Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Maksymilian Majer
Nr albumu 3368

Juliusz Zakrzewski
Nr albumu 3132

.Net Data Provider dla ODRA obsługujący zapytania sformułowane w SBQL

Praca Magisterska
Napisana pod kierunkiem
Dr. Piotra Habeli

Warszawa, październik, 2008

Streszczenie

Celem niniejszej pracy jest wykazanie możliwości współpracy systemu obiektowej bazy danych ODRA zrealizowanej w technologii JAVA z platformą .NET w sposób nieróżniący się od trybu interakcji komercyjnych technologii bazodanowych ze środowiskiem .NET (z punktu widzenia programisty). Aby to udowodnić został opracowany i zaimplementowany interfejs programistyczny umożliwiający programistom pracującym w pakiecie technologii Microsoft.NET® na korzystanie z rozwijanego na PJWSTK silnika bazodanowego „ODRA".

Platforma Microsoft .NET® posiada grupę technologii „ADO.NET", która definiuje standardy w łączeniu się i interakcji ze źródłami danych (zarówno z relacyjnymi bazami danych jak i danymi przechowywanymi w innych formatach jak np. pliki CSV, XML itp.) Dzięki tym standardom programiści piszący w takich językach jak C#.NET czy VB.NET mogą w ujednolicony sposób korzystać z heterogenicznych źródeł danych bez ponoszenia dodatkowych kosztów związanych z nauką odmiennych interfejsów czy wprowadzaniem istotnych zmian w istniejącym kodzie (w przypadku migracji na inny typ przechowywania danych lub rozbudowy istniejącego programu o obsługę innych źródeł danych).

Na rynku istnieje wiele produktów mających na celu umożliwienie korzystania z danego źródła danych z poziomu platformy .NET w sposób zgodny z praktykami ADO.NET. Przykładami są tzw. Data Providers dla MySQL, IBM DB2, Microsoft SQL Server czy Oracle.

Głównym założeniem niniejszej pracy było stworzenie Data Providera dla systemu ODRA, który wykorzystywałby te same konstrukcje programistyczne jak Data Providers dla wyżej wymienionych platform bazodanowych. Ze względu na dużą skalę zagadnienia, zrealizowany Data Provider implementuje techniki ADO.NET w stopniu pozwalającym na wykorzystanie go jako Data Access Layer dla prostego serwisu WWW ASP.NET korzystającego z operacji CRUD na bazie danych ODRA.

Spis treści

1 Wstęp

1.1 Kontekst Pracy

Praca dotyczy działu informatyki związanego z aplikacjami opartymi na bazach danych (w tym konkretnym przypadku o organizacji obiektowej) a konkretnie sposobu komunikacji pomiędzy bazą danych a aplikacją w języku wysoko poziomowym zgodnym z platformą Microsoft® .NET.

Na rynku oraz w świecie akademickim zostały zaimplementowane różne modele/typy baz danych oraz języków programowania, które potrzebują mechanizmów do interakcji ze źródłami danych. W przypadku niniejszej pracy problem komunikacji języka programowania z bazą danych ograniczono do implementacji systemu obiektowej bazy danych ODRA, zaimplementowanej na PJWSTK, oraz do języków operujących na platformie .NET, takich jak C#.NET, Visual Basic.NET.

Platforma .NET jest bardzo popularną ostatnio technologią tworzenia aplikacji biznesowych opartych o interakcję z różnymi źródłami danych takimi jak relacyjne/obiektowe bazy danych czy XML. Opracowanie sposobu korzystania z systemu ODRA z poziomu technologii .NET udowadnia, że system bazodanowy opracowany na PJWSTK jest wystarczająco dojrzały, aby współpracować z popularnymi rynkowymi rozwiązaniami.

1.2 Problem, którego rozwiązaniem zajmuje się praca

Podstawowym problemem, który został rozwiązany w ramach pracy to brak możliwości korzystania z systemu bazodanowego ODRA bezpośrednio z platformy Microsoft.NET. System ODRA został zrealizowany w języku i technologii JAVA a co za tym idzie, nie można się do niego bezpośrednio podłączyć z konkurencyjnej rodziny technologii Microsoft.NET.

W dzisiejszym świecie IT, ze względu na dużą heterogeniczność języków programowania oraz modeli i implementacji baz danych, o sukcesie danej technologii przesądzają elastyczność i uniwersalność pozwalające współdziałać z odmiennymi technologiami w

identyczny dla programisty sposób jak z tymi „natywnymi” dla danej platformy. Każda licząca się na rynku baza danych dostarcza biblioteki umożliwiające wykorzystanie jej z poziomu różnych języków programowania/platform aplikacyjnych (przykładowo opensource-owy MySQL posiada sterowniki m.in. do Java, PHP, Microsoft.NET, Python).

W ramach niniejszej pracy położono nacisk nie tylko na samo opracowanie i implementację mechanizmu pozwalającego na komunikację ODRA - .NET ale również na wytworzenie interfejsu bazodanowego maksymalnie zbliżonego do obowiązujących w .NET, a konkretnie w technologii dostępu do źródeł danych ADO.NET, standardów i praktyk.

1.3 Krótkie omówienie stanu sztuki i proponowanego rozwiązania

Problem heterogeniczności źródeł danych oraz języków programowania i konieczności ich współpracy wynika z ciągłej rozbudowy istniejących systemów o moduły dodatkowej funkcjonalności, które często korzystają z innej technologii składowania danych niż rozbudowywany system. Wypracowane rozwiązania problemu integracji odmiennych systemów informatycznych doprowadziły do powstania i upowszechnienia się na rynku kilku czołowych technologii będących uniwersalnym interfejsem pomiędzy źródłem danych a aplikacją:

- JDBC
- ODBC
- OLE DB
- LINQ
- ADO.NET (będące również warstwą pośredniczącą pomiędzy ODBC/OLE DB a platformą .NET);

Wszystkie te technologie sprowadzają się zasadniczo do dwóch podstawowych problemów:

- Wydawania bazie danych poleceń i zapytań w sposób dla niej zrozumiały
- Interpretowania zwróconych z bazy danych wyników i ich konwersja do struktur / bytów programistycznych obecnych w danym języku oprogramowania oraz w drugą stronę: konwersja danych ze środowiska języka oprogramowania do danych zrozumiałych dla źródła danych.

Pierwszy problem najczęściej jest rozwiązywany głównie za pomocą techniki „zanurzania SQL w język programowania", czyli podawania poleceń SQL sterownikowi/interfejsowi jako ciągu znaków bez jakiejkolwiek ingerencji w samo polecenie ani też w składnię języka wysokopoziomowego (tekst polecenia, przechowywany jako zmienna tekstowa, bez żadnej dodatkowej obróbki przekazywany jest bezpośrednio do silnika bazy danych).

Rozwiązaniem drugiego problemu jest zazwyczaj mapowanie wartości z baz danych na odpowiadające im typy języka programowania oraz obudowywanie ich w struktury logicznie kompatybilne z konstrukcjami z bazy danych (przykładowo obiekty DataSet, DataTable, DataColumn z technologii ADO.NET).

Na uwagę zasługuje technologia o odmiennym podejściu – LINQ, która integruje język zapytań w stylu powszechnie znanego SQLa ze składnią języków programowania na platformie .NET. Technologia wyjątkowo nowatorska gdyż pozwala odpytywać w języku zbliżonym do SQL nie tylko obiekty bazy danych, ale także praktycznie wszystkie kolekcje obiektów występujących w samym języku programowania. Ponieważ jest to technologia bardzo młoda, wciąż mało popularna i przede wszystkim narzucająca określoną składnię zapytań (pozwalając jedynie na ich odpowiednią konwersję do zapytań dla konkretnego źródła danych – jak np. LINQ to SQL) nie zdecydowano się na jego wykorzystanie w niniejszej pracy preferując budowę interfejsu zgodnie z technikami powszechnie zaakceptowanej technologii ADO.NET.

W niniejszej pracy problem wydawania poleceń/zapytań rozwiązano w myśl techniki „zanurzonego SQL" a raczej „zanurzonego SBQL", czyli języka zapytań systemu bazodanowego ODRA. Zapytania i polecenia formowane są zupełnie niezależnie od składni języków .NET i zamykane jako ciąg znaków typu System.String. Tak spreparowane zapytanie przekazywane jest bezpośrednio do silnika ODRA.

Problem zwracania wartości z bazy danych ODRA okazał się o tyle skomplikowany, że system ten jest obiektowy i wspiera relatywizm obiektów, czyli możliwość tworzenia wielopoziomowej hierarchii podobiektów. Na potrzeby pracy zwracany rezultat zostaje „spłaszczony" do postaci dwupoziomowej, co nie oznacza jednak niemożliwości odpytywania „w głąb" hierarchii obiektów – funkcjonalność ta jest jak najbardziej wspierana przez odpowiednie sformułowanie zapytania przez programistę.

W celu pobierania rezultatów zapytań i wykorzystywania ich w technologii .NET zostały wykorzystane klasy i interfejsy ADO.NET, co umożliwiło zaimplementowanie interfejsu bazodanowego dla ODRY w sposób nieróżniący się znacząco od standardowych interfejsów dostarczonych przez Microsoft dla baz danych SQL Server czy Access. Fakt ten jest ważny gdyż oznacza, że programista biegle posługujący się technologiami ADO.NET może bez problemu przesiąść się z dotychczasowych rozwiązań bazodanowych na system ODRA.

1.4 Technologie i narzędzia wykorzystane w pracy

Podstawowe technologie wykorzystane w pracy to obiektowy system baz danych ODRA zaimplementowany w środowisku/języku Java w ramach prac badawczo rozwojowych na PJWSTK oraz platforma aplikacyjna Microsoft .NET będąca popularnym narzędziem do tworzenia m.in. bazodanowych aplikacji wspierających biznes (systemu typu CRM, ERP, obiegu dokumentów, portale internetowe itp.).

Wykorzystano platformę Microsoft .NET w wersji 2.0 – konkretnie moduł wykonawczy (maszynę wirtualną) CLR 2.0 i technologię ADO.NET 2.0 oraz system ODRA w ostatniej wersji dostępnej w chwili oddania pracy. Zwrócono na ten fakt uwagę, ponieważ na rynku jest już obecna platforma Microsoft.NET w wersji 3.5 (.Net Framework 3.5), która korzysta jednak z modułu wykonawczego CLR w wersji 2.0 a obecna w niej technologia ADO.NET 3.5 różni się od wersji 2.0 dodatkowymi komponentami (m.in. Entity Framework) nie zaś nowszymi wersjami już istniejących komponentów ADO.NET.

Podstawowym narzędziem, które posłużyło do zaimplementowania interfejsu bazodanowego było zintegrowane środowisko do rozwijania aplikacji (Integrated Development Environment – IDE) Microsoft Visual Studio 2008 będące jedynym dostępnym tego typu produktem na rynku dla aplikacji Microsoft.NET.

Całość rozwiązania została uruchomiona i przetestowana na systemie Windows XP i 2003.

1.5 Rezultaty pracy

Podstawowym rezultatem pracy jest „ODRA Data Provider” dla systemu ODRA czyli interfejs programistyczny pozwalający na interakcję z bazą danych ODRA z poziomu języków programowania dla platformy .NET (m.in. C# i Visual Basic).

Rozwiązanie składa się z 2 modułów:

- Natywnego sterownika ODRA – modułu napisanego w języku C# na wzór interfejsu bazodanowego ODRA dla Javy
- ODRAClient – biblioteki zawierającej implementację klas i interfejsów technologii ADO.NET. Biblioteka stanowi warstwę pośredniczącą pomiędzy sterownikiem a programistą. Dzięki niej programista może korzystać z systemu ODRA w analogiczny sposób do korzystania z innych systemów bazodanowych posiadających interfejs ADO.NET

Dzięki zaimplementowaniu Data Providera programiści operujący w środowisku .NET mogą korzystać z systemu ODRA na równi z innymi systemami baz danych obecnymi na rynku co może spowodować popularyzację ODRY a także dowodzi dojrzałości tej technologii. Pozwala to na zmianę istniejącego źródła danych na system ODRA bez ponoszenia dodatkowych kosztów przepisywania kodu.

Data Provider został zaimplementowany w stopniu pozwalającym na zbudowanie prostej aplikacji Web w technologii ASP.NET lub klienckiej aplikacji dla systemu Windows w technologii Windows Forms wykorzystującej podstawowe operacje CRUD wyrażone w języku SBQL. Pełna implementacja pozwalająca wykorzystać wszystkie możliwości systemu ODRA oraz technologii ADO.NET wykracza poza zakres pracy, której celem jest przede wszystkim zbudowanie solidnego fundamentu pod przyszłą, jeszcze ściślejszą, integrację ODRY z platformą Microsoft .NET.

1.6 Organizacja Pracy

Praca składa się z dwóch części:

- Teoretycznej, zawartej w niniejszym dokumencie – jej celem jest:
  - Omówienie technologii ADO.NET służącej interakcji z heterogenicznymi źródłami danych z poziomu platformy .NET
  - Przedstawienie koncepcji przyjętego rozwiązania oraz problemów, jakie można napotkać integrując ODRĘ z platformą .NET. Część ta powinna być fundamentem dla programistów chcących w przyszłości opracować rozwiązania mające na celu integrację platformy .NET z ODRĄ w szerszym niż obecnie zakresie (przykładowo może to być obsługa transakcji systemu ODRA z poziomu technologii ADO.NET)
- Praktycznej, na którą składa się kod źródłowy Data Providera dla ODRA pod postacią zestawu plików projektowych Visual Studio 2008. Projekt ten może zostać bezpośrednio wykorzystany w aplikacjach Web ASP.NET oraz klienckich Windows Forms w celu wykonywania operacji CRUD na systemie bazodanowym ODRA. Jest też fundamentem do dalszej rozbudowy samego Data Providera o możliwości ADO.NET oraz ODRY niezaimplementowane w obecnej wersji. Sam projekt Visual Studio składa się z dwóch bibliotek:
  - ODRADriver, natywnego sterownika ODRA będącego przede wszystkim przeniesieniem na .NET kodu w Javie służącego do obsługi komunikacji z instancją bazy danych ODRA. Projekt ten zawiera niskopoziomowe mechanizmy konwersji danych w postaci strumieni bajtów z formatu Java na format .NET. Sterownik ten wykorzystuje niskopoziomowe komponenty sieciowe sockets do komunikacji z konkretną instancją bazy danych ODRA. ODRADriver można wykorzystywać także niezależnie od jego wrappera (ODRAClient) w celu niskopoziomowej komunikacji z systemem ODRA, co może być przydatne, jeśli programista zamierza wykorzystywać specyficzne funkcje ODRA niedostępne z poziomu ODRAClient, który implementuje jedynie funkcjonalność zgodną ze standardami ADO.NET.
  - ODRAClient – biblioteki, której głównym celem jest nadanie interakcji z systemem ODRA charakterystyk znanych programistom z interakcji z innymi

źródłami zgodnymi z technologią ADO.NET. Biblioteka ta wykorzystuje w tym celu funkcjonalność dostarczaną przez ODRADriver (jest tzw. „wrapperem”). Projekt zawiera implementację interfejsów ADO.NET niezbędnych do zbudowania tzw. Data Provider, czyli komponentu obsługującego interakcję aplikacji platformy .NET ze źródłem danych.

