STUDIA INFORMATICA 2013
Volume 34 Number 1 (110)

Michał ŚWIDERSKI, Piotr BUGLA
Politechnika Śląska, Instytut Informatyki

PORÓWNANIE MECHANIZMÓW DOSTĘPU DO DANYCH LINQ I ADO.NET NA PRZYKŁADZIE APLIKACJI WEBOWEJ BUDOWANEJ W MODELU SPIRALNYM

Streszczenie. Technologia LINQ jest jednym z przedstawicieli mapowania relacyjno-obiektowego. Praca stawia tezę, że technologia ta zmniejsza liczbę błędów w dostępie do bazy danych oraz ułatwia tworzenie oprogramowania. W pracy starano się udowodnić słuszność tej tezy, porównując technologie dostępu do danych LINQ oraz ADO.NET na przykładzie aplikacji webowej. Celem pracy było napisanie dwóch aplikacji przy wykorzystaniu obydwu dostępów do bazy danych. Tak postawione założenia pozwoliły jasno stwierdzić słuszność tezy. Wnioski płynące z pracy potwierdzają, że technologia LINQ zwiększa efektywność pisania kodu. Również interakcja z bazą danych za pomocą samego języka programowania zmniejsza liczbę błędów w dostępie do bazy danych. Praca wykazała także wadę LINQ, jaką jest niższa wydajność. Można jednak stwierdzić, że niższa wydajność LINQ nie jest aż tak wygórowaną ceną, w zamian za wiele pożytecznych cech LINQ.

Słowa kluczowe: LINQ, ADO.NET, ORM

COMPARISON OF LINQ AND ADO.NET DATABASE ACCESSING MECHANISMS ON THE GROUND OF WEB APPLICATION BUILT IN A SPIRAL MODEL

Summary. LINQ technology is one of the representatives of the object-relational mapping. Article proposes the thesis that this technology reduces the number of errors in the access to the database and facilitates the creation of software. The study attempted to prove the validity of this thesis by comparing the data access technologies LINQ and ADO.NET on the ground of a Web application. The aim of this study was to write two applications using both approaches, what helped to state the validity of the thesis. The conclusions of the study confirm that the LINQ technology increases the efficiency of writing code. Also interaction with the database using the programming language itself, reduces the number of errors in the access to

the database. Work has also shown the only disadvantage of LINQ, which is being less efficient than ADO.NET. However, we claim that the lower performance is not as expensive price to pay, if one takes many advantages of LINQ technology into account.

Keywords: LINQ, ADO.NET, ORM

1. Wprowadzenie

Wraz ze wzrostem wydajności komputerów rozwija się podejście do tworzenia oprogramowania. Oprogramowanie staje się coraz bardziej skomplikowane i wolniejsze, ze względu na zwiększającą się funkcjonalność i bezpieczeństwo. Drugim problemem jest natomiast dążenie do maksymalnego skrócenia czasu tworzenia oprogramowania. Stosuje się obecnie języki wysokiego poziomu, często także aplikacje wspomagające proces programowania, jednocześnie obniżając wymagania względem efektywności i wydajności działania. Niniejsza praca będzie się głównie zajmować tym drugim aspektem, na poziomie współpracy między aplikacją a bazą danych. Jedną z kluczowych składowych nowo powstających systemów jest baza danych. Programiści skupiają swoją uwagę na schemacie bazy danych i muszą pamiętać bądź często doszukiwać się nazw występujących w schemacie. Powstałe w ostatnim czasie technologie ORM (mapowanie relacyjno-obiektowe) próbują rozwiązać ten problem. Zakładają, że programista nie powinien skupiać się na detalach związanych ze schematem bazy danych, by pisać aplikacje. Może skierować swoją uwagę na zadania postawione przed nim, a nie na uzyskaniu dostępu do danych. Zakłada się bardziej intuicyjne podejście do wytwarzania oprogramowania: główną zasadą tych technologii jest założenie, że odczyt i zapis danych odbywa się poprzez model obiektowy, będący odzwierciedleniem relacyjnej bazy danych.

2. Założenia pracy

Zadaniem niniejszej pracy było napisanie dwóch aplikacji z wykorzystaniem mechanizmów: LINQ [1] i ADO.NET [2]. Tak powstające merytorycznie bliźniacze oprogramowanie, miało być porównywane pod kątem inżynierii oprogramowania. Należało się przyjrzeć, jak wybrana technologia wpływa na cykl życia oprogramowania (specyfikacja, projektowanie, implementacja, integracja, ewolucja). W trakcie analizy zadania powstały dwie alternatywne koncepcje implementacji. Pierwszą możliwością było napisanie dwóch całkiem oddzielnych aplikacji, skutkiem czego mogłaby powstać duża różnica w architekturze obydwu aplikacji. Taki sposób mógłby faworyzować technologię LINQ, która łatwo może się przeplatać z kodem aplikacji i nie wymaga wysublimowanej architektury.

Alternatywą dla tego typu podejścia była strategia polegająca na ujednoliceniu, w pewnym stopniu, mechanizmów w warstwie dostępu do danych. Taki krok pozwoliłby na zastosowanie wspólnego interfejsu użytkownika oraz bardziej przemyślanej architektury, wspólnego cyklu życia obu aplikacji (w fazach specyfikacji oraz projektowania, oraz częściowo w pozostałych). Rozwiązanie takie może jednak przyćmić niektóre potencjalne zalety technologii LINQ. Ostatecznie wybór padł na drugą opcję, w głównej mierze z uwagi na fakt mniejszego nakładu pracy i identycznego kontekstu dostępu do danych, co znacząco ułatwia porównanie technologii.

