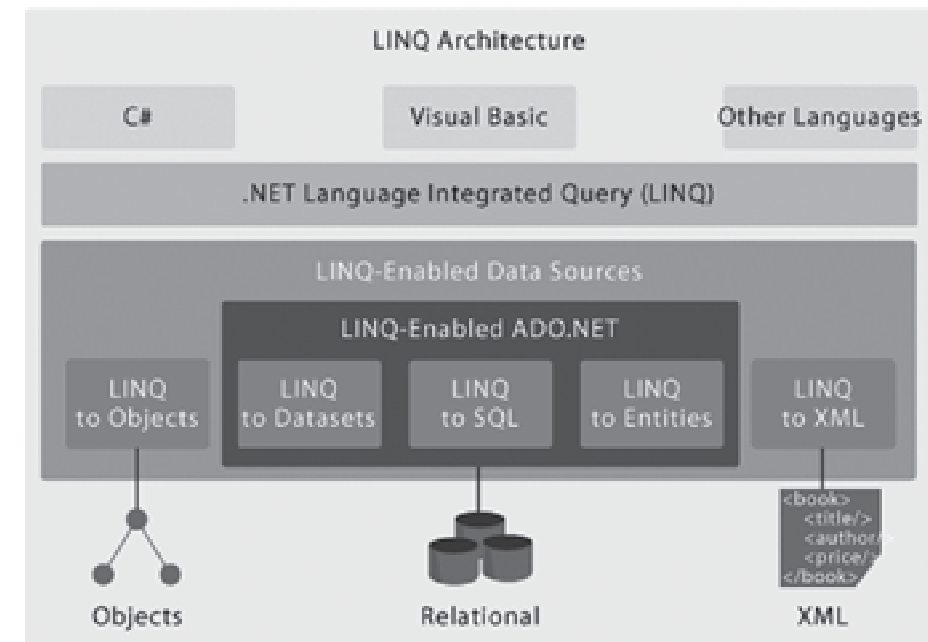
9 LINQ TO SQL

Przy komunikowaniu się z bazą danych zwykle występował problem budowania zapytań. Niezależnie od języka programowania, w którym tworzona była aplikacja, język zapytań do bazy (czy innego źródła danych) był zawsze odrębny i stanowił „wyspy” w kodzie aplikacji. To zjawisko dało się zauważyć w obu zaprezentowanych do tej pory przykładach w postaci poleceń SQL budowanych w mniej lub bardziej złożony sposób.

LINQ (ang. *Language Integrated Query*), jak sama nazwa wskazuje, stanowi odejście od tej koncepcji i wprowadza język budowania zapytań zintegrowany z językiem programowania. Występuje w kilku wariantach umożliwiających korzystanie z różnych źródeł danych (relacyjnych baz danych, XML, obiektów DataSet, innych obiektów).

W ramach wykładu skorzystamy z jednego z wariantów – LINQ to SQL, który można sklasyfikować jako O/RM (ang. *Object/Relational Mapping*). Rozwiązanie to znakomicie ułatwia i przyspiesza tworzenie warstwy dostępu do danych, a wręcz zwalnia programistę z tworzenia jakichkolwiek zapytań SQL.

Głównym elementem biblioteki stworzonej za pomocą LINQ to SQL jest **model**. Tworzy się go poprzez dodanie do projektu szablonu LINQ to SQL Classes. Dalsza praca z modelem polega na przeciąganiu i upuszczaniu obiektów z bazy danych (z narzędzia Server Explorer/Data Connections), co powoduje generowanie w tle potrzebnych klas. Modyfikacja modelu zwykle ogranicza się do dopasowania nazw wygenerowanych typów, uzupełnienia ich o dodatkowe metody i właściwości oraz skonfigurowania ich zachowania przy komunikacji z bazą danych (patrz rysunek 10).



Rysunek 9.
Architektura LINQ [źródło http://www.codeproject.com/KB/linq/UnderstandingLINQ.aspx]

Rysunek 10.
Tworzenie modelu LINQ

W wygenerowanym modelu można także tworzyć relacje pomiędzy encjami. Nie muszą one odpowiadać kluczom obcym w bazie. Można je budować w sposób dowolny. Klucze obce są natomiast podstawą do automatycznego wygenerowania relacji. Efektem istnienia relacji pomiędzy encjami jest pojawienie się w nich dodatkowej właściwości zawierającej referencję do encji znajdującej się na drugim końcu relacji. Przykładowo klasa Student będzie miała właściwość StudentNotes prowadzącą do listy ocen studenta, a każda ocena będzie miała (zależnie od tego czy sobie tego życzymy) referencję do encji Studenta, do którego ocena należy.

Utworzenie modelu daje efekt w postaci wygenerowanych klas. Podstawową jest klasa DataContext, której instancję tworzy się zawsze, gdy chce się skorzystać z danych. Należy zaznaczyć, że zawiera ona wiele dodatkowych metod, właściwości i zdarzeń pozwalających na operowanie na danych oraz śledzenie zmian.

```
public class StudentNotesDBLINQ
{
    private StudentNotesDataContext context = new StudentNotesDataContext();

    public IQueryable<Student> GetStudents()
    {
        return context.Students;
    }

    public void AddNote(int studentId, int subjectID, decimal noteValue)
    {
        context.AddStudentNote(studentId, subjectID, noteValue);
    }

    public decimal GetAverageNoteForStudent(int studentID)
    {

        Student selectedStudent = context.Students.Where(s => s.StudentID == studentID).SingleOrDefault();
        if (selectedStudent == null)
            return 0;
        return selectedStudent.StudentNotes.Average(sn => sn.NoteValue);
    }
```

Rysunek 11.