2 Architektura stosowa i system ODRA

Rozdział ten ma na celu przybliżenie czytelnikowi podstawowe koncepcje, na bazie których powstała niniejsza praca. Zostaną tu omówione architektura stosowa (Stack Based Architecture – SBA) oraz system ODRA (Object Database for Rapid Application development), dla którego autorzy napisali interfejs dostępowy dla platformy .NET Framework.

Podejście stosowe do języków zapytań oraz język SBQL zostały opisane w oparciu o książkę Prof. Kazimierza Subiety *„Teoria i konstrukcja obiektowych języków zapytań”* znajdującej się na pozycji (1) prac cytowanych.

2.1 SBA i SBQL

2.1.1 Co to jest SBA?

Architektura stosowa (inaczej podejście stosowe – Stack Based Approach) jest formalną metodologią odnoszącą się do języków zapytań oraz programowania obiektowych baz danych. W SBA koncepcje języków zapytań zostały zrekonstruowane z punktu widzenia języka programowania. To podejście jest motywowane wiarą twórców, że nie istnieje definitywna granica pomiędzy odpytywaniem (querying) a programowaniem, więc powinna istnieć uniwersalna teoria jednolicie pokrywająca oba te aspekty. SBA oferuje ujednoliconą i uniwersalną koncepcyjnie i semantycznie bazę dla zapytań oraz programów obejmujących zapytania zawierające programistyczne abstrakcje, takie jak procedury, funkcje, klasy, typy, metody, perspektywy, itp. (2)

2.1.2 Abstrakcyjny model składu obiektów w SBA

W SBA wprowadza się następujące modele składu obiektów:

- **M0:** obejmuje dowolnie powiązane hierarchiczne struktury danych
  Definicja:

W modelu M0 skład obiektów jest zdefiniowany jako para <S, R>, gdzie S jest zbiorem obiektów, zaś R jest zbiorem identyfikatorów określanych przez nas jako

„identyfikatory startowe”.

S – Obiekty:

```
<i1, Autor, { <i2, Imię, "Piotr">,
              <i3, Nazwisko, "Jemioła"> ,
              <i17, Napisał, i11> }>
<i4, Autor, { <i5, Imię, "Andrzej">,
              <i6, Nazwisko, "Kołodziej">,
              <i7, Adres, {
                     <i8, Miasto, "Kraków">,
                     <i9, Ulica, "Jasna">,
                     <i10, Nr, "15/3"> }
              <i18, Napisał, i13> ,
              <i19, Napisał, i15> }>

<i11, Książka, { <i12, Tytuł, "Boliwia" >,
              <i13, NapisanaPrzez, i1> }>
<i14, Książka, { <i15, Tytuł,
                     "Kuchnie Świata" >
              <i16, NapisanaPrzez, i4> }>
<i17, Książka, { <i18, Tytuł,
                     "Kuchnia Chińska" >
              <i19, NapisanaPrzez, i4> }>
```

R – Identyfikatory startowe:

$i_1, i_4, i_{11}, i_{13}, i_{15}$

Diagram klas

Tabela 2.1.2.1 Model M0 na przykładzie niewielkiej bazy danych

Powyższa tabela ilustruje skład obiektów w modelu M0 na przykładzie niewielkiej bazy danych.

- **M1:** uzupełnia M0 o pojęcia klasy, dziedziczenia i wielodziedziczenia
- **M2:** uzupełnia M1 o dziedziczenie pomiędzy obiektami i dynamiczne role
- **M3:** uzupełnia model M1 i M2 o pojęcie hermetyzacji (własności publiczne i prywatne)

2.1.3 Stos środowisk ENVS

Konstrukcja semantyki języka opartego o podejście stosowe opiera się na 3 pojęciach – nazywania (naming), zakresu (scoping) i wiązania (binding). Pojęcia te znane są z języków programowania i realizowanych przez stos środowisk.

Stos środowisk (ENVS – *enviromental stack* lub *call stack*) znany jest w informatyce już od dawna. W językach programowania, takich jak Java, C++, Pascal, jest dynamiczną

strukturą w postaci stosu, która przechowuje informacje o aktywnych bytach programistycznych czasu wykonania (obiektach, metodach, procedurach, funkcjach lub podprogramach), które są dostępne w danym punkcie sterowania programu komputerowego. (3) (1)

W językach programowania przyjmuje się, że środowisko programu komputerowego podzielone jest na podśrodowiska (sekcje stosu ENVS), które pojawiają się i znikają wraz z przesuwaniem się punktu sterowania programu.

W językach programowania stos środowisk odpowiedzialny jest za:

- Kontrolowanie nazw zmiennych i wiązanie tych nazw
- Przechowywanie wartości lokalnych zmiennych funkcji, procedur lub metod
- Przechowywanie wartości parametrów aktualnych funkcji i procedur
- Przechowywanie tzw. śladu powrotu, tj. adresu instrukcji, do której ma przejść sterowanie po zakończeniu działania funkcji, procedury lub metody

Stos środowisk w podejściu stosowym do języków zapytań spełniał będzie następujące założenia:

- Będzie zgodny z określonym wcześniej modelem składu obiektów M0 – M3. Wszystkie założenia danego modelu składu są odwzorowane w konstrukcji stosu oraz w jego działaniu. Zgodnie z tymi modelami, stos w jednorodny sposób traktuje dane indywidualne, jak i kolekcje.
- Maksymalny rozmiar stosu nie jest ograniczona od strony koncepcyjnej, a występujące ewentualne ograniczenia implementacyjne nie są istotne dla programisty.
- Stos składa się z sekcji, gdzie każda sekcja przechowuje informację o pewnym środowisku czasu wykonania (np. środowisku wywołania pewnej funkcji). Rozmiar sekcji nie jest ograniczony koncepcyjnie.
- Sekcje najbardziej lokalnych środowisk jest umieszczona na wierzchołku stosu, zaś sekcje wcześniej wywołanych procedur (funkcji, metod) są umieszczone odpowiednio coraz niżej.

- Na dole stosu umieszczone są sekcje globalne, więc stos umożliwia wiązanie do dowolnej nazwy, która może wystąpić w zapytaniu lub innych dozwolonych konstrukcjach.
- Stos w jednakowy sposób traktuje dane trwałe i ulotne (chwilowe, dostępne jedynie w czasie wykonania programu)
- Stos przechowuje informacje o definicjach wprowadzanych w zapytaniach lub programach.
- Stos jest strukturą danych odseparowaną od składu obiektów, a rezultaty zapytań lub programów są składowane na stosie rezultatów (Q_RES).

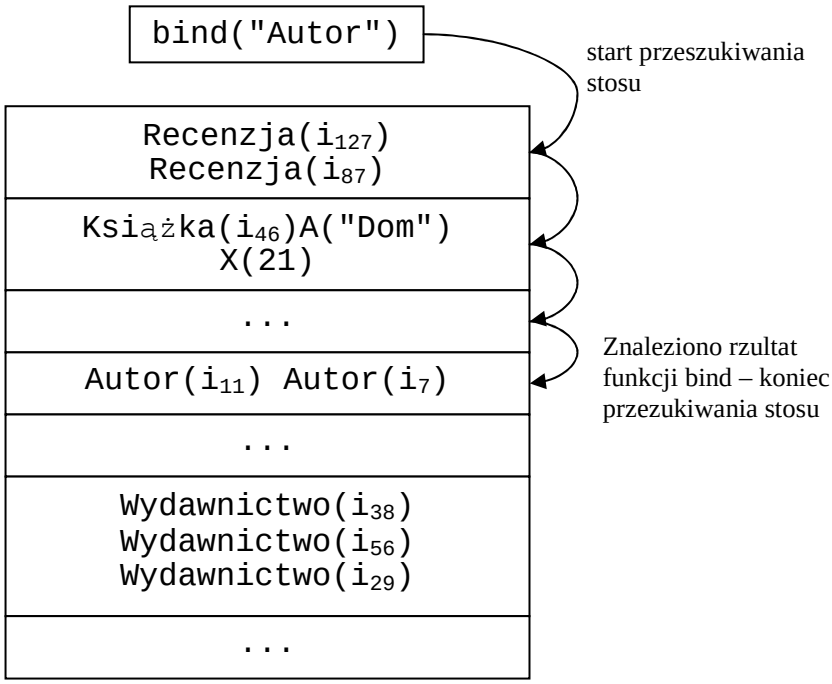
Definicja (1)

Wiązanie (*binding*) nazw użytych w programie jest to zastępowanie nazw występujących w tekście programu przez byty programistyczne czasu wykonania, np. na adresy pamięci operacyjnej, identyfikatory obiektów, adresy startowe procedur itd.

Na stosie wiązania nazw do bytów czasu wykonania przechowywane są w postaci *binderów*. Binder zapisuje się jako *n(x)*, gdzie *n* jest zewnętrzną nazwą zdefiniowaną przez programistę, a *x* dowolnym rezultatem zapytania, w szczególnym wypadku pojedynczą referencją do obiektu

Sekcja stosu to zbiór *binderów* do bytów programistycznych odpowiadających jej środowisku.

Wiązanie nazw przy pomocy stosu środowisk realizuje funkcja *bind(n)*, która zwraca rezultat wiązania nazwy *n*, wcześniej określony jako *x* (czyli *bind(n) = x*). Działanie funkcji bind jest zilustrowane na poniższym rysunku:



Rys. 2.1.3.1 Przykład przeszukiwania stosu podczas wiązania nazwy „Autor”

[Źródło: Opracowanie własne]

2.1.4 Możliwe rezultaty zapytań

W podejściu stosowym rezultaty zapytań mogą być następujące:

- Każda wartość atomową
- Każda referencja do obiektu (czyli jego identyfikator)
- Dowolny *binder* postaci *n(x)*, czyli *nazwaną wartość*
- Strukturę każdego z wymienionych rezultatów, czyli **struct**{ x_1, x_2, x_3, ...}, gdzie **struct** jest flagą określającą rezultat, a x_n dowolnym rezultatem.
- Kolekcję rezultatów, czyli **bag**{ x_1, x_2, x_3, ...} lub **sequence**{ x_1, x_2, x_3, ...} gdzie **bag** i **sequence** są flagami określającymi rezultat, a x_n dowolnym rezultatem.

Flagi **struct**, **bag** i **sequence** nie są *konstruktorami typów*, lecz *konstruktorami wartości*, czyli określają struktury danych zwracane przez zapytania.

Przykładowe rezultaty zapytań to:

- Atomowe
  - true

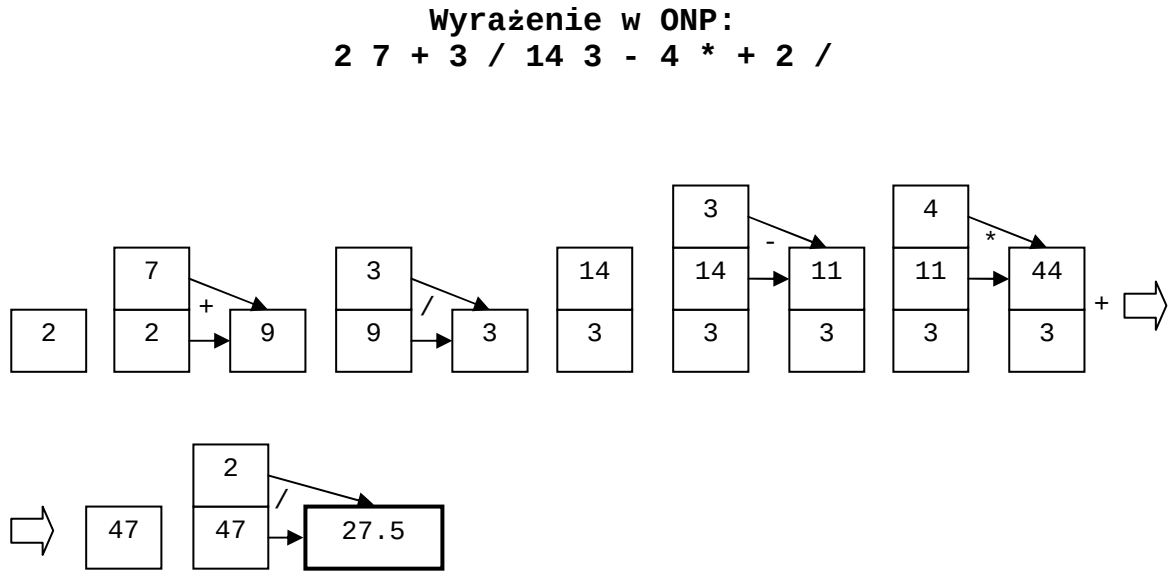
- „Nowak”
- i_{34}
- Pies(i_{19})

- Złożone
  - **struct**{true, „Andrzej”, „Nowak”}
  - **bag**{ **struct**{„Andrzej”, „Nowak”}, **struct**{„Jan”, „Kowalski”} }
  - **bag**{ Autor(**struct**{ fn(„Andrzej”), ln(„Nowak”) }, Autor(**struct**{ fn(„Jan”), ln(„Kowalski”) } }

2.1.5 Stos rezultatów QRES

Jak wspomniano wcześniej stos rezultatów (QRES) jest strukturą odseparowaną od stosu środowisk. Na stos QRES odkładane są wszystkie pośrednie i końcowe rezultaty zapytań.