3. Model spiralny

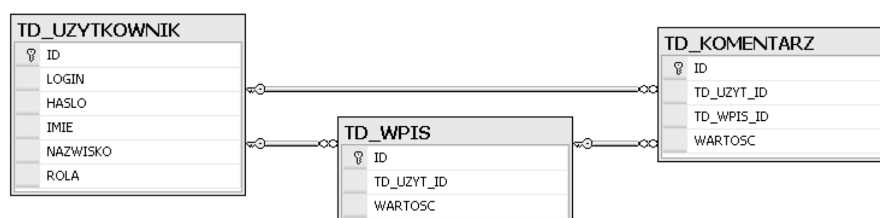
Model spiralny [3] [4] jest procesem wytwarzania oprogramowania, który ewoluował na przestrzeni lat, korzystając z doświadczeń przy udoskonalaniu modelu kaskadowego. Ma postać spirali, w której każdy cykl reprezentuje jedną fazę procesu. Cykl rozpoczyna się od identyfikacji celów dotyczących części produktu. Następnie należy zidentyfikować alternatywne sposoby realizacji oraz ograniczenia nałożone na aplikację. Kolejnym krokiem jest rozpoznanie ryzyka oraz zagrożeń: proces ten sprowadza się do oceny alternatyw w stosunku do celów i ograniczeń, często proces ten będzie identyfikować obszary niepewności, które są istotnymi źródłami ryzyka projektu. Następny krok powinien obejmować sformułowanie opłacalnych strategii rozwiązywania źródeł ryzyka (np. prototypowanie, symulacje, analizy porównawcze, modelowanie analityczne lub kombinacje tych i innych technik). Te działania prowadzą do powstania prototypu, który jest już operacyjnie przydatny i wystarczająco silny, aby służyć jako podstawa przyszłego rozwoju produktu zapewniająca niskie ryzyko. Taki prototyp będzie ewoluował w kolejnych iteracjach. Ważną cechą modelu spiralnego jest fakt, że każdy cykl jest zakończony przeglądem z udziałem głównych zainteresowanych, czyli ludzi lub organizacji związanych z produktem.

4. Realizacja aplikacji w modelu spiralnym

Rozwój aplikacji jest powiązany z szeregiem decyzji. To od podjęcia takich postanowień, jak wybór architektury, technologii czy kształtu schematu bazy danych może zależeć to, w jakim stopniu projekt będzie łatwy w utrzymaniu. Przyjecie pewnych założeń może mocno utrudnić rozwój aplikacji oraz wygenerować w przyszłości wysokie koszty. W skrajnych przypadkach może się okazać, że rozwój w pewnym kierunku jest wręcz niemożliwy. W naszych aplikacjach, w czasie pierwszych cykli spirali celowo popełniono kilka istotnych

błędów (w modelu danych), co miało na celu sprawdzenie mechanizmów LINQ i ADO.NET w sytuacji, w której takie błędy należy poprawić, by móc rozwijać nadal aplikację.

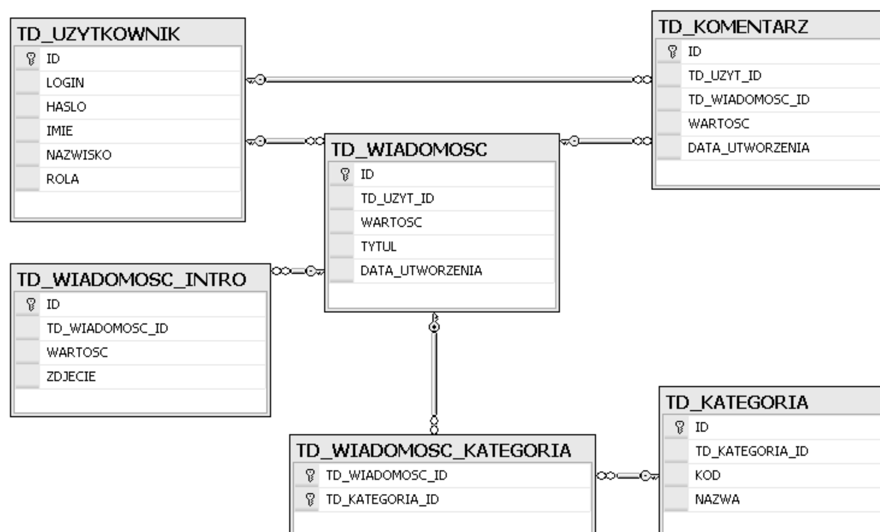
Do porównania mechanizmów wybrano zadanie stworzenia portalu internetowego w trzech iteracjach. W pierwszej iteracji założono prostotę portalu, co spowodowało, że aplikacja miała wiele ograniczeń w schemacie bazy danych (rys. 1) (m.in. długość pól informacyjnych, brak zarządzania rolami i uprawnieniami użytkowników)**.**



Rys. 1. Schemat bazy danych w pierwszej iteracji
Fig. 1. Database schema in the first development iteration

Czas poświęcony na zbudowanie obu warstw dostępu do danych był podobny i wyniósł odpowiednio 7 godzin dla LINQ oraz 10 godzin dla ADO.NET.

Kolejny cykl nie przyniósł poprawy poprzednich błędów oraz zaniechań, jakie poczyniono. Postanowiono natomiast rozbudować portal (rys. 2) o nowe funkcjonalności: budowa strony głównej, dodanie kategorii wiadomości, wiadomości otrzymały nowe cechy (m.in. data oraz temat), przeprowadzono refaktoring w postaci zmiany nazw tabel oraz kolumn.