LINQ to SQL znakomicie upraszcza i przyspiesza proces tworzenia kodu organizującego dostęp do bazy danych. Korzysta z dobrych praktyk i jest zaprojektowane tak, aby zapewnić wysoką wydajność. Polecenia, które LINQ to SQL przekaże do bazy danych oraz momenty, w których dojdzie do takiego przekazania są dobierane w taki sposób, żeby zapewnić jak najlepszą wydajność oraz by sięgać do danych tylko, gdy są one bezpośrednio potrzebne. W powyższym przykładzie, wykonanie metody GetStudents() nie powoduje wykonania żadnego polecenia w bazie. Zwraca ono tylko obiekt, którego można dalej używać, wywołując na jego rzecz kolejne metody – tak jak ma to miejsce na przykładzie na rysunku 12:

```
public partial class StudentList : System.Web.UI.Page
{
    protected StudentNotesDBLINQ service = new StudentNotesDBLINQ();

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            studentsGrid.DataSource = service.GetStudents().ToList();
            studentsGrid.DataBind();
        }
    }

    protected void studentsGrid_Sorting(object sender, GridViewSortEventArgs e)
    {
        if (e.SortExpression == "LastName")
            studentsGrid.DataSource = service.GetStudents().OrderBy(s => s.LastName).ToList();
        if (e.SortExpression == "FirstName")
            studentsGrid.DataSource = service.GetStudents().OrderBy(s => s.FirstName).ToList();
        studentsGrid.DataBind();

    }
```

Rysunek 12.

Dopiero wykonanie metody ToList() powoduje scalenie wszystkich dotychczasowych wywołań i wygenerowanie polecenia SQL, które jest niezwłocznie przekazane do bazy.

Dzięki silnej kontroli typów w LINQ to SQL programiści mogą uniknąć popełniania błędów, które ujawnią się dopiero po uruchomieniu aplikacji. Przykładowo przy korzystaniu z SqlDataReadera, jeżeli popełnimy błąd literowy w nazwie kolumny lub wykonamy rzutowanie na niewłaściwy typ – efektem będzie powstanie wyjątku w trakcie działania aplikacji. W przypadku LINQ błąd pojawi się już na etapie kompilacji.

LINQ to SQL jest z powodzeniem stosowany przy tworzeniu prostych projektów RAD (ang. *Rapid Application Development*) dzięki swojej prostocie. Sprawdza się w większości prostych systemów. Ograniczenie do SQL Servera 2005 i 2008 zwykle nie powoduje problemów. Jeżeli natomiast trzeba łączyć się z innym serwerem baz danych lub chce się operować na bardziej abstrakcyjnym modelu, to można skorzystać ze „starszego brata" LINQ to SQL – ADO.NET Entity Framework. Jest on znaczniej bardziej skomplikowany, ale po nabraniu wprawy umożliwia równie szybkie tworzenie aplikacji.

PODSUMOWANIE

Zaprezentowane na tym wykładzie rozwiązania stanowią tylko część istniejących narzędzi służących do organizowania dostępu do danych. W Internecie można znaleźć wiele metod umożliwiających realizowanie tego celu w różny sposób. Są tam rozwiązania bardzo proste, ale nie brak też bardzo rozbudowanych i skomplikowanych. Każde z nich ma grupę swoich zwolenników i przeciwników, a Internet jest pełen informacji na temat tego, jak dobre/złe jest konkretne rozwiązanie.

Przy podejmowaniu decyzji o zastosowaniu takiego czy innego rozwiązania trzeba brać pod uwagę wiele czynników. Najważniejsze z punktu widzenia programisty są łatwość nauczenia się danego rozwiązania oraz dostępność dokumentacji i szeroko rozumianego wsparcia. Z punktu widzenia projektanta czy architekta brane pod uwagę są inne cechy – m.in. skalowalność, koszty, to czy projekt jest nadal rozwijany itd.
Wśród innych przykładów mechanizmów dostępu do danych można wymienić:

- **Subsonic** (http://subsonicproject.com)
- **nHibernate** (https://www.hibernate.org/343.html)
- **ADO.NET Entity Framework** (http://msdn.microsoft.com/en-us/library/bb399572.aspx)

Warto zapoznać się z ich możliwościami zanim podejmie się decyzję dotyczącą koncepcji mechanizmu organizowania dostępu do danych w realizowanym projekcie. Świadoma i przemyślana decyzja w tym zakresie może oszczędzić wielu godzin pracy programistom, a także przyczynić się w znacznym stopniu do sukcesu projektu.

LITERATURA

1. Chappell D., *Zrozumieć platformę .NET.*, Helion, Gliwice 2007
2. Matulewski J., Orłowski S., *ASP.Net i ADO.Net w Visual Web Developer,* Helion, Gliwice 2007
3. Matulewski J., *C# 3.0 i .Net 3.5 Technologia LINQ*, Helion, Gliwice 2008
4. Michelsen K., *Język C# Szkoła programowania*, Helion, Gliwice 2007
5. Vieira R., *SQL Server 2005. Programowanie. Od Podstaw,* Helion, Gliwice 2007