Przykładowe stany stosu dla wyrażenia ((2+7)/3+(14-3)*4)/2 wyglądają następująco:



Rys. 2.1.5.1 Stany stosu rezultatów dla prostego wyrażenia arytmetycznego [Źródło: Opracowanie własne]

Po wszystkich operacjach arytmetycznych (w tym wypadku) na wierzchołku stosu znajduje się rezultat wyrażenia. Dla języka zapytań stos QRES będzie bazował na tej samej zasadzie, z tym że elementami stosu mogą być wszystkie rezultaty wymienione w rozdziale 2.1.4. Ponadto na stosie rezultatów znajdować będą się elementy pomocnicze do obliczania zapytań (np. liczniki pętli iteracyjnych).

2.1.6 Czym jest SBQL?

SBQL to język zapytań skonstruowany zgodnie z architekturą stosową. Intencją twórców jest, aby SBQL pełnił tę samą rolę, co algebra relacji dla modelu relacyjnego. SBQL jest jednak językiem o nieporównywalnie większej sile. Autorem języka SBQL jest Prof. Kazimierz Subieta. (2)

Podczas konstrukcji języka SBQL jego twórcy założyli, że wyrażenia tego języka będą miały wszystkie cechy języków programowania (np. C#, Java) a dodatkowo posiadać będą operatory umożliwiające pojęciową „hermetyzację" iteracji, których celem jest obsługa kolekcji. Takie dodatkowe operatory to np. selekcja, projekcja, złączenie.

Podstawowe założenia składniowe SBQL (1):

- Literały (wartości atomowe, np. „Piotr", true, 26) są elementarnymi zapytaniami
- Nazwy zmiennych wykorzystywanych w zapytaniach (x, y, z) są elementarnymi zapytaniami
- Dowolne nazwy użyte do konstrukcji schematu bazy danych (Osoba, Nazwisko, Zarobek) są elementarnymi zapytaniami
- Wyróżnia się pewną liczbę operatorów binarnych (+, -, *, where itp.) oraz unarnych (*sum*, *count*, itp.), które dzielą się na algebraiczne (nie zmieniają stosu środowisk) i niealgebraiczne (definicja oparta o odpowiednie zachowanie się stosu środowisk)
- Operatory mogą służyć do łączenia zapytań w większe zapytania np.

```
2 + 2;
Nazwisko = "Jemioła";
2 + (Osoba where Nazwisko = "Jemioła").Zarobek);
```

Tabela 2.1.3.1 Przykład łączenia zapytań w SBQL [Źródło: Opracowanie własne]

- Każde tego rodzaju zapytanie (w szczególności, np. Nazwisko = "Jemioła") będzie posiadało semantykę niezależną od kontekstu.

Zapytania w języku SBQL mogą zatem wyglądać następująco:

```
2 * 10;
"Andrzej " + "Nowak";
```

```
Osoba;
Osoba where Nazwisko = "Nowak”;
(Osoba where Nazwisko = "Nowak”).Wiek;
```

Tabela 2.1.3.2 Przykładowe zapytania [Źródło: Opracowanie własne]

Język SBQL jest też wyposażony w niezbędne wyrażenia imperatywne, takie jak *create*, *delete*, przypisanie (:=), wstawianie (:<).

Przykłady konkretnych zapytań, które zostały wykorzystane do stworzenia aplikacji osadzonej w rzeczywistych realiach (oczywiście bardzo uproszczonej) znajdują się w rozdziale 7 niniejszej pracy.

Dokładny opis operatorów i wyrażeń języka SBQL można znaleźć w podręczniku *„ODRA Description and Programmer Manual”* znajdującym się pod adresem: http://sbql.pl/various/ODRA/ODRA_manual.html (4)

2.2 ODRA

ODRA jest system zarządzania obiektowymi bazami danych napisanym w całości w języku Java. Głównym celem projektu ODRA jest stworzenie nowej idei tworzenia aplikacji bazodanowych. Twórcy chcą osiągnąć ten cel podnosząc poziom abstrakcji, na którym pracują programiści. (4)

Serwer ODRA składa się z dwóch części:

- Procesu komunikacji – obsługuje asynchroniczne połączenia od aplikacji klienckich
- Proces serwera – reprezentuje zarówno stronę serwerową, jak i kliencką systemu ODRA. W niniejszej pracy strona kliencka nie będzie wykorzystywana, ponieważ do komunikacji z serwerem wykorzystana będzie, opisana dalej, własna warstwa kliencka.

Do uruchomienia systemu ODRA niezbędny jest komputer z zainstalowanym środowiskiem JRE (Java Runtime Enviroment) w wersji 1.6+.

3 Istniejące interfejsy dostępowe do baz danych (API)

Niniejszy rozdział wyjaśnia potrzebę stosowania programistycznych interfejsów dostępowych do baz danych oraz omawia interfejsy stosowane w popularnych językach programowania.

3.1 Niezgodność impedancji

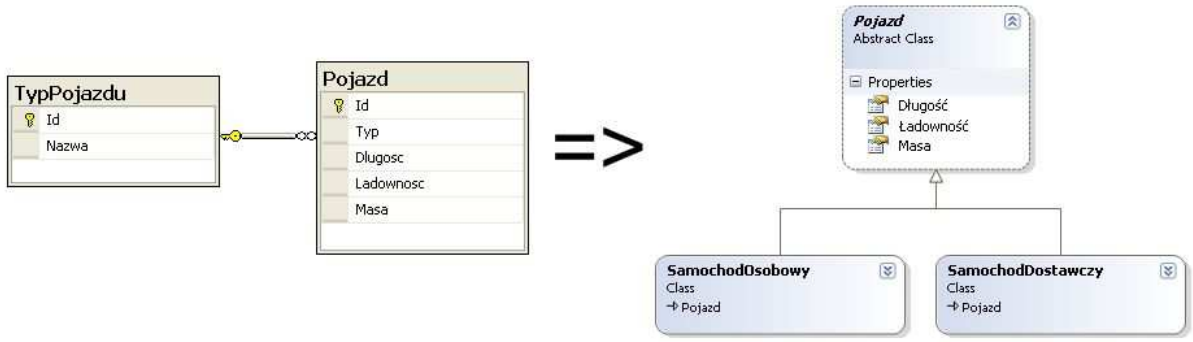
Niezgodność impedancji jest określeniem na zbiór koncepcyjnych i technicznych problemów związanych z używaniem relacyjnych systemów baz danych w programach napisanych w technice obiektowej (Object-Oriented Programming). Problemy te wynikają z zasadniczych różnic pomiędzy podstawowymi założeniami języków zapytań (najczęściej SQL) i uniwersalnych języków programowania.

Niezgodność impedancji objawia się różnicami w zakresie (1):

- **Składni.** Programista musi w jednym tekście programu używać dwóch stylów językowych i przestrzegać reguł dwóch różnych gramatyk.
- **Systemu typów.** Język zapytań operuje na typach zdefiniowanych w schemacie bazy danych, m.in. relacjach, natomiast język programowania posiada zwykle odmienny system typów, w którym nie występuje typ relacja. Większość języków programowania ma wbudowaną statyczną kontrolę typów, podczas gdy SQL takiej kontroli nie przewiduje.
- **Semantyki i paradygmatów języków.** Koncepcja semantyki języków jest zasadniczo różna. Język zapytań bazuje na stylu deklaracyjnym (co wyszukać, a nie jak) podczas gdy języki programowania bazują na stylu imperatywnym (jak wyszukać, skąd pośrednio wynika co).
- **Poziomu abstrakcji.** Język zapytań uwalnia programistę od wielu szczegółów organizacji i implementacji danych (np. organizacji zbiorów, obecności lub nieobecności indeksów, itd.), podczas gdy w języku programowania te szczegóły muszą być oprogramowane explicite.
- **Faz i mechanizmów wiązania.** Języki zapytań są oparte o późne wiązanie (są interpretowane) podczas gdy języki programowania zakładają wczesne wiązanie

(podczas kompilacji i konsolidacji). Stwarza to problemy m.in. dla mocnej kontroli typów, obrazu przestrzeni nazw, itd.

- **Przestrzeni nazw i reguł zakresu.** Język zapytań i język programowania posiadają własne przestrzenie nazw, które mogą zawierać identyczne nazwy o różnych znaczeniach. Odwzorowania pomiędzy przestrzeniami nazw wymagają dodatkowych środków syntaktycznych i semantycznych.
- **Traktowania wartości zerowych.** Bazy danych i języki zapytań posiadają wyspecjalizowane środki dla przechowywania i przetwarzania wartości zerowych. Środki te nie występują w językach programowania.
- **Schematów iteracyjnych.** W języku zapytań iteracje są wtopione w semantykę operatorów takich jak selekcja, projekcja i złączenie. W języku programowania iteracje muszą być organizowane explicite przy pomocy pętli for, while, repeat lub innych.
- **Traktowania cechy trwałości danych.** Języki zapytań przetwarzają wyłącznie trwałe dane (znajdujące się na dysku), podczas gdy języki programowania przetwarzają wyłącznie dane nietrwałe znajdujące się w pamięci operacyjnej. Połączenie obu cech wymaga wprowadzenia specjalnych środków językowych.
- **Środków programowania ogólnego (generic).** Środki te w języku zapytań są oparte o refleksję (np. dynamiczny SQL). Użycie podobnego środka w języku programowania jest trudne z powodu wczesnego wiązania.



Rys. 3.1.1 Przykład mapowania między schematem relacyjnym a obiektowym.
[Źródło: opracowanie własne]

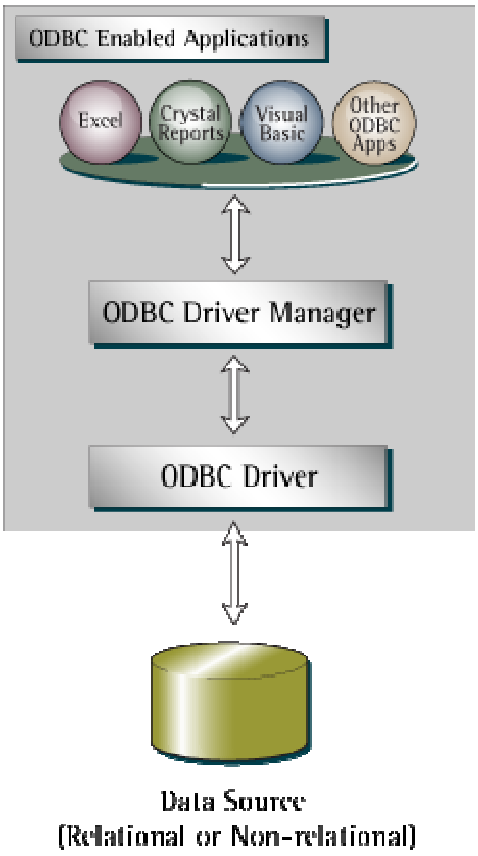
Powyższe niezgodności oraz popularyzacja języków zapytań (w szczególności SQL) wymusiły stworzenie odrębnej warstwy oprogramowania odpowiedzialnej za połączenie tych

dwóch odmiennych koncepcji w jednym programie. Poniżej omówione zostaną najbardziej popularne interfejsy dostępu do baz danych, które stanowią właśnie tę łączącą warstwę.

3.2 ODBC (Open Database Conectivity)

ODBC jest otwartym standardem API zbudowanym w 1992 roku przez Microsoft we współpracy z Simba Technologies w oparciu o specyfikację CLI (Call Level Interface), którą przygotowała SQL Access Group (SAG).

Poniższy rysunek przedstawia architekturę aplikacji wykorzystującej ODBC jako interfejs dostępu do bazy danych:

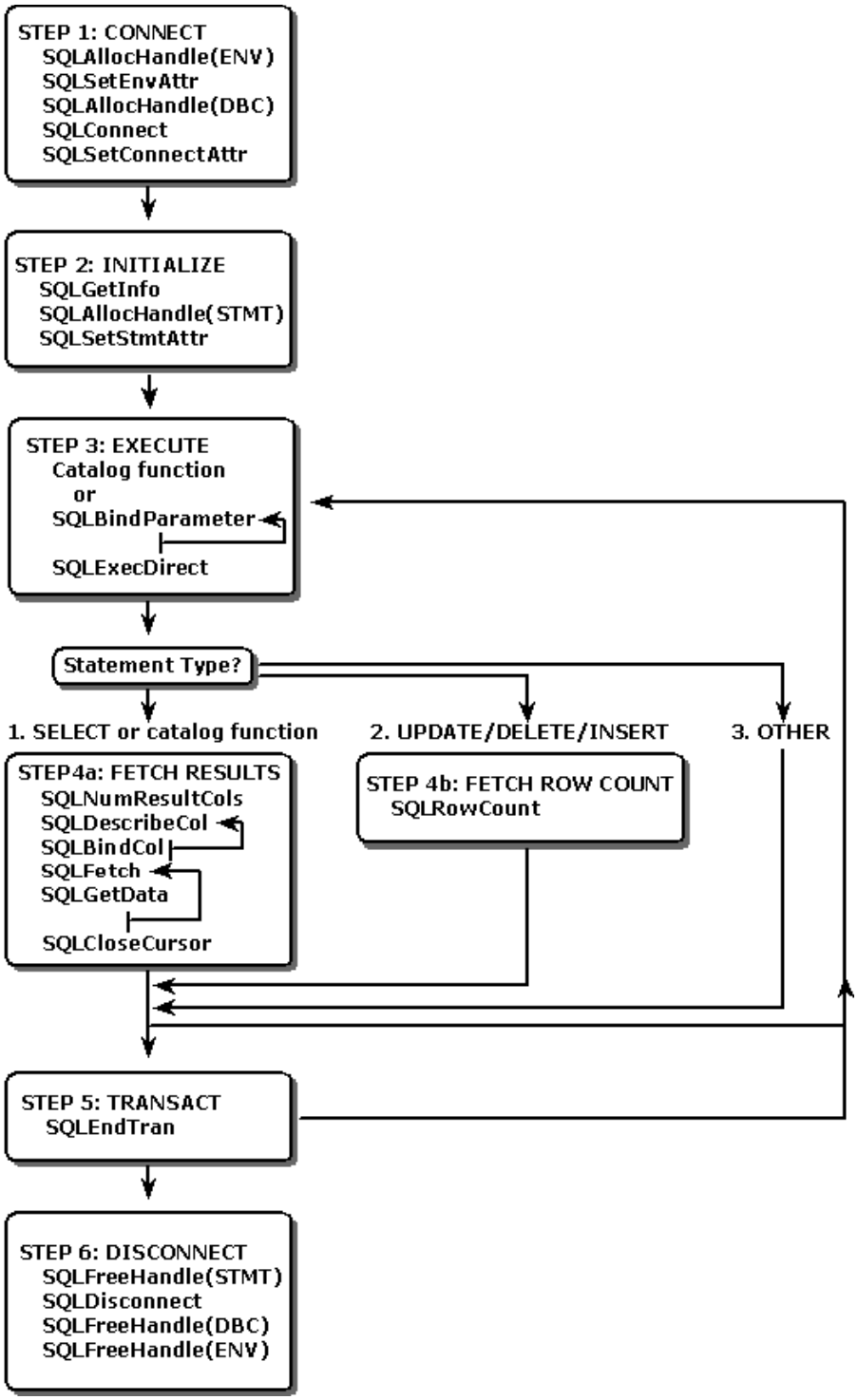


Rys. 3.2.1 Architektura wykorzystująca ODBC [Źródło: SIMBA Technologies]

Komunikacja z bazą danych za pomocą interfejsu ODBC w uproszczeniu odbywa się w poniższych krokach:

- Aplikacja wykorzystująca interfejs ODBC wysyła wyrażenia SQL do ODBC Driver Manager'a i w odpowiedzi otrzymuje rezultaty zapytań.
- ODBC Driver Manager jest warstwą, która odwołuje się do konkretnych sterowników ODBC (ODBC Drivers) na rzecz aplikacji bazodanowej.
- ODBC Driver (sterownik ODBC) przetwarza wołania ODBC i przekazuje zapytania do konkretnego systemu zarządzania bazami danych a rezultaty zapytań przekazuje do aplikacji.