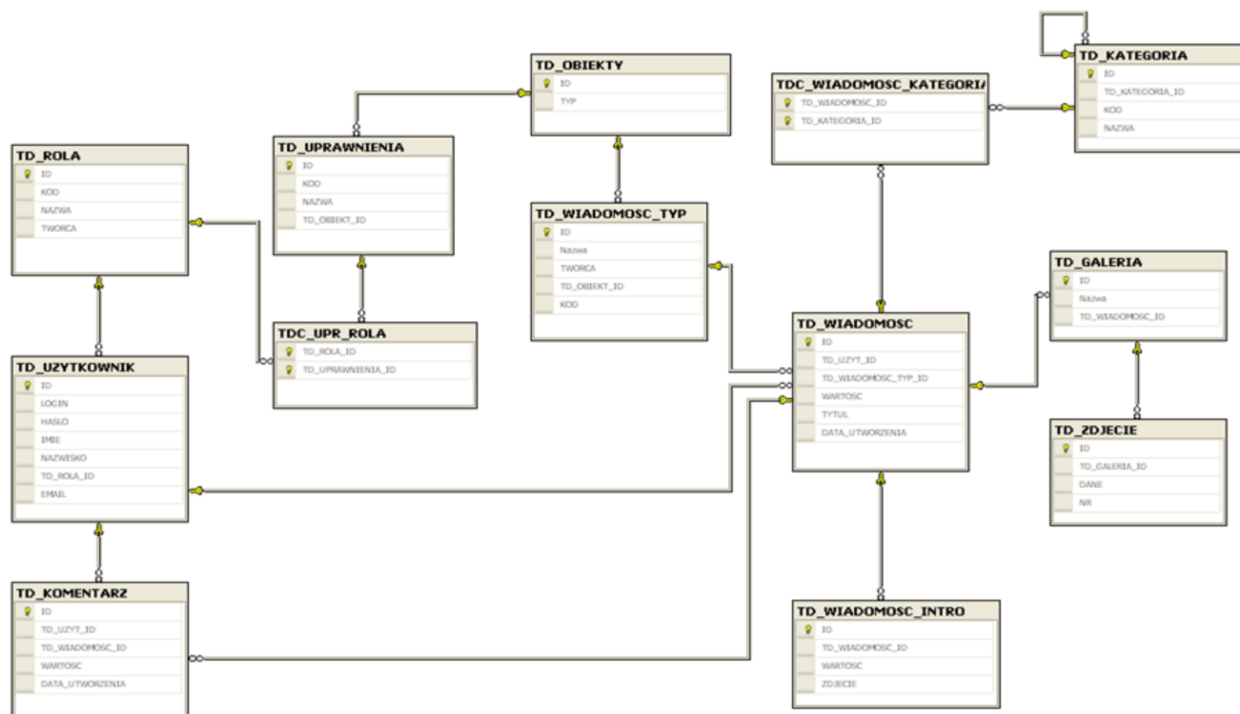


Rys. 2. Schemat bazy danych w drugiej iteracji
Fig. 2. Database schema in second development iteration

Czas poświęcony na modyfikacje warstw dostępu do danych wyniósł: 4 godziny dla LINQ oraz 10 godzin dla ADO.NET.

W trzecim przebiegu spirali poprawiono błędy popełnione we wcześniejszych cyklach (rys. 3). Dodano również nowe funkcjonalności, które pośrednio wynikają z konieczności

poprawy poprzednich błędów, m.in.: zmiana typów oraz długości typów danych niektórych kolumn. Poprawiono także wcześniejsze błędy związane z zarządzaniem rolami, uprawnieniami oraz użytkownikami (przebudowa dużej części schematu).



Rys. 3. Schemat bazy danych w trzeciej iteracji
Fig. 3. Database schema in the third development iteration

W iteracji trzeciej znów lepiej sprawdziło się LINQ. Czas, jaki trzeba było poświęcić na modyfikację warstw dostępu do danych, wyniósł: 5 godzin dla LINQ oraz 12 godzin dla ADO.NET.

5. Testowanie

W opracowywanej aplikacji skupiono się na testach jednostkowych warstwy dostępu do danych. Dzięki zastosowaniu wzorca Strategii, testowanie oznaczało napisanie jednego testu dla każdej klasy, który sprawdzał poprawność kodu napisanego zarówno w LINQ, jak i kodu napisanego w ADO.NET. Testy te sprawdzają poprawność takich operacji, jak: wstawienie, pobieranie oraz usuwanie danych oraz innych metod klas dostępu do danych. Pozostałe funkcjonalności poddano testom czarnej skrzynki, które dają większą szansę wykrycia błędnej implementacji, ale jednocześnie nie dostarczają precyzyjnej informacji na temat przyczyny wystąpienia błędu w programie [5]. Z testów automatycznych okazało się, że: używając LINQ popełnia się niewiele błędów, ponieważ odpadają wszelkie drobne pomyłki,

takie jak np. literówki. Z kolei dla ADO.NET testy automatyczne były nieocenione, ponieważ wykazały wiele błędów, takich jak literówki oraz błędy w zapytaniach.

6. Szczegółowe porównanie LINQ i ADO.NET

Poniżej zaprezentowano wnioski płynące z prac nad trzema cyklami implementacyjnymi, a także porównanie wydajności technologii oraz rozważanie nt. przyszłości ORM.

6.1. Wnioski płynące z pierwszego cyklu spirali

Pierwszą różnicą między LINQ oraz ADO.NET jest ilość kodu, którą trzeba napisać, by osiągnąć ten sam cel. Przykładem na pracochłonność może być pobranie pojedynczego rekordu z tabeli TD_WPIS:

```
//LINQ:
var wpis = Helper.Instance.DataContext.TD_WPIs.First(x => x.ID == id);
//Zawartość wiadomości
wpis.WARTOSC

//ADO.NET:
DataSet ds = new DataSet()
SqlDataAdapter sda = new SqlDataAdapter("SELECT * FROM TD_WPIS WHERE ID = @id", conn)
sda.SelectCommand.Parameters.AddWithValue("id", id);//Paramery chronią przed SQL-Injection
sda.Fill(ds);
DataRow row = ds.Tables[0].Rows[0];
//Zawartość wiadomości
row["WARTOSC"].ToString()
```

W tym miejscu warto również zaznaczyć, że LINQ nie wymaga od programisty ścisłej znajomości schematu bazy danych, włączając w to dokładne nazwy kolumn i ich typy. Takie rozwiązanie nie rozprasza programisty, który skupia się nie na technicznym, lecz na merytorycznym problemie.

Kolejną wadą w ADO.NET na tle LINQ, jest duża ilość logicznie istotnego kodu, który jest zapisywany jako ciąg znaków – są to m.in. zapytania, nazwy kolumn oraz nazwy parametrów. Takie podejście może prowokować programistę do stylu programowania typu *hard-coding*[1].

Również dodanie nowego rekordu do bazy danych (podobnie ma się sprawa z edycją danych) jest prostsze przy użyciu LINQ niż przy użyciu ADO.NET: wystarczy stworzyć nowy obiekt encji (klasy) i wypełnić go danymi, a następnie użyć metody *InsertOnSubmit* danej tabeli. Z kolei w ADO.NET wiersz można wstawić na kilka sposobów. Pierwszym z nich jest

[1] Praktyka programowania, która polega na bezpośrednim podaniu pewnej wartości zamiast użycia nazwy symbolicznej dla danych używanych często, mogących zmienić się w późniejszym czasie.

użycie klasy *SqlComand,* która przyjmuje odpowiedni tekst komendy[2] oraz posiada odpowiednio skonstruowane i wypełnione parametry. By zapytanie zostało wykonane, należy wywołać metodę *ExecuteNonQuery*. Poniżej przedstawiono kod, który wstawia nowy wiersz do tabeli TD_WPIS.

```
//LINQ:
Helper.Instance.DataContext.TD_WPIs.InsertOnSubmit(new TD_WPI()
{
    ID = Guid.NewGuid(),
    WARTOSC = wpis,
    TD_UZYT_ID = idUzytkownika
});

//ADO.NET:
SqlCommand cmd = new SqlCommand(@"
INSERT INTO
TD_WPIS(ID, WARTOSC, TD_UZYT_ID)
VALUES(@id, @wartosc, @tdUzytId)", conn);
cmd.Parameters.AddWithValue("id", Guid.NewGuid());
cmd.Parameters.AddWithValue("wartosc", wpis);
cmd.Parameters.AddWithValue("tdUzytId", idUzytkownika);
cmd.ExecuteNonQuery();
```

Podejście, które zostało przedstawiane dla ADO.NET, jest nieco naiwne. Lepszym rozwiązaniem w przypadku ADO.NET jest użycie klasy *SqlDataAdapter*. Klasa ta pozwala pracować na obiekcie *DataSet*, a nie bezpośrednio na bazie danych, tak jak w poprzednim przykładzie. Pozwala to na pobranie z bazy danych więcej niż jednej tabeli. Właściwości *SelectComand*, *InsertCommand*, *UpdateCommand* i *DeleteCommand* odpowiadają za pobieranie oraz modyfikację danych przeprowadzaną na obiekcie (odpowiedniki operacji SELECT, INSERT, UPDATE i DELETE w języku SQL). Cechą takiego rozwiązania jest fakt, że dane mogą być przechowywane w pamięci w postaci obiektu klasy *DataSet* i mogą odwzorować stan bazy danych. Programista musi być świadomy, jakie dane i jak długo przechowuje w pamięci aplikacji, jednak jest to rozwiązanie bardziej elastyczne niż w przypadku LINQ, gdzie nad pamięcią podręczną mamy praktycznie bardzo małą kontrolę.

W LINQ pobranie rekordu/encji można napisać w jednej linii kodu dzięki metodom rozszerzającym, które opierają się na wyrażeniach lambda. Wyrażenia lambda to nic innego jak metody anonimowe, tylko zapisane w bardziej skondensowanej składni. Wyrażenie lambda jest zapisywane jako lista parametrów poprzedzająca znak „=>” oraz jako blok instrukcji zaraz za nimi. Niektóre wyrażenia lambda mają nazwy zgodne z ich przeznaczeniem:

[2] W tym wypadku INSERT.

- Predykat jest logicznym wyrażeniem, które jest przeznaczone do oznaczania przynależności elementu w grupie, np. jest używany do określenia, jak filtrować elementy wewnątrz pętli.
- Projekcja jest wyrażeniem, które służy do prezentacji zwracanej kolekcji w nowej formie.

```
var sensacje = Helper.Instance.DataContext.TD_WIADOMOSCs
    .Where(w => w.TYTUL.Contains("sensacja"))   //predykat
    .Select(w => w.TYTUL);                      //projekcja
```

W języku C# zaimplementowano zbiór metod operujących na kolekcjach, jako rozszerzenia dla interfejsów *IQueryable* oraz *IEnumerable*, co oznacza, że mogą być używane z każdą z technologii z rodziny LINQ. Zbiór jest bardzo bogaty, zawiera prawie czterdzieści metod. Ponadto, w Internecie można znaleźć wiele darmowych bibliotek tego typu, jeszcze bardziej rozszerzających możliwości LINQ [6]. Jednak wyrażenia lambda to nie jedyny i nie największy atut LINQ. W takich językach, jak C# oraz Visual Basic postanowiono wbudować język zapytań w język programowania. Programista może pisać kod zbliżony składnią do języka SQL. Mamy takie frazy, jak FROM, JOIN, WHERE, SELECT. Programista znający składnię języka SQL, może się nauczyć posługiwać tą składnią w jeden dzień. Z kolei kompilator tłumaczy taką składnię na kod zbudowany z metod-rozszerzeń LINQ. Niech przykładem będzie zapytanie pobierające informację, w jakich kategoriach pisali autorzy:

```
var res = (from w in dc.TD_WIADOMOSCs
join tdc in dc.TD_WIADOMOSC_KATEGORIAs on w.ID equals tdc.TD_WIADOMOSC_ID
join k in dc.TD_KATEGORIAs on tdc.TD_KATEGORIA_ID equals k.ID
select new { w.TD_UZYTKOWNIK.LOGIN, k.NAZWA });
```

Takie zapytanie jest bardzo użyteczne: nie dość, że możemy pisać zapytania w kodzie, to w dodatku możemy sterować typem zwracanego wyniku. Dzieje się tak dzięki typom anonimowym, które tworzymy na etapie projekcji.