Szczegółowo powyższy schemat działania aplikacji wykorzystującej ODBC jest przedstawiony na rysunku 3.2.2. Z tego schematu wynika, jakie operacje musi implementować sterownik ODBC, aby zapewnić dostęp do bazy danych.



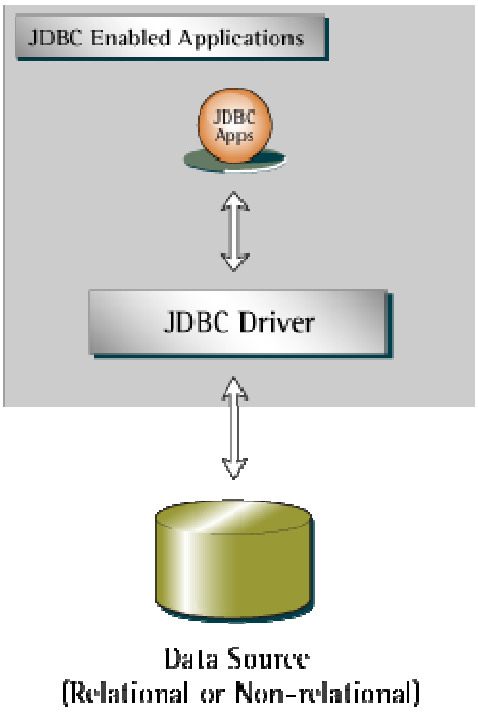
Rys. 3.2.2 Schemat działania aplikacji ODBC [Źródło: MSDN]

3.3 JDBC (Java Database Connectivity)

JDBC jest interfejsem programistycznym (API), który umożliwia uniwersalny dostęp do baz danych programistom języka Java. JDBC pozwala na zadawanie zapytań do silników

zarządzania bazami danych (SZBD lub DBMS – Database Management System) w języku dla nich specyficznym (zwykle SQL). DBMS zwraca rezultaty zapytania poprzez podobny interfejs. JDBC, zarówno jak ODBC, pozwala jednemu programowi na dostęp do wielu źródeł danych znajdujących się w różnych miejscach sieci.

Poniższy rysunek przedstawia architekturę aplikacji wykorzystującej JDBC jako interfejs dostępu do bazy danych.



Rys. 3.3.1 Architektura wykorzystująca JDBC [Źródło: SIMBA Technologies]

JDBC Drivers (sterowniki JDBC) to implementacje interfejsu JDBC pozwalające na używanie konkretnego SZBD (np. MySQL, PostgreSQL). JDBC Driver udostępnia połączenie z bazą danych oraz implementuje protokół transferu rezultatów zapytań między serwerem a klientem.

Dostawcy, którzy chcą, aby programiści Java mogli korzystać z ich DBMS zwykle tworzą własne implementacje sterowników JDBC.

Sterowniki w technologii JDBC mieszczą się w jednej z czterech kategorii:

- **Typ 1: JDBC-ODBC bridge.**
  Sterownik typu JDBC-ODBC bridge (most) wykorzystuje, odpowiednie dla danego DBMS, sterowniki ODBC jako medium w komunikacji z DBMS. Most konwertuje wołania metod JDBC na wołania funkcji ODBC. Tego typu sterowniki wykorzystuje się zwykle, gdy nie istnieje żaden sterownik napisany w czystej Javie dla danego DBMS.
  Taki sterownik jest przywiązany do danego systemu operacyjnego (platform dependent), ponieważ opiera się o sterownik ODBC, który pod ten system jest napisany.
- **Typ 2: Native-API Driver specification**
  Sterowniki drugiego typu wykorzystują biblioteki klienckie systemu DBMS (np. Oracle, DB2, MySQL) do połączenia z bazą danych. Taki sterownik tłumaczy wołania JDBC na natywne wołania DBMS. W tym jest podobny do sterownika typu 1 i wymaga, aby biblioteki klienckie były dostępne na tej samej maszynie, co JVM.
- **Typ 3: Network-Protocol Driver**
  Te sterowniki wykorzystują pośrednią warstwę między programem wołającym, a DBMS. Tą warstwą pośrednią może być serwer aplikacji, który tłumaczy wołania JDBC na wołania odpowiednie dla danej bazy danych. Tego typu sterowniki są napisane całkowicie w Javie i mogą być wykorzystywane do połączenia z różnymi systemami bazodanowymi. Niestety wymagana jest warstwa pośrednia, która musi zajmować się wszystkimi aspektami związanymi z dostępem do konkretnych systemów DBMS działających na różnych platformach.
- **Typ 4: Native-Protocol Driver**
  Ostatni typ sterownika jest napisany w czystej Javie i nie wymaga pośredniego oprogramowania do dostępu do bazy danych. Native-Protocol Driver łączy się z system zarządzania bazami danych bezpośrednio poprzez protokół obsługiwany przez ten system. Tego typu sterowniki są pisane dla konkretnego systemu DBMS.

Każdy sterownik JDBC powinien być zgodny ze specyfikacją JDBC API (obecnie w wersji 4.0), a zatem implementować kluczowe interfejsy JDBC API. Poniżej jest lista interfejsów, które zaimplementować powinien każdy twórca sterownika JDBC:

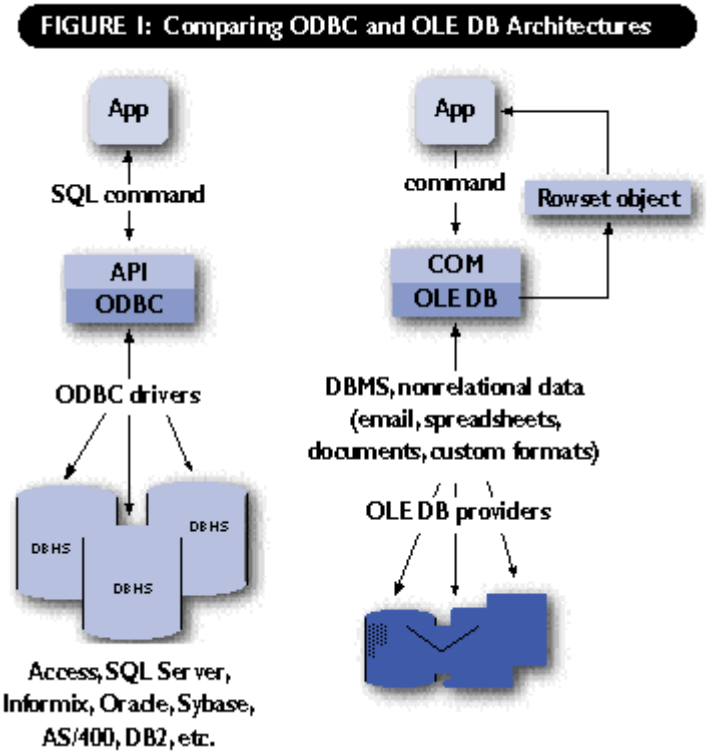
Poniższe interfejsy muszą być zaimplementowane w całości:
`java.sql.Driver`
`java.sql.DatabaseMetaData`
`java.sql.ParameterMetaData`
`java.sql.ResultSetMetaData`
`java.sql.Wrapper`
`javax.sql.DataSource`
Poniższe interfejsy muszą być zaimplementowane, ale niektóre metody mogą być pominięte:
`java.sql.CallableStatement`
`java.sql.Connection`
`java.sql.PreparedStatement`
`java.sql.ResultSet`
`java.sql.Statement`

Tabela 3.3.1 Interfejsy do zaimplementowania przez JDBC Driver [Źródło: JDBC™ 4.0 Specification]

3.4 OLE DB (Object Linking and Embedding, Database)

OLE DB jest kolejną propozycją interfejsu dostępu do danych firmy Microsoft. OLE DB to zestaw interfejsów bazujących na modelu COM (Component Object Model) i został zaprojektowany, aby zastąpić ODBC jako jego następca. OLE DB ma rozszerzać swojego przodka o możliwość współpracy z gronem źródeł danych opartych o nierelacyjny model – takich jak obiektowe bazy danych, arkusze kalkulacyjne, itp.

OLE DB jest podzielony koncepcyjnie na konsumentów (consumers) i dostawców (providers), co przedstawia poniższy rysunek porównujący interfejsy ODBC i OLE DB.



Rys. 3.4.1 Porównanie architektury ODBC i OLE DB [Źródło: SQL Server Magazine]

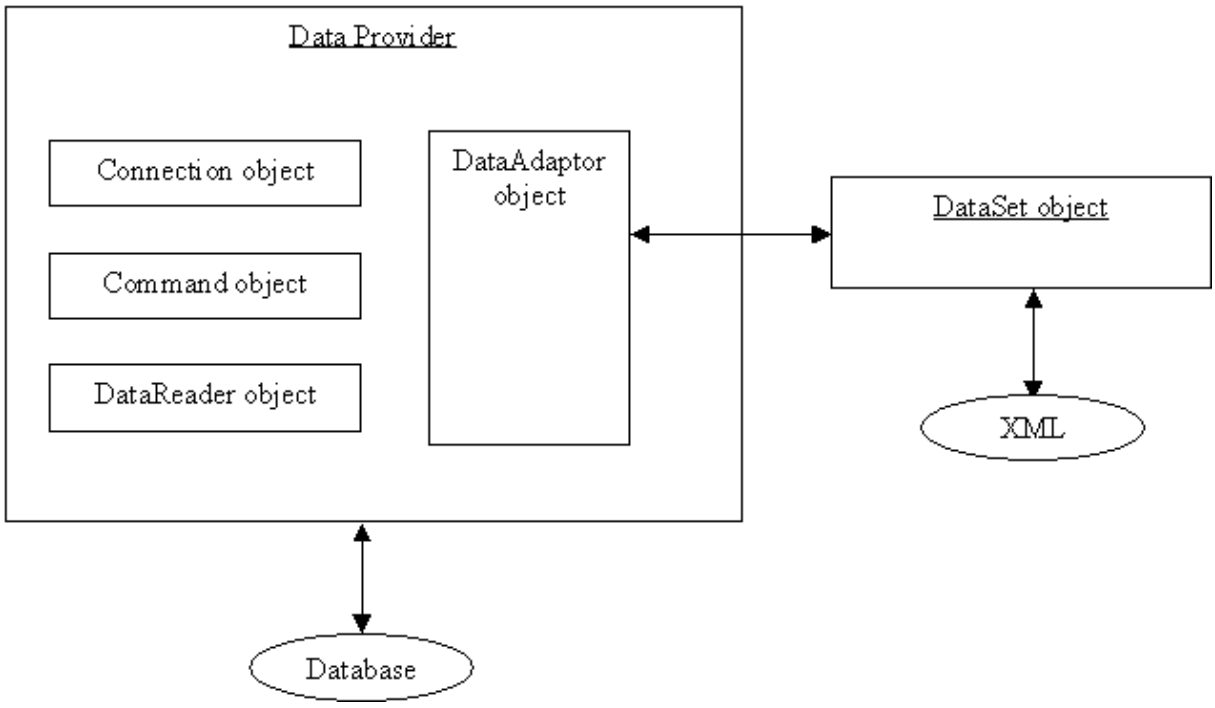
Mimo, że OLE DB zostało zaprojektowane aby wyprzeć ODBC według Simba Technologies ODBC i JDBC są nadal najpowszechniej używanymi interfejsami dostępu do baz danych.

3.5 ADO.NET

ADO.NET jest zestawem komponentów programistycznych wykorzystywanych przez programistów środowiska .NET Framework do dostępu do baz danych. Koncepcyjnie dzieli się na dwie części:

- DataProvider – implementacja interfejsów dostępu do danych. W ADO.NET zaimplementowani zostali dostawcy (providers) do systemów zarządzania bazami danych SQL Server, Oracle oraz dostawca dla OLE DB
- DataSet – klasy odwzorowujące fragment relacyjnej bazy danych i ułatwiające dostęp do danych w niej zawartych.

Poniższy rysunek ilustruje koncepcyjny podział ADO.NET



Rys. 3.5.1 Architektura ADO.NET [Źródło: CodeGuru]

Efektem niniejszej pracy jest implementacja interfejsów DataProvider'a w celu uzyskania dostępu do systemu ODRA i dlatego ADO.NET będzie szerzej omówione w rozdziale 4.5.

3.6 LINQ (Language Integrated Query) a niezgodność impedancji

Microsoft w najnowszej wersji środowiska .NET Framework 3.5 wprowadził możliwość zanurzania zapytań do składów danych wprost w programach napisanych w języku typowym dla .NET Framework (C#, VB, C++). Nie jest to podejście innowacyjne (ODMG zaproponowało wiązania OQL do C++, SmallTalk i Javy), ale na tyle wygodne w użyciu, że w znacznym tempie zdobywa popularność.

LINQ nie eliminuje problemu niezgodności impedancji, ale jednak zmniejsza jego efekt na tyle, aby umożliwić programistom wygodne zadawanie zapytań z poziomu ich ulubionego języka programowania. Poniżej przedstawiono zakres w jakim LINQ zmniejsza

niedogodności spowodowane niezgodnością impedancji. Zmniejszenie niedogodności można odczuć w zakresie[1]:

- **Składni.** Twórcy LINQ spróbowali połączyć cechy składni języka SQL ze składnią typową dla języków programowania. Dużym udogodnieniem zanurzenia zapytań LINQ (tzw. wyrażeń-λ) jest podpowiadanie dopełnień kodu (IntelliSense) w trakcie pisania – przyspiesza pracę. Jest to cecha Visual Studio IDE, ale pomaga oswoić się z nowym językiem poprzez zachowanie typowe dla pozostałego kodu.
  Poniżej zilustrowane są dwa sposoby na zadanie zapytania w LINQ:

```
IOrderedQueryable<ext_Article> articles =
                                from a in db.ext_Articles
                                     orderby a.date_added
                                     select a;
List<ext_Article> linqArticles =
            articles.Where(art => art.title.Contains("LINQ")).ToList();
```

Tabela 3.6.1 Zapytanie prezentujące składnię LINQ

- **Systemu typów.** Dla celów zadawania zapytań w LINQ generuje się typy odpowiadające typom (tabelom) zdefiniowanym w bazie danych, dzięki czemu na etapie kompilacji można wykryć niektóre błędy typologiczne.
  Poniższy kod prezentuje podobny przykład do tego z tabeli 3.6.1, ale kompilator wyświetli błąd, ponieważ typ oczekiwany (kolekcja obiektów `ext_ArticleType`) przez programistę nie zostanie zwrócony przez zapytanie:

```
IOrderedQueryable<ext_ArticleType> articles =
                                from a in db.ext_Articles
                                     orderby a.date_added
                                     select a;//błąd kompilatora
```

Tabela 3.6.2 Zapytanie prezentujące kontrolę typów w czasie kompilacji

[1] Są to wnioski autorów pracy wyciągnięte na bazie doświadczenia w używaniu LINQ i innych interfejsów dostępu do baz danych. Niedogodności wybrane na podstawie (1)

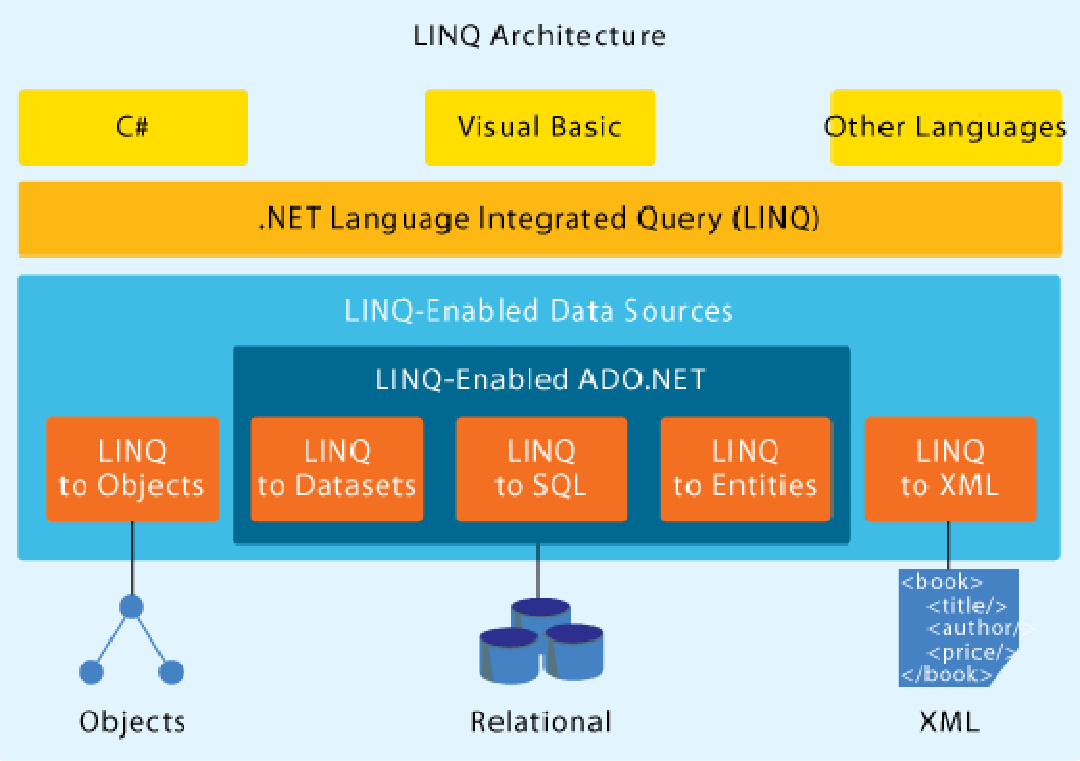
- **Schematów iteracyjnych.** LINQ udostępnia programistom operatory, w których iteracje są wtopione semantycznie. Te operatory umożliwiają dokonanie selekcji, projekcji i złączania na kolekcjach obiektów ukrywając przed programistą szczegóły implementacji iteracji. Oczywiście tych samych operacji można dokonywać na składzie danych z różnych źródeł (np. relacyjna baza danych, xml lub ulotna kolekcja obiektów).

```
List<int> no = new List<int>();
        no.Add(1);
        no.Add(2);
        no.Add(3);
        no.Add(4);
        int sum = no.Sum();
```

Tabela 3.6.3 Zapytanie prezentujące składnię LINQ

Powyższa lista udogodnień pokazuje, że próba rozwiązania problemu niezgodności impedancji, również w przypadku LINQ, nie jest wysokich lotów, jednak prawdopodobnie nie to było intencją jego twórców. Zatem pomimo wygodniejszego użycia LINQ nie wnosi rewolucyjnych zmian ideowych w interfejsach dostępów do danych pozostając jedynie kolejną, atrakcyjną warstwą pośredniczącą między językiem programowania a systemem zarządzania bazą danych.

Poniższy rysunek prezentuje architekturę aplikacji używających LINQ:



Rys. 3.6.1 Architektura LINQ [Źródło: The Code Project]

LINQ jest nadal środkiem pośrednim, niekompletnym językiem programowania, który wymaga warstwy dodatkowych komponentów między programem napisanym w języku takim, jak C# lub VB a DBMS. Ta warstwa jest dość obszerna (diagramy mapujące obiekty relacyjne a obiekty języków programowania, biblioteki LINQ, które tłumaczą zapytania na wyrażenia lub operacje odpowiednie dla używanego typu składu danych), ale mimo to siła jego promocji i mocne zaplecze powodują, że staje się coraz bardziej powszechny nawet wśród zwolenników technologii niezwiązanych z Microsoftem[2].

3.7 JOBC (Java Object Base Connectivity)

ODRA JOBC jest interfejsem dostępowym do systemu ODRA wzorowanym na JDBC. Umożliwia on wykorzystanie baz danych systemu ODRA w aplikacjach JAVA. JOBC stanowi zatem podobny interfejs do tego, który jest efektem niniejszej pracy magisterskiej.

[2] Patrz wywiad *LINQ is the best option for a future Java query API*, na http://www.odbms.org/blog/

Zasadniczą różnicą pomiędzy JOBC, a np. JDBC, czy implementacją providera ADO.NET przedstawioną w niniejszej pracy jest to, że rezultaty wołań do systemu ODRA zwracane są w postaci wewnętrznych typów danych, takich jak:

```
BagResult
BinderResult
StructResult
StringResult
Itp.
```

Tabela 3.7.1 Przykładowe typy rezultatów zwracanych przez ODRA JOBC

Jest kilka wad takiego rozwiązania:

- Zwrócone rezultaty są nieintuicyjne dla użytkowników korzystających wcześniej z typowych sterowników JDBC, co utrudnia pracę i wymaga nabycia nowych przyzwyczajeń
- Rezultaty należy poddać dodatkowej obróbce zanim dane wyświetli się końcowemu użytkownikowi. Wymaga to większego nakładu pracy – napisania procedur do wyciągania danych istotnych dla użytkownika.

Wad tych niestety trudno uniknąć zachowując jednocześnie funkcjonalność, którą udostępnia system ODRA i język SBQL, ponieważ wynika to z zupełnie innego modelu składu danych w porównaniu z istniejącymi w relacyjnych systemach zarządzania bazami danych.

4 Platforma .NET i technologia dostępu do danych ADO.NET

4.1 Podstawowe cechy platformy Microsoft.NET

Platforma Microsoft.NET jest zbiorem technologii tworzących jednolitą, uniwersalną platformę do budowania aplikacji, bez jakichkolwiek ograniczeń dziedzinowych, działających w oparciu o system operacyjny Microsoft Windows. Podstawową przyczyną opracowania platformy .NET było zapotrzebowanie na technologię wytwarzania oprogramowania dla Windows umożliwiającą pisanie wysokopoziomowych aplikacji biznesowych w stopniu równie prostym, jak pisanie aplikacji w technologii/języku Java. Przez prostotę należy w tym przypadku rozumieć wyręczenie programisty z pisania kodu infrastrukturalnego dostarczając gotowe łatwo konfigurowalne mechanizmy, które okazują się uniwersalną częścią większości wysokopoziomowych aplikacji biznesowych. Do tego typu mechanizmów należą m.in.:

- Zarządzanie pamięcią i ogólniej cyklem życia poszczególnych obiektów, usług aplikacyjnych i samych aplikacji klienckich lub serwerowych.
- Zarządzanie bezpieczeństwem wykonywanego w danym kontekście programu na podstawie tożsamości użytkownika, pochodzenia i cech kodu. Wprowadzono również mechanizm definiowania odpowiednich zachowań i uruchamiania oddzielnego, wirtualnego systemu plików w przypadku aplikacji pobieranych z sieci Internet, co jest ważną funkcjonalnością ze względu na rosnącą liczbę aplikacji publikowanych za pośrednictwem Internetu.
- Tworzenie graficznych interfejsów użytkownika (GUI) i jednolitą obsługę związanych z nimi mechanizmów wyświetlania danych i elementów interaktywnych; obsługę zdarzeń i komunikacji między aplikacją a systemem operacyjnym
- Zapewnienie ujednoliconego modelu dostępu do heterogenicznych źródeł danych jak i umożliwienie definiowania w zestandaryzowany metod wyświetlania i interakcji z danymi (tzw. „Data Binding” czyli zespół klas, interfejsów, kontrolek wizualnych, dostarczonych zarówno dla aplikacji klienckich Windows Forms jak i dynamicznych stron internetowych zbudowanych w oparciu o ASP.NET, pozwalających w sposób deklaratywny konfigurować połączenia interfejsu użytkownika z danymi)

- Zarządzanie w sposób ujednolicony komunikacją pomiędzy aplikacjami klienckimi i sieciowymi wspierając przy tym dostępne na rynku standardy takie jak oparte na XML i SOAP Web Services oraz bazujące na protokole HTTP mechanizmy REST i AJAX.
- Zarządzanie wersjonowaniem i dystrybucją aplikacji oraz ich komponentów (plików .dll) a także możliwość instalacji oprogramowania i jego ustawień konfiguracyjnych bez potrzeby wykorzystywania systemowego rejestru Windows, co okazało się problematyczne w przypadku częstego uaktualniania istniejących aplikacji.

Główną ideą przyświecającą projektantom platformy .NET było stworzenie technologii wytwarzania aplikacji dla systemu Windows, dzięki której programista przystępując do pisania programu może natychmiastowo zająć się dziedziną problemową swojego klienta i pracą nad rozwiązaniem problemu biznesowego. W przeszłości pierwsza faza budowy aplikacji dla Windows koncentrowała wysiłki na pisaniu kodu narzędziowego służącego umożliwieniu działania aplikacji.

4.2 Języki programowania i model uruchamiania aplikacji w .NET

Ukierunkowanie na programistę i chęć stworzenie dla niego wygodnego środowiska pracy odzwierciedlone jest również w przyjętym modelu języków programowania. Oprogramowanie dla platformy Microsoft.NET może być tworzone w wielu obecnych na rynku i w świecie akademickim językach pod warunkiem opracowania ich wersji wspierającej standard Common Language Specification (Wspólna Specyfikacja Języka) czyli zestandaryzowany przez organizację ECMA zbiór reguł, dzięki którym program napisany w danym języku i odpowiednio skompilowany może zostać uruchomiony na platformie .NET i korzystać z wszystkich wymienionych wcześniej mechanizmów.