6.2. Wnioski płynące z drugiego cyklu spirali

W drugim cyklu spirali poczyniono zmiany oraz dodano nowe funkcjonalności, które miały wykazać wyższość jednej z technologii. Pierwszą z nich była refaktoryzacja kodu – zmiana nazw tabel oraz kolumn po stronie bazy danych. Miało to na celu ukazanie, która technologia jest w tej kwestii bardziej przyjazna. W LINQ to SQL, by zmienić nazwę tabeli (lub kolumny) na odpowiadającą jej w bazie danych, wystarczy edytować obiekt *Data-Context*, wybrać interesującą nas encję (lub jej właściwość) i zmienić jej właściwość *Source*. Jeśli chcemy zmienić nazwę encji lub jej właściwości, wystarczy użyć opcji *Rename*, dostępnej w Visual Studio. W LINQ to SQL czynności, które zostały wykonane, nie wpływają

w żaden sposób na jakość kodu. W ADO.NET również można wykonać zmianę nazwy tabeli (lub kolumny) relatywnie szybko. Można użyć do tego celu funkcji *QuickRepleace* środowiska Visual Studio, która zamieni wszystkie stare nazwy na nowe. Jednak dla przypadku refaktoryzacji w ADO.NET programista nie ma żadnej pewności, czy operacja została przeprowadzona poprawnie. Dzieje się tak, gdyż mechanizm *QuickRepleace* działa jedynie na tekstach, bez sprawdzania jakichkolwiek zależności w projekcie, co może być powodem niepożądanych zmian w projekcie. Taka zmiana w kodzie niesie za sobą konieczność ponownych testów.

Bardziej rozbudowany schemat bazy danych w cyklu 2 pozwala dostrzec kolejną zaletę technologii LINQ: obsługę relacji jeden-do-wielu. Relacje te są gotowe do użycia po wygenerowaniu klasy *DataContext*. Są one generowane na podstawie wszystkich kluczy obcych w bazie danych. Weźmy pod uwagę relację między tabelami TD_WIADOMOSC i TD_KOMENTARZ. Schemat bazy danych opisuje relacje między tymi tabelami tak, że jedna wiadomość może mieć wiele komentarzy. Według takiej relacji zostały stworzone właściwości w obu klasach. Klasa wiadomości posiada kolekcje komentarzy, natomiast klasa komentarza posiada obiekt wiadomości, której dotyczy. Dodatkowo możemy zmienić nazwy tych właściwości, jeżeli generator nie wygenerował przyjaznych nazw. Wygenerowany kod można użyć następująco:

```
var wiadomosc = Helper.Instance.DataContext.TD_WIADOMOSCs.First();
//Pobieramy wszystkie komentarze wiadomosci
var komentarzeWiadomosci = wiadomosc.TD_KOMENTARZs;

var komentarz = Helper.Instance.DataContext.TD_KOMENTARZs.First();
//Pobieramy wiadomosc komentarza
var wiadomoscKometarza = komentarz.TD_WIADOMOSC;
```

Takie podejście ma wiele zalet. Pierwszą z nich jest fakt, że programista nie musi myśleć o pisaniu zapytań, jeśli pragnie odwołać się do innej tabeli po kluczu obcym. Wyobraźmy sobie tabelę posiadającą kilkanaście kluczy obcych (przykład teoretyczny), dodatkowo w wielu bazach danych takie klucze są nazywane skrótowo. Gdybyśmy chcieli napisać zwykłe zapytanie SQL, to niejednokrotnie takie zadanie jest czasochłonne dla programisty. W LINQ programista nie musi dokładnie pamiętać schematu bazy danych, pisanie logiki biznesowej odbywa się bardziej intuicyjnie.

Kolejną zmianą mającą na celu wyrobienie zdania o każdej z technologii było utworzenie oraz oprogramowanie struktury drzewiastej. Obiektowe podejście do danych w LINQ sprawia, że programista może swobodnie poruszać się po każdej strukturze – także po drzewiastej. Możemy używać rekurencji, która jest oczywiście również możliwa w ADO.NET, jednak zwięzłość kodu w LINQ jest w tym wypadku wielkim atutem. Warto

wspomnieć, że nie jesteśmy wstanie napisać *stricte* rekurencyjnego zapytania. Dopiero użycie procedur i funkcji pozwala na pełną rekurencję.

W drugim cyklu spirali można dostrzec także wady LINQ. Generator nie dostarcza gotowych właściwości odpowiedzialnych za relację wiele-do-wielu. Taka relacja jest mocno schematyczna – da się automatyczne znaleźć tabele, które są w takiej relacji: zazwyczaj są to tabele połączone za pomocą tabeli pomocniczej, składającej się z minimum dwóch kluczy obcych. Pewnym wytłumaczeniem takiego stanu rzeczy jest fakt, że mogą istnieć tabele z wieloma kluczami obcymi, a wtedy istniałoby ryzyko wygenerowania mnóstwa niepotrzebnych właściwości. Również może zdarzyć się relacja, która ma swoje właściwości – tutaj pojawia się problem, z której strony ją wyświetlać. Z drugiej strony z taką relacją wiąże się jeszcze jeden problem. Dla tabeli składającej się tylko z dwóch kluczy obcych generuje się schemat działający niepoprawnie. Przy jakiejkolwiek próbie zmiany danych w tabeli łączącej otrzymujemy wyjątek:

```
Can't perform Create, Update or Delete operations on 'Table(TD_WIADOMOSC_KATEGORIA)' because it has no primary key.
```

Powodem takiego stanu jest brak klucza głównego w tabeli. Są trzy rozwiązania takiego problemu:

- ustawienie klucza głównego ręcznie w DataContext,
- dodanie klucza głównego w tabeli,
- zastosowanie klucza głównego złożonego.