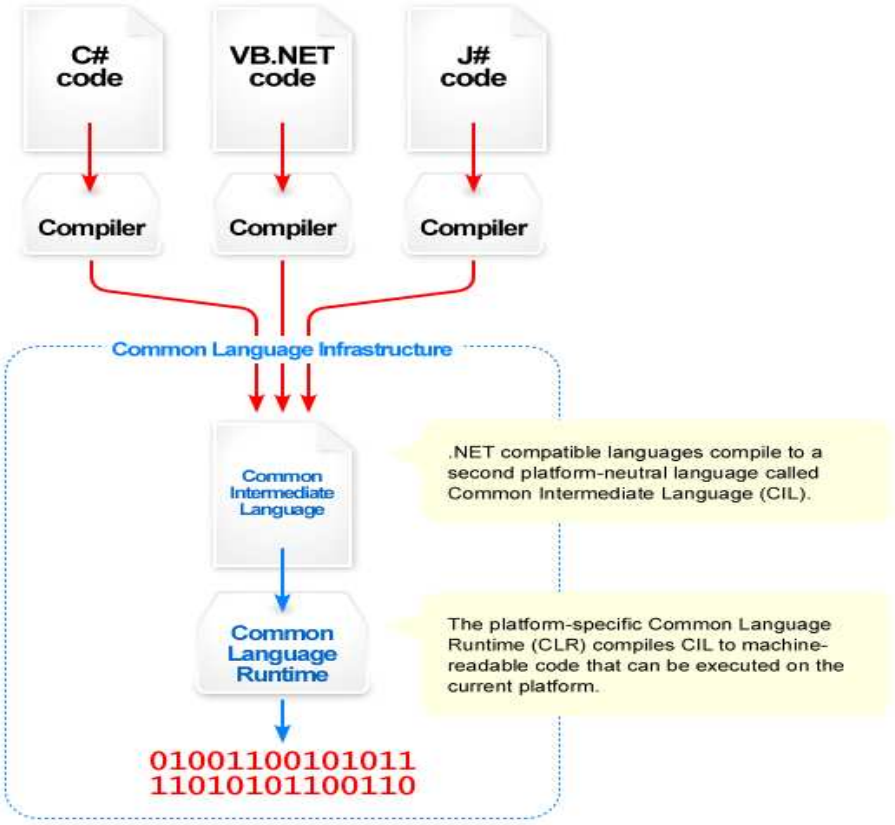
W celu umożliwienia operowania wieloma językami, Microsoft opracował jeszcze szerszy standard, zatwierdzony przez organizację standaryzującą ECMA pod numerem ECMA-335 oraz przez instytut ISO pod numerem ISO/IEC 23271) noszący nazwę Common Language Infrastructure (Wspólna Infrastruktura Językowa), na który składają się następujące standardy/komponenty:

- Common Type System (CTS, Wspólny System Typów). Standard ten określa zbiór typów dostępnych na platformie Microsoft.NET i sposób ich reprezentacji w pamięci komputera. Jest to wyjątkowo ważny element gdyż pozwala odmiennym językom na

dzielenie się danymi. Definiuje też między innymi sposoby budowania typów obiektowych i przekazywania obiektów do metod.

- Common Language Specification – wymieniony wcześniej standard określający reguły budowy języków kompatybilnych z Microsoft.NET. Określa m.in. takie charakterystyki języka jak reguły zakresu („scoping”), nazewnictwa typów i metod, modelu dziedziczenia, przeciążania i przesłaniania typów i metod itp.
- Metadane – sposób opisywania typów tak, aby typ obiektowy napisany w danym języku był zrozumiały dla całej platformy .NET
- Virtual Execution System (Wirtualny System Uruchamiania) – standard definiujący sposób uruchamiania aplikacji skompilowanej do kodu pośredniego a nie kodu odpowiadającego rozkazom maszynowym danego procesora.

Zarys architektury Common Language Infrastructure przedstawiony jest diagramie 1



Rys. 4.2.1 Common Language Infrastucture [Źródło: www.wikipedia.org]

4.3 Kod Zarządzany

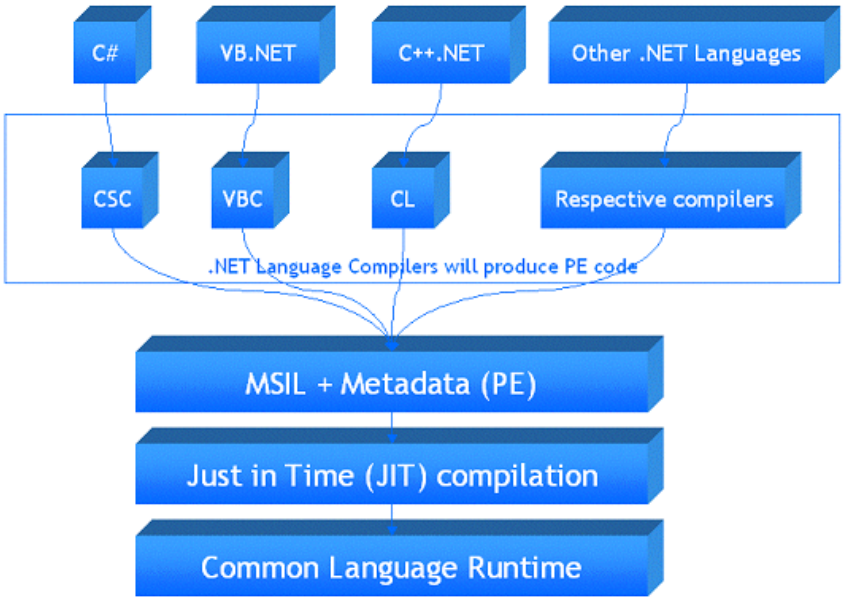
Bardzo ważnym elementem platformy Microsoft .NET jest koncepcja kodu zarządzanego (ang. managed code). Pod pojęciem tym należy rozumieć kod programu napisanego w jednym z języków kompatybilnych z platformą .NET, który nie zostanie samodzielnie wykonany w systemie lecz, aby móc spełniać swoje funkcje, korzysta z mechanizmów opisanych w sekcji 4.1. Aby umożliwić taki tryb wykonania z technicznego punktu widzenia, kod programu języka .NET nie jest bezpośrednio tłumaczony na rozkazy procesora i uruchamiany, lecz tłumaczony jest na specjalny język pośredni tzw. Common Intermediate Language (CIL) czyli Wspólny Język Pośredni. Został on zaimplementowany jako Microsoft Intermediate Language (MSIL) i jest odpowiednikiem obecnego na platformie Java – „Java bytecode". Kod pośredni tworzony jest przez kompilator odpowiedniego języka i uruchamiany przez Common Language Runtime, czyli moduł wykonawczy wspólny dla wszystkich języków .NET. Moduł ten, będący wcieleniem Virtual Execution System, ładuje kod pośredni do pamięci i uruchamia tłumacząc na kod maszynowy za pomocą mechanizmu JIT i zapewniając współdziałanie wszelkich mechanizmów .NET takich jak zarządzanie pamięcią i bezpieczeństwem.

Wspomniany mechanizm JIT (Just In Time compilation – kompilacja w ostatniej chwili) jest bezpośrednio odpowiedzialny za odpowiednie tłumaczenie MSIL do kodu maszynowego. Jest również ważnym elementem z uwagi na optymalizację szybkości wykonania programu, dzięki czemu kod wykonywany przez CLR jest szybszy od np. języków interpretowanych. JIT zawiera też mechanizmy optymalizujące kod na podstawie cech konkretnego systemu jak wersja procesora (przykładowo zawierająca dodatkowe zestawy rozkazów typu SSE) czy systemu operacyjnego, co czyni program szybszym aniżeli w przypadku kompilacji statycznej tworzącej kod generyczny dla danej rodziny procesorów.

Microsoft .NET jest także wyposażony w narzędzie Native Image Generator (ngen.exe), które można wykorzystać do wygenerowania programu w postaci kodu maszynowego (co ma miejsce na przykład w procesie instalacyjnym IDE Microsoft Visual Studio 2005 i 2008) Dzięki temu otrzymane pliki wykonywalne są równie szybkie jak programy bez kodu pośredniego a nawet szybsze, ponieważ kod optymalizowany jest pod konkretny procesor i system operacyjny podobnie jak w przypadku JIT.

Te dwa wymienione wyżej mechanizmy są bardzo istotne gdyż pozwalają aplikacjom .NET działać zauważalnie szybciej dla użytkownika niż aplikacje pisane w technologii JAVA. Duża szybkość jest także ważnym argumentem w toczącej się od lat debacie na temat wad podejścia zakładającego kod/warstwę pośrednią (jak byte code Java czy MSIL Microsoftu) – do niedawna większość aplikacji wykonujących skomplikowane obliczenia musiało być pisane w językach bezpośrednio kompilowalnych do postaci kodu maszynowego.

Oparcie platformy aplikacyjnej o kod pośredni jest także ważnym czynnikiem w rozwiązywaniu problemu heterogeniczności systemów operacyjnych i sprzętu, na którym działają. Technologia Java z założenia miała być uniwersalna, aby pisane w niej aplikacje mogły być uruchamiane na systemach operacyjnych o odmiennej architekturze. Podobnie platforma .NET doczekała się tego typu uniwersalizacji pomimo braku takich założeń ze strony jej twórcy – Microsoftu. Grupa programistów opracowała wersję open-source platformy .NET w ramach projektu „Mono". Projekt ten umożliwił uruchamianie aplikacji .NET na komputerach pracujących pod kontrolą systemu operacyjnego Linux.



Rys. 4.3.1 Komponenty odpowiedzialne za kompilację i uruchamianie aplikacji .NET
[Źródło: www.wikipedia.org]

4.4 Architektura .NET Framework

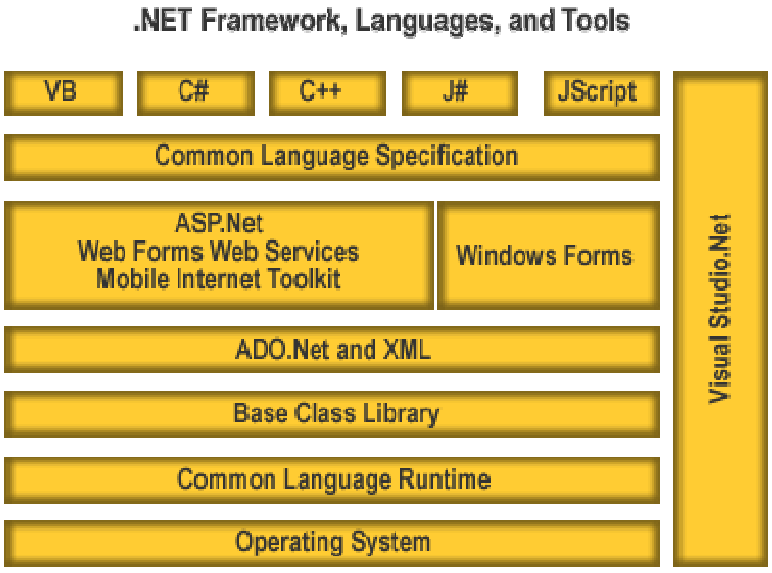
Zespół kluczowych technologii platformy .NET niezbędnych do uruchamiania aplikacji został nazwany .NET Framework. Jego dwoma podstawowymi komponentami są:

- Modułu wykonawczy Common Language Runtime (opisany w sekcji 4.3 opisującej kodu zarządzany)
- Base Class Library (Podstawowy Zestaw Klas)

Base Class Library (BCL) to zestaw klas i interfejsów z których bezpośrednio korzysta programista pisząc aplikacje .NET. Jest on zorganizowany w tzw. przestrzenie nazw (ang. namespaces), które grupują klasy i interfejsy dostarczające zbliżoną funkcjonalność. W jego skład wchodzą m.in. biblioteki do:

- Obsługi systemu plików – zarówno poprzez niskopoziomowe strumienie jak i wysoko poziomowe klasy File i Direcotry (przestrzeń nazw System.IO)
- Budowania interfejsów systemu Windows w technologii Windows Forms z obsługą wyświetlania; obsługą zdarzeń, interakcji ze źródłami danych (data binding). Biblioteki te znajdują się w przestrzeni System.Windows.Forms.
- Obsługa komunikacji sieciowej – zarówno na podstawie niskopoziomowych socketów jak i wyokopoziomowych klas obsługujących konkretne protokoły (TCP, HTTP) oraz implementacje standardów takich jak Web Services (przestrzenie nazw System.Net, System.Net.Sockets, System.Web.Services)
- Zestaw bibliotek składających się na ASP.NET, czyli technologię dynamicznego generowania stron WWW. Zawiera biblioteki zarówno do tworzenia samych stron jak i zarządzania hostingiem aplikacji (m.in. przestrzeń System.Web).
- Zestaw bibliotek składających się na ADO.NET, czyli technologię łączenia się i interakcji ze źródłami danych poprzez wbudowane sterowniki takie jak SqlClient, OracleClient jak również dzięki technologiom ODBC, OLE DB. Biblioteki te zawarte są w przestrzeni nazw System.Data.
- Zestaw bibliotek do zarządzania ustawieniami konfiguracyjnymi zapisywanymi w plikach XML (przestrzeń nazw System.Configuration)

- Zestaw bibliotek do bezpośredniej komunikacji z systemem operacyjnym Windows poprzez interfejs Win32 API oraz do komunikacji z komponentami zbudowanymi w oparciu o poprzednika Microsoft.NET – technologię COM. (przestrzeń nazw System.Runtime.InteropServices).



Rys. 4.4.1 Architektura Platformy .NET [Źródło: http://www.dotvb.com]

Z wyżej wymienionych, szczególnie ważne dla niniejszej pracy są komponenty ADO.NET, ASP.NET oraz Windows Forms. ADO.NET jest technologią w ramach, której został zrealizowany ODRA Data Provider, czyli komponent umożliwiający interakcję aplikacji platformy .NET z systemem bazodanowym ODRA. ASP.NET i Windows Forms są technologiami tworzenia odpowiednio Web-owych i Windows-owych interfejsów graficznych, w których można w bezpośrednia sposób wykorzystać funkcjonalność ODRA Data Provider. Obie wspomniane technologie wspierają technikę data binding, czyli deklaratywnego łączenia kontrolek wizualnych ze źródłami danych zgodnych z ADO.NET.

Również kluczowe okazały się komponenty zawarte w przestrzeni System.Net.Sockets. W modelu zawartego w pracy rozwiązania pełnią rolę pomostu między technologią Java (instancją bazy danych ODRA nasłuchującej na porcie programowym) a platformą .NET, w której zrealizowany jest ODRA Data Provider.

4.5 Technologia ADO.NET

4.5.1 Ogólny zarys technologii ADO.NET

Podstawową technologią do łączenia się i interakcji ze źródłami danych jest na platformie Microsoft.NET technologia ADO.NET, która wywodzi się z technologii wcześniejszej od samego Microsoft.NET – ADO czyli ActiveX Data Objects. Był to komponent mapujący dane (głównie relacyjne) ze źródeł danych niezależnych dostawców na obiekty programistyczne gotowe do użycia w aplikacjach C++ i Visual Basic dla systemu Windows a także z poziomu technologii generowania dynamicznych stron WWW ASP - Active Server Pages – protoplasty technologii wchodzącej w skład platformy .NET ASP.NET.

Poprzednikiem ADO na platformie Microsoft Windows były z kolei technologie Microsoft Data Access Objects (DAO) oraz Microsoft Remote Data Objects, które były pierwszym powszechnym obiektowych interfejsem do źródeł danych dla programistów Windows.