Warto zauważyć, że w ADO.NET nie ma powyższego problemu. Z jednej strony jest to zaletą ADO.NET, ponieważ skoro baza danych pozwala nam stworzyć tabelę bez klucza głównego, więc również możemy modyfikować jej dane. W pierwszej chwili wyjątek rzucony przez LINQ może się wydawać mocno przesadzony, jednak z drugiej strony tworzenie tabeli bez klucza głównego nie jest dobrą praktyką. Taki wyjątek w pierwszych fazach tworzenia aplikacji może zaoszczędzić wiele problemów w przyszłości. Warto jednak zaznaczyć, że brak generacji relacji wiele-do-wielu jest jedynie niedociągnięciem, ponieważ oprogramowanie takiej relacji jest mimo wszystko prostsze w LINQ niż w ADO.NET.

6.3. Wnioski płynące z trzeciego cyklu spirali

Trzeci cykl spirali w większości przypadków potwierdza wnioski z pierwszego oraz drugiego cyklu. W tym cyklu również na jaw wyszły pewne niedociągnięcia LINQ. Pierwszą poważną wadą LINQ jest niepoprawnie zaimplementowana obsługa kolumn, których dane są pojedynczym znakiem oraz nie mogą być puste. Błąd polegał na tym, że gdy nie wypełnimy takiego pola, system bazodanowy nie rzuca żadnego wyjątku – a powinien. W tej sytuacji

nieoceniona okazała się pomoc SQL Profiler. Okazało się, że podczas wstawiania wiersza do tabeli, dla właściwości, której nie wypełnialiśmy, wstawia się znak '\0'. Znak ten został stworzony przez domyślny konstruktor typu *char*. Jest to błąd trudny do wykrycia, w szczególności gdy zakładamy poprawne działanie mechanizmów, z których korzystamy.

Kolejnym mankamentem LINQ jest sposób, w jaki zaimplementowano obsługę operacji typu JOIN. Dla programisty znającego składnię języka SQL i sprawnie posługującego się złączeniami, przejście na podejście zaproponowane w LINQ może stanowić pewien problem. Najszybciej można opanować złączenie INNER JOIN: składnia jest podobna do tej z języka SQL z tą różnicą, że kolejność fraz SELECT, FROM, JOIN oraz WHERE jest nieco inna, a zamiast znaku równości mamy słowo kluczowe EQUALS. Nieco mniej naturalna jest obsługa operacji LEFT OUTER JOIN. Kłopot dla tej operacji polega na tym, że nie zdefiniowano w języku programowania słowa kluczowego LEFT. W LINQ do złączeń tego typu przewidziano metodę *DefaultIfEmpty*() [1]. Przykład kodu, który obsługuje takie złączenie oraz kod SQL, jaki się generuje w takim przypadku, znajduje się poniżej.

```
//Operacje left join w LINQ
var dc = Helper.Instance.DataContext;
var res = from u in dc.TD_UZYTKOWNIKs
    join r in dc.TD_ROLAs on u.TD_ROLA_ID equals r.ID into joined
    from r in joined.DefaultIfEmpty()
    select new { u.LOGIN, r.KOD };

//Zapytanie, jakie zostanie wywołane na bazie danych
SELECT [t0].[LOGIN], [t1].[KOD] AS [KOD]
FROM [dbo].[TD_UZYTKOWNIK] AS [t0]
LEFT OUTER JOIN [dbo].[TD_ROLA] AS [t1] ON [t0].[TD_ROLA_ID] = [t1].[ID]
```

Sprawa komplikuje się jeszcze bardziej w przypadku operacji RIGHT OUTER JOIN. Nie istnieje słowo kluczowe RIGHT, choć istnieje oczywiście możliwość zamiany miejscami zbiorów i wykonania operacji LEFT OUTER JOIN. Może się to okazać kłopotem, gdy taka operacja będzie nam potrzebna (np. tłumaczenie złożonego zapytania SQL na LINQ) [1].

Również niezaimplementowaną funkcjonalnością w LINQ jest złączenie FULL OUTER JOIN. Złączenie takie można napisać za pomocą LINQ, co przedstawia poniższy kod [7] dla naszego schematu bazy danych:

```
//Kod wykonujący złączenie FULL OUTER JOIN.
var dc = Helper.Instance.DataContext;

var left = from u in dc.TD_UZYTKOWNIKs
           from r in dc.TD_ROLAs.Where(r => u.TD_ROLA_ID == r.ID).DefaultIfEmpty()
           select new { u.LOGIN, r.KOD };

var right = from r in dc.TD_ROLA
             from u in dc.TD_UZYTKOWNIKs.Where(u => u.TD_ROLA_ID == r.ID).DefaultIfEmpty()
            select new { u.LOGIN, r.KOD };

var res = left.Concat(right).Distinct();
```

```
//Zapytanie, jakie zostanie wywołane na bazie danych
SELECT DISTINCT [t5].[LOGIN], [t5].[value] AS [KOD]
FROM (
    SELECT [t4].[LOGIN], [t4].[value]
    FROM (
        SELECT [t0].[LOGIN], [t1].[KOD] AS [value]
        FROM [dbo].[TD_UZYTKOWNIK] AS [t0]
        LEFT OUTER JOIN [dbo].[TD_ROLA] AS [t1] ON [t0].[TD_ROLA_ID] = [t1].[ID]
        UNION ALL
        SELECT [t3].[LOGIN] AS [value], [t2].[KOD]
        FROM [dbo].[TD_ROLA] AS [t2]
        LEFT OUTER JOIN [dbo].[TD_UZYTKOWNIK] AS [t3] ON [t3].[TD_ROLA_ID] = [t2].[ID]
        ) AS [t4]
    ) AS [t5]
```