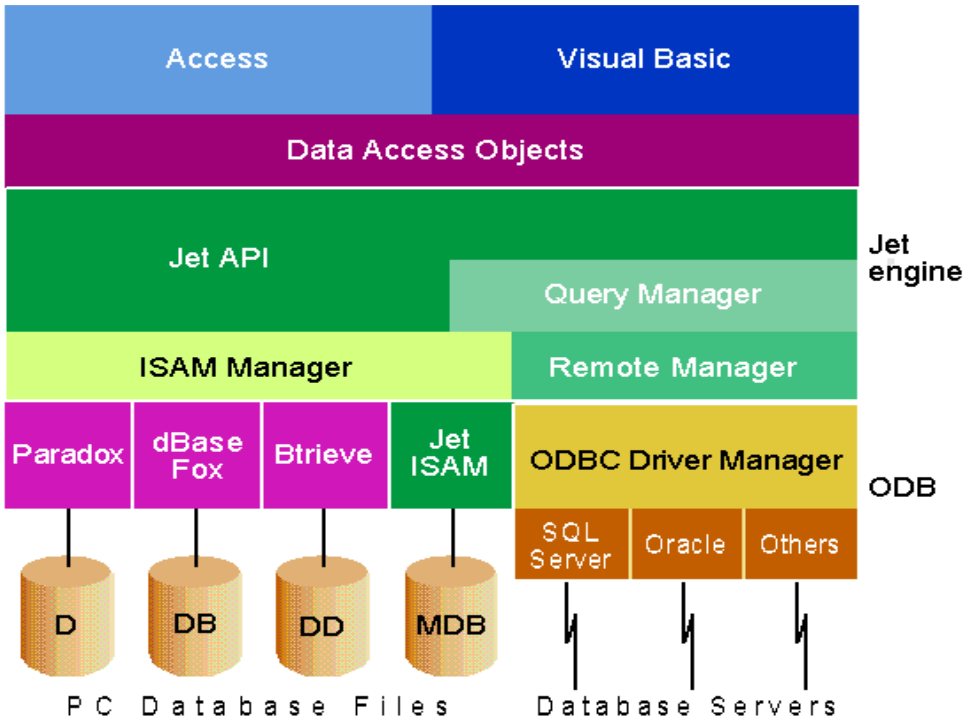
Ewolucja interfejsów środowisko programowania – źródła danych od DAO do ADO.NET odzwierciedla ewolucję, jaką przeszyły aplikacje dla komputerów osobistych w ciągu ostatnich 15 – 20 lat – od aplikacji działających w pełnej izolacji na jednym komputerze do aplikacji zbudowanych wedle architektury klient – serwer, gdzie aplikacja łączy się do wielu heterogenicznych i często odległych źródeł danych. Ponieważ w architekturze klient – serwer komputery występujące w roli serwera muszą często obsługiwać setki współbieżnych żądań na sekundę, utrzymywanie stałych połączeń byłoby zbyt kosztowne i uniemożliwiłoby skalowalność, czyli zdolność obsługi szybko rosnącej liczby współbieżnych połączeń przy stosunkowo niskim nakładzie środków na rozbudowę infrastruktury. Z tego powodu ADO.NET jest technologią szczególnie ukierunkowaną na pracę na „odłączonych danych” (disconnected data), czyli paradygmacie pracy, w którym należy maksymalnie krótko utrzymywać aktywne połączenie ze źródłem danych (głównie w celu pobrania i wysłania danych) a wszelkie czasochłonne obliczenia wykonywać w trybie „offline”.

4.5.2 Technologia DAO

Data Access Objects (DAO) to technologia, którą Microsoft opublikował w 1992 roku razem z silnikiem bazodanowym JET, który to, podobnie jak same biblioteki DAO, zyskał dużą popularność dzięki zbudowanemu na jego podstawie programowi Microsoft Access.

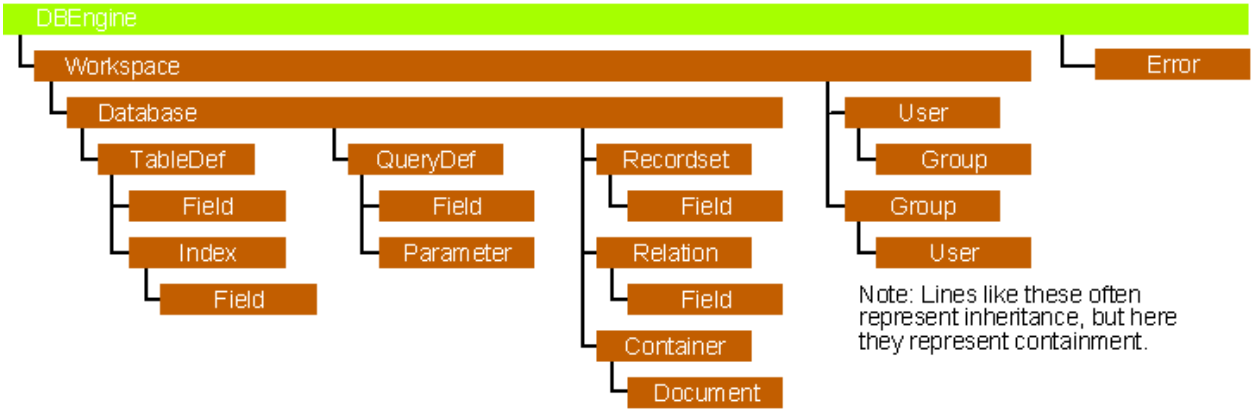
Data Access Objects to zestaw klas i interfejsów dostarczonych w postaci obiektów OLE (technologia komponentowa Windows będąca protoplastą technologii COM) pozwalających w wygodny dla programisty wysoko poziomowy sposób obsługiwać łączenie się i interakcję ze źródłem danych w technologii JET.

DAO pozwalało m.in. programistom Visual Basic i C++ dostęp do operacji ISAM (Indexed Sequential Access Method), które były podstawowym elementem technologii JET. Było to szczególnie ważne w przypadku programistów C++, którzy przed pojawieniem się wersji DAO kompatybilnej z tym językiem, musieli odwoływać się do źródła danych JET poprzez ODBC (technologia ta była przystosowana do baz danych w architekturze klient serwer co czyniło ją mało wygodną w przypadku obsługi JET).



Rys. 4.5.2.1 Schemat rozwiązań bazodanowych wykorzystujących Data Access Objects
[Źródło: www.microsoft.com/msj/archive/S212.aspx]

Technologia DAO udostępniała programiście obiektowy model reprezentujący strukturę bazy danych JET oraz operacje, które możemy na niej wykonać. Podobnie późniejsze technologie ADO oraz ADO.NET posiadają swój obiektowy model źródła danych. Model ten, ze względu na swoją niezależność od konkretnie podpiętego do aplikacji źródła danych, pozwala programiście w uniwersalny sposób obsługiwać heterogeniczne bazy danych.



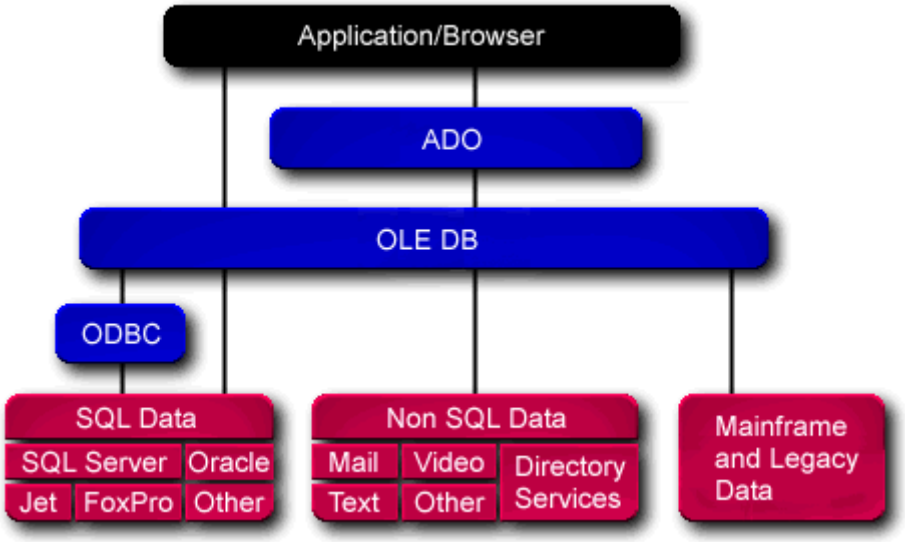
Rys. 4.5.2.2 Model obiektowy DAO [Źródło Microsoft Systems Journal (archiwum)]

Sukces technologii DAO polegający na wygodzie pracy programistów doprowadził do powstania technologii Remote Data Objects (RDO), czyli obiektowego interfejsu do źródeł danych kompatybilnych z technologią ODBC, której interfejs programistyczny ODBC API jest dużo bardziej skomplikowany z punktu widzenia programisty aniżeli DAO.

4.5.3 Technologia ActiveX Data Objects

ActiveX Data Objects (ADO) to programistyczny interfejs do obsługi heterogenicznych źródeł danych zbudowany w oparciu o doświadczenia z technologiami DAO oraz RDO. Podstawową ideą przyświecającą powstaniu ADO był uniwersalny dostęp do danych (Universal Data Access – UDA). DAO był interfejsem programistycznym dla silnika bazodanowego JET, który okazał się tak wygodny dla programistów, że na jego podstawie opracowano RDO aby pozwolić w równie prosty i jednolity sposób korzystać z heterogenicznych baz danych kompatybilnych z ze standardem ODBC. Kolejnym krokiem mającym na celu zbudowanie jednolitego modelu dostępu do danych była technologia OLE DB, która miała na celu dalsze rozszerzenie uniwersalnego dostępu do danych o możliwość korzystania ze źródeł nie kompatybilnych z ODBC. W technologii OLE DB wyróżniamy dwa komponenty prowadzące ze sobą interakcję: data provider (dostarczający dane) i data consumer (konsumujący dane). Oba komponenty zaimplementowane są jako obiekty COM implementujące wyspecyfikowane w standardzie OLE DB interfejsy. ADO jest technologią, która stanowi warstwę pośredniczącą pomiędzy komponentami COM OLE DB a aplikacją korzystającą z danego źródła danych. ADO występuje w roli data consumer i udostępnia aplikacji obiekty, dzięki którym programista w jednolity sposób obsługuje heterogeniczne

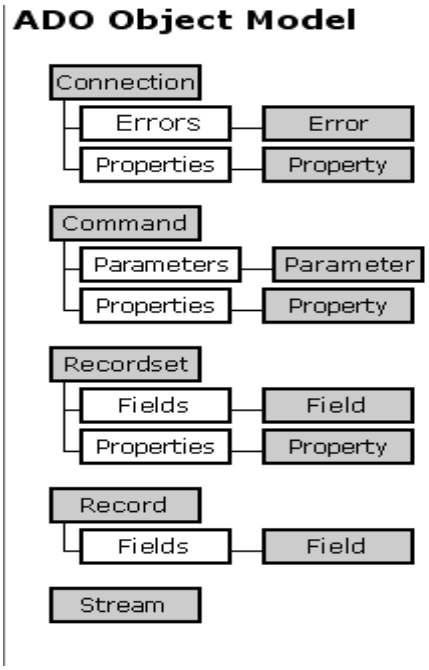
źródła danych. Pełni również funkcję walidatora sprawdzając i poprawiając dostarczane przez OLE DB dane.



Rys. 4.5.3.1 Rola ActiveX Data Objects w architekturze aplikacji bazodanowych.

Dzięki ADO programista uzyskuje jednolity dostęp do odmiennych źródeł danych bez potrzeby znajomości szczegółów ich organizacji i implementacji gdyż odwołuje się do nich poprzez uniwersalne obiekty ADO. Co więcej programista nie musi znać języka SQL w przypadku relacyjnych baz danych (chociaż może zejść do jego poziomu) co pozwala na wykorzystanie specyficznych funkcji danej bazy. Podstawowe obiekty ADO:

- Connection – reprezentuje połączenie do konkretnej instancji źródła danych
- Command – reprezentuje polecenie/zapytanie
- Field – reprezentuje kolumnę/atrybut obiektów składowanych w źródle danych
- Parameter – reprezentuje parametr polecenia/ zapytania
- Recordset – reprezentuje zbiór rekordów zwrócony przez źródło danych w wyniku wywołania polecenia/zapytania



Rys. 4.5.3.2 Model obiektów ActiveX Data Objects [Źródło: www.faculty.mcneese.edu]

Typowy paradygmat pracy za pomocą ADO to

1. Utworzenie obiektu Connection powiązanego z instancją danego źródła danych i otwarcie go
2. Utworzenie obiektu Recordset, do którego zostaną załadowane rezultaty
3. Otworzenie połączenia
4. Wywołanie metody Open na obiekcie Recordset i podanie jako parametru zapytania. Metoda ta wypełni Recordset wynikami zapytania.
5. Wykonanie operacji logiki biznesowej aplikacji na danych w obiekcie Recordset
6. Wywołanie metody Update lub UpdateBatch w celu załadowania zmienionych danych z powrotem do źródła.
7. Zamknięcie obiektów Recordset i Connection

ADO stało się podstawową technologią interakcji ze źródłami danych dla platformy Windows przed nadejściem platformy .NET. Było wykorzystywane w aplikacjach klienckich z poziomu języków C++ oraz Visual Basic oraz w technologii dynamicznego generowania stron WWW – Active Server Pages (ASP).