Można wysnuć pewien wniosek co do intencji twórców LINQ. Dostarczyli oni podstawowe operacje złączeń, takie jak INNER oraz LEFT JOIN, metody operujące na zbiorach, takich jak *Distinct*, *Union*, *Contact*, *Except*, *Intersect* oraz metody selekcji danych. Dzięki tym mechanizmom można zaimplementować niemal dowolną operację złączeń. Jednak, jak już zostało wspomniane, problem może się pojawić, gdy chcielibyśmy złączyć większą liczbę tabel: w takim przypadku czytelność kodu napisanego w LINQ może być na niskim poziomie w porównaniu z tym, co oferuje nam składnia SQL. Trzeba zatem przyznać, że LINQ jest niedopracowane pod tym względem. Pewnym panaceum na tę sytuację jest fakt, że za pomocą LINQ to SQL można mapować na obiekty również widoki, co pozwoli napisać trudne złączenia za pomocą SQL, zaś resztę logiki można wykonać zay pomocą LINQ.

6.4. Porównanie wydajności LINQ oraz ADO.NET

Pierwszy test wydajności polegał na pobraniu wszystkich rekordów z tabeli. Test ten został przeprowadzony dla odpowiednio 256, 600 tysięcy oraz 1,5 miliona rekordów. Z obiegowych opinii na temat wydajności technologii ORM wynika, że LINQ powinno być wolniejsze. Testy na pierwszym zbiorze danych ukazały zupełnie inny obraz rzeczy. Pierwsze zapytanie do bazy danych w obu technologiach odbyło się w niemal takim samym czasie. W kolejnych podejściach LINQ było około trzykrotnie szybsze. Ponieważ oczekiwania co do wydajności LINQ były odmienne, przeprowadzono kolejne testy, które dla większej ilości danych wykazały, że ADO.NET jest wydajniejsze o około 8%. Jednak bardzo ważną cechę LINQ to SQL ukazuje pierwszy wynik. Badając, czemu dla mniejszej liczby rekordów otrzymano tak zaskakujące wyniki, okazało się, że powodem był obiekt *DataContext* wspólny dla całej aplikacji. Wystarczyło inicjalizować *DataContext* przed każdym zapytaniem i wyniki zrównały się z tymi, jakie można było zaobserwować dla ADO.NET. Wniosek z tego jest taki, że dla mniejszej ilości danych zadziałał mechanizm pamięci podręcznej (caching), który może mocno poprawić wydajność, ale także powodować niepożądane sytuacje w postaci obsługi nieaktualnych danych.

Tabela 1

Wydajność dla pobierania dużej liczby danych. Dane podane w sekundach

256 000 rekordów								Średnia
LINQ	3,06	0,76	1,07					1,63
ADO.NET	2,92	2,98	3,06					2,99
600 000 rekordów								
LINQ	49,54	9,38	10,15	8,55	9,27	9,71		16,10
ADO.NET	47,65	7,48	8,02	8,69	8,52	8,46		14,80
1 500 000 rekordów								
LINQ	169,05	28,88	27,02	20,21	19,66	23,15	26,52	44,93
ADO.NET	167,50	23,17	18,83	19,04	19,93	24,27	19,03	41,68

Kolejny test sprawdzał, jak mechanizmy zachowują się podczas wstawiania rekordów do bazy. W pętli zostało wprowadzonych po 100 rekordów (w oddzielnych zapytaniach). Wyniki okazały się bardzo czytelne i takie, jak oczekiwano. LINQ było wolniejsze od ADO.NET średnio o 50% (tabela 2).

Tabela 2

Wydajność mechanizmów LINQ oraz ADO.NET dla operacji wstawiania pojedynczego wiersza. Dane podane w sekundach. Wykonano po 100 pomiarów

	ADO.NET	LINQ
Średnia	0,013792	0,018297
Mediana	0,011575	0,015875

Kolejnym testem był sekwencyjny zapis dużej ilości danych po 100 i po 1000 rekordów. Test ten również wykazał przewagę ADO.NET nad LINQ. Przewaga ADO.NET nad LINQ wahała się od kilku do kilkunastu procent (tabela 3).

.Tabela 3

Wydajność mechanizmów LINQ oraz ADO dla operacji sekwencyjnego wstawiania 100 i dla 1000 wierszy. Dane podane w sekundach

Dla 100 wierszy							Średnia
LINQ	0,11	0,12	0,13	0,12	0,05	0,09	0,103
ADO.NET	0,08	0,13	0,07	0,12	0,06	0,10	0,093
Dla 1000 wierszy							
LINQ	4,73	4,95	4,88	4,90	4,84	4,87	4,862
ADO.NET	3,06	3,35	3,96	3,34	3,55	3,41	3,445

Niektóre rozwiązania LINQ poprawiają wydajność - dobrym przykładem jest metoda ANY. Rozwiązanie to jest o tyle ciekawe, że korzysta z funkcji EXIST, która działa tak, że wyszukuje pierwszy rekord spełniający warunek i kończy działanie, gdy taki rekord znajdzie. Jest to rozwiązanie wydajniejsze od kodu, w którym liczymy ilość wierszy i przyrównujemy do zera, ponieważ w takim przypadku niezbędne jest skanowanie całej tablicy z danymi. W projektach tworzonych w ADO.NET bardzo często można spotkać to drugie rozwiązanie.

Testy wydajności ukazały, że mechanizm ADO.NET jest wydajniejszy od LINQ to SQL. ADO.NET wyraźnie prowadzi, jeśli chodzi o wstawianie danych do bazy: różnica wydajności sięga od kilkunastu do pięćdziesięciu procent. Wydawać by się mogło, że taki wynik deklasuje mechanizm LINQ to SQL. Jednak trzeba zaznaczyć, że test dotyczył wstawiania danych, a operacja zajmuje czas rzędu setnych sekund. Jest to czas niezauważalny dla człowieka. Zazwyczaj w aplikacjach za jednym razem następuje wstawienie od kilkunastu do kilkudziesięciu rekordów. Dodatkowo, jeśli wziąć pod uwagę, że aplikacja jest webowa, gdzie przeładowanie kolejnej strony trwa dłużej, to przewaga ADO.NET przestaje mieć jakiekolwiek znaczenie. Z kolei pobieranie danych LINQ w niewielkim stopniu ustępuje ADO.NET – różnica sięga kilku procent.

6.5. Przyszłość ORM

LINQ to SQL jest jednym z przedstawicieli technologii ORM. Oprócz LINQ to SQL, dla platformy .NET najpopularniejsze systemy ORM to Entity Framework oraz NHibernate. Entity Framework został wprowadzony wraz z wersją 3.5 SP1 platformy .NET, jednak zyskał na popularności wraz z wprowadzeniem platformy .NET 4.0. Z kolei NHibernate jest cenione przez zwolenników Open Source. Również nie bez znaczenia jest fakt, że NHibernate jest wspierany przez platformę MONO. Jak na razie przyszłość technologii ORM wydaje się być niezagrożona, można nawet powiedzieć, że aktualnie panuje trend na wykorzystywanie tych technologii. Trudno się dziwić ich popularności, gdy skala wysiłku i czasu, jakie możemy dzięki nim zaoszczędzić w trakcie pisania aplikacji korzystającej z bazy danych, jest duża. Jednak trzeba również zwrócić uwagę na wiele kontrowersji związanych z mapowaniem relacyjno-obiektowym. Często podnoszonym przeciwko rozwiązaniom typu ORM argumentem są kwestie wydajnościowe. W przypadku części konkretnych zapytań SQL może okazać się, że kod wygenerowany automatycznie przez warstwę pośrednią będzie mniej efektywny niż napisany ręcznie przez programistę.

7. Podsumowanie

Wnioski, jakie płyną z niniejszej pracy, potwierdzają w głównej mierze obiegowe opinie i kontrowersje wokół technologii typu ORM. LINQ pozwala skrócić (2-, 3-krotnie) czas potrzebny na napisanie kodu współpracującego z bazą danych, ponieważ: (1) istnieje wsparcie dla relacji, które ułatwia programowanie, (2) nie rozprasza programisty potrzebą pisania kodu zapisującego bądź pobierającego dane. Z drugiej strony, ADO.NET wygrywa przede wszystkim na polu wydajności. Również nie bez znaczenia jest fakt, że w LINQ

podejście do operacji złączeń jest niedopracowane i dla programistów znających SQL może wydawać się nieco dziwne.

BIBLIOGRAFIA

1. Kimmel P.: LINQ Unleashed for C#. 2009.
2. McClure W., Beamer G., Croft J., Little A., Ryan B., Winstanley P., Yack D., Zongkernd J.: Professional ADO.NET 2 Programming with SQL Server 2005. Oracle, and MySQL,: Wiley Publishing, Inc., Indianapolis 2006.
3. Boehm B.: Spiral Model of Software Development and Enhancement., 1986.
4. Paquet J.: Dr. Joey Paquet' Home Page. http://newton.cs.concordia.ca/~paquet/wiki-/index.php/Spiral_model.
5. Testy funkcjonalne. http://pl.wikipedia.org/wiki/Testy_funkcjonalne, Wikipedia, 2011.
6. 101 LINQ Samples, http://msdn.microsoft.com/en-us/vcsharp/aa336746, Microsoft, 2011.
7. Hatch R. D. Full Outer Join : LINQ Extension, http://blogs.geniuscode.net/RyanD-Hatch/p?=116, 2011.

Wpłynęło do Redakcji 6 grudnia 2012 r.

Abstract

LINQ technology is one of the representatives of the object-relational mapping (ORM). Article proposes the thesis that this technology reduces the number of errors in the access to the database and facilitates the creation of software. The study attempted to prove the validity of this thesis by comparing the data access technologies LINQ and ADO.NET on the ground of a Web application. The aim of this study was to write two applications using both approaches, what helped to state the validity of the thesis. This objective was realized with use of Strategy pattern, that allowed using consist code for both LINQ and the ADO.NET.

The conclusions of the study confirms that the LINQ technology greatly increases the efficiency of writing code. The main advantage is the smaller amount of code needed to achieve the same goal. An important feature of LINQ is automatically generated DataContext. In LINQ everything is already prepared: we can immediately write code for business logic, we have support from InteliSense mechanism that primarily helps the programmer to write code,

the operations INSERT, UPDATE, and DELETE are simpler. A useful thing is the lambda expression, that allows you to write a condensed code.

Also interaction with the database using the programming language itself, reduces the number of errors in the access to the database. Work has shown that the only disadvantage of LINQ is being less efficient than ADO.NET. However, we claim that the lower performance is not as expensive price to pay, if one takes many advantages of LINQ technology into account.

Adresy

Michał Świderski: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, michal.swiderski@polsl.pl.
Piotr Bugla: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, piotrbugla@gmail.com.