4.5.4 ADO.NET

ADO.NET to bezpośredni następca technologii ActiveX Data Objects. Jest nie tylko dostosowaniem technologii ADO do współpracy z platformą .NET, ale także znacznym rozwinięciem poprzedniej technologii zorientowanym na pracę w systemach o architekturze klient – serwer i w środowisku odłączonych danych (disconnected data). Podstawowe usprawnienia wprowadzone w ADO.NET to:

- Zorientowanie na prace w trybie żądanie/odpowiedź w przeciwieństwie do skoncentrowanego na aktywnym połączeniu trybowi pracy ADO.
- Zorientowanie na pracę w aplikacji o architekturze wielowarstwowej, w której istnieje potrzeba przenoszenia pomiędzy warstwami odłączonych od bazy obiektów.
- Wyższy poziom abstrakcji obiektu przechowującego dane – DataSet w porównaniu do Recordset w ADO. DataSet jest w stanie reprezentować całą odległą bazę danych (w tym relacje pomiędzy tabelami i ograniczenia), podczas gdy Recordset przechowywał jedynie zbiór rekordów otrzymanych w wyniku konkretnego zapytania i umożliwiał do nich dostęp sekwencyjny tylko do przodu (forward only). Dodatkowo Recordset przez cały czas interakcji na danych musiał być aktywnie połączony ze źródłem danych.
- Wszechstronne wsparcie XML – zarówno w komunikacji między warstwami, serwerem a klientem a także w przypadku serializacji obiektów.
- Większa kontrola nad wykonywanymi na DataSet-cie operacjami dzięki obiektom typu DataAdapter pozwalającym definiować konkretne polecenia w języku bazy danych oraz dzięki mechanizmowi śledzenia zmian.
- Wsparcie meta danych – ADO odkrywało typy danych ze źródła w sposób jedynie automatyczny. ADO.NET pozwala definiować schematy obiektów DataSet w XMLu w tym również za pomocą graficznego narzędzia dostępnego w Microsoft Visual Studio.
- Dane pobrane ze źródła do obiektu DataSet mogą być udostępniane za pomocą kolekcji i tablic danego języka programowania .NET, a nie jak w przypadku ADO za pomocą obiektu Recordset, który był obiektowym opakowaniem bazodanowego kursora.

- W ADO.NET zwracane ze źródła dane mogą być bezpośrednio mapowane na typy .NET lub być reprezentowane przez specjalne typy dostarczone przez sterownik do danej bazy danych – przykładowo dla Microsoft SQL Server są to typy zawarte w przestrzeni nazw System.Data.SqlTypes. W ADO zwracane rezultaty był typu Variant.
- Dataset ADO.NET posiada mechanizmy śledzenia zmian na danych w przypadku pracy w trybie offline, dzięki czemu przy ponownym połączeniu ze źródłem danych wysyłane są jedynie faktycznie zmienione rekordy.

Zorientowanie mechanizmów ADO.NET na pracę w środowisku rozłączonym (ang. disconnected) to odwzorowanie tendencji ostatniej dekady do budowania dużych aplikacji o architekturze klient – serwer z wykorzystaniem rozległych sieci teleinformatycznych. Wykorzystywane wcześniej przez ADO techniki interakcji z danymi wymagające utrzymywania aktywnych połączeń i obiektów typu kursor po stronie serwera nie pozwalały na budowę systemów skalowalnych, czyli takich, które w miarę dynamicznego wzrostu współbieżnych połączeń wymagają stosunkowo niskich nakładów na infrastrukturę. Jedną z podstawowych zasad budowy systemów skalowalnych jest jak najmniejsze wykorzystanie zasobów komputera występującego w roli serwera a utrzymywanie aktywnych połączeń i kursorów serwerowych w ADO w oczywisty sposób przeczą tej zasadzie.

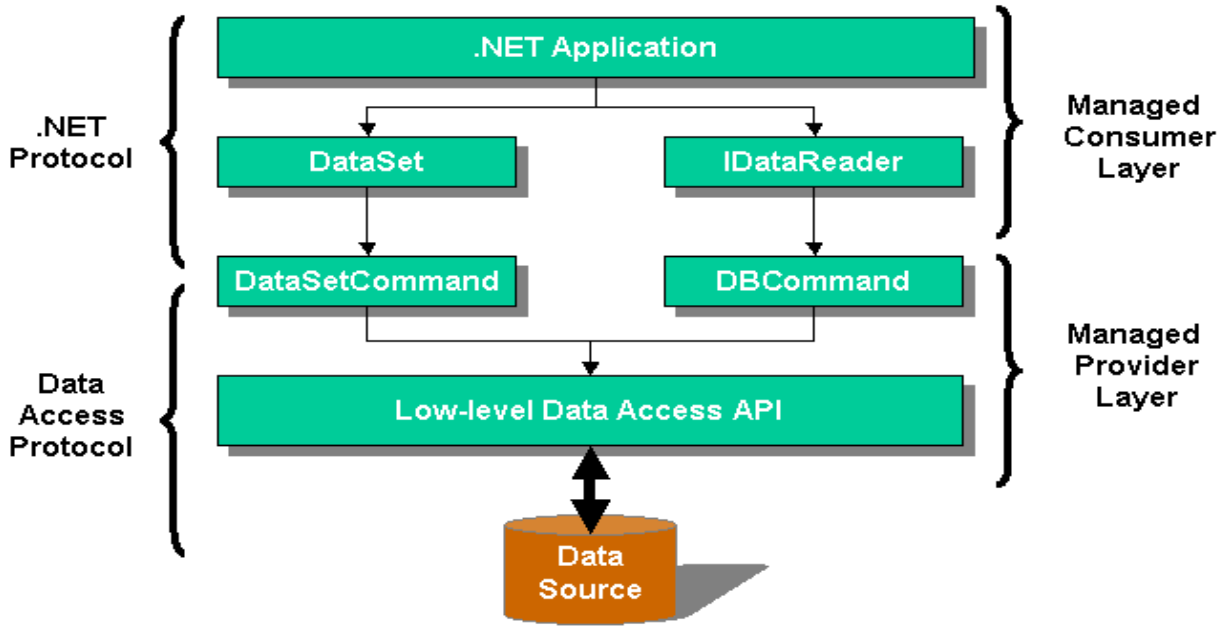
Praca w trybie disconnected oraz dogłębne wsparcie dla technologii XML (od możliwości przechowywania całego obiektu Dataset w postaci pliku XML aż po używanie tego języka w komunikacji sieciowej) to także odzwierciedlenie coraz większej potrzeby integracji niezależnych i odizolowanych niegdyś systemów różnych dostawców. Jedną z podstawowych technologii mających na celu integrację oprogramowania pochodzącego od niezależnych dostawców jest XML Web Services, które platforma .NET wspiera od swojej pierwszej wersji. W przypadku Web Services obiekt Dataset nabiera szczególnego znaczenia gdyż jego łatwa serializacja do postaci XML pozwala na wymianę oraz synchronizację danych pomiędzy odległymi bazami/źródłami danych w bardzo wygodny sposób typu message – driven (ukierunkowanie na komunikaty), czyli w trybie jednorazowych transmisji niewymagających istnienia stałych, aktywnych połączeń.

4.5.5 Architektura ADO.NET

Architektura ADO.NET składa się z dwóch podstawowych warstw:

- .NET Protocol – czyli zespół klas i interfejsów do interakcji z danymi wspólny dla całej platformy .NET i niezależny od modelu konkretnego źródła danych. To z obiektami tej warstwy prowadzi interakcję programista budujący aplikacje. Oprogramowanie tego poziomu odwzorowuje także dane niższej warstwy na uniwersalne konstrukcje .NET takie jak DataReader czy DBCommand. W niniejszej pracy odwzorowaniem tego poziomu jest projekt ODRAClient
- Data Access Protocol to oprogramowanie wykorzystujące specyficzne cechy konkretnego źródła danych i wykonujące polecenia pochodzące z .NET Protocol. Jest to warstwa dostarczana przez dostawce konkretnej bazy danych, zapisująca i pobierająca dane w formacie specyficznym dla danego produktu. Odwzorowaniem tego poziomu w niniejszej pracy jest projekt ODRADriver, który komunikuje się z systemem ODRA za pomocą niskopoziomowych gniazd (sockets), konwertuje strumienie bajtów na typy .NET (wciąż jednak zbudowane w sposób specyficzny dla ODRA) a następnie udostępnia je projektowi ODRAClient, który z kolei udostępnia je aplikacjom .NET w postaci obiektów ADO.NET takich jak DataReader.

W ADO.NET warstwę Data Access Protocol i .NET Protocol zapewniają komponenty zwane Data Providers (dostawcy danych). Są to komponenty łączące w sobie oprogramowanie wyspecjalizowane w obsłudze specyficznego źródła danych (jego formatu/modelu/organizacji oraz języka poleceń/zapytań) z oprogramowaniem odpowiedzialnym za odwzorowanie funkcjonujących w nim typów danych na typy platformy .NET oraz odpowiedzialnym za dostarczenie standardowych obiektów ADO.NET takich jak Connection, DBCommand czy DataReader, które wykorzystywane są bezpośrednio przez programistę budującego aplikację .NET. Platforma .NET posiada wbudowane Data Providers dla ODBC, OLE DB oraz Microsoft SQL Server. Data Providers dla innych baz danych dostarczane są głównie przez ich dostawców.



Rys. 4.5.5.1 Architektura technologii ADO.NET [Źródło: CodeGuru]

4.5.6 Model obiektów składających się na ADO.NET

Podobnie jak Data Access Objects i ActiveX Data Objects także ADO.NET ma swój jednolity model obiektów za pomocą, których programista aplikacyjny prowadzi interakcję ze źródłem danych. W przypadku ADO.NET obiekty te można podzielić na 2 grupy:

- Obiekty wchodzące w skład Data Providera
- Obiekty wchodzące w skład DataSet.

Obiekty tworzące Data Provider to:

- Connection – jest to obiekt reprezentujący aktywne połączenie z konkretną instancją danego źródła danych. Obsługuje niskopoziomową komunikację pomiędzy obiektami ADO.NET takimi jak Command czy DataReader a źródłem danych. Połączenie z konkretną instancją źródła odbywa się na podstawie informacji zawartych w tzw. Connection String czyli ciągu znaków zawierającym informacje w postaci klucz = wartość potrzebnymi do uruchomienia połączenia. Standardowo są to adres i port źródła danych, dane autentykacyjne login i hasło oraz ewentualnie opcje specyficzne dla danego źródła danych. Connection obsługuje także obiekt typu Transaction reprezentujący transakcję ACID na źródle danych.

- Command – jest to obiekt reprezentujący konkretne polecenie lub zapytanie wyrażone w języku obsługiwanym przez źródło danych (czyli np. SQL dla relacyjnych baz danych bądź SBQL dla systemu ODRA). Obiekt Command działa na podstawie otwartego/aktywnego obiektu Connection oraz w ramach danego obiektu Transaction jeśli taki istnieje i jest przypisany do Connection. Command udostępnia programiście następujące metody:
    - ExecuteNonQuery() – jest to metoda pobierająca jako argument polecenie/zapytanie w języku źródła danych i wykonująca je bez zwracania rezultatów ale zwracając liczbę rekordów, które zostały przez zapytanie przetworzone. Przeznaczona jest do poleceń mających na celu zmiany w strukturze danych bez potrzeby wyświetlenia ich nowego stanu.
    - ExecuteScalar() – metoda pobierająca jako argument polecenie w języku źródła i zwracająca pojedynczą wartość – w przypadku, gdy polecenie zwróci większą ilość danych, zwracana jest wartość pierwszego atrybutu pierwszego zwróconego obiektu.
    - ExecuteReader() – metoda pobierająca jako argument zapytanie w języku źródła i zwracająca pełny zbiór danych będący wynikiem zapytania w postaci obiektu DataReader, który umożliwia jednokierunkowy (forward Orly – tylko do przodu), sekwencyjny dostęp do zwróconych danych. Jest to najczęściej wywoływana metoda w pracy z obiektem Command przy czym często wywoływana jest przez obiekty wyższego rzędu takie jak table adapter.

  Command korzysta z obiektów typu Parameter w przypadku, gdy zapytanie do źródła danych powinno być sparametryzowane np. danymi z interfejsu użytkownika. Parameter jest także obiektowym opakowaniem parametrów przekazywanych do procedur składowanych w źródle danych.
- DataReader – obiekt odpowiadający po części obiektowi Recordset w technologii ADO –zapewnia szybki, sekwencyjny, dostęp w trybie tylko do przodu (forward only) do danych zwróconych przez konkretne zapytanie. Obiekt ten utrzymuje aktywne połączenie, dlatego wykorzystywany jest do szybkiego wczytania danych ze źródła w celu ich wyświetlenia bądź późniejszego przetworzenia i od razu zamykany. DataReader pozwala na odwoływanie się do zwróconych danych zarówno bez podania konkretnego typu .NET, jakiego daną chciałoby się otrzymać (zwracany jest obiekt

typu Object) jak również poprzez metody narzucające konkretny typ (co jednak może wywołać wyjątek w przypadku próby zwrócenia błędnego typu lub gdy zapytanie zwróciło obiekt typu DBNull.Value). Do odczytu danych ze źródła służy metoda Read(), która przy każdym wywołaniu przesuwa kursor wyników o jedną pozycję do przodu i udostępnia poprzez indeksator obiektu DataReader lub przez metody zwracające obiekty konkretnych typów, dane z konkretnych pól aktualnego wiersza. W wersji 2.0 ADO.NET możliwe jest także utrzymywanie wielu aktywnych kursorów do wyników zapytania. DataReader uzyskiwany jest poprzez wywołanie metody ExecuteReader na obiekcie DBCommand.

- DataAdapter – jest obiektem pośredniczącym pomiędzy obiektami Connection, Command, DataReader a obiektem DataSet reprezentującym dane ze źródła w trybie odłączonym. DataAdapter definiuje 4 obiekty Command odpowiadające za operacje CRUD na źródle danych – odpowiednio InsertCommand, SelectCommand, UpdateCommand, DeleteCommand. Obiekty te tworzone są przez specjalny obiekt CommanBuilder lub mogą być definiowane explicite przez programistę, co daje dużą elastyczność w korzystaniu z danego źródła danych. Najważniejsze metody obiektu DataAdapter to:
    - Fill() – metoda wypełniająca danymi podany, jako argument, obiekt typu DataSet. Korzysta w tym celu ze swojego obiektu SelectCommand a dokładniej ze zwróconego przez SelectCommand DataReader-a. Metoda ta pobierając dane ze źródła jest w stanie odtworzyć schemat danych, czyli m.in. podział na tabele i kolumny (przydziela także nazwy kolumn na podstawie zwróconych przez źródło danych meta danych)
    - Update() – metoda uaktualniająca źródło danych wedle zmian wprowadzonych w DataSet od chwili załadowania. Ponieważ obiekt DataSet śledzi zmiany, DataAdapter przegląda wszystkie rekordy zawarte w DataSet i po natrafieniu na te, które zostały oznaczone jako zmienione, korzysta z obiektu UpdateCommand w celu załadowania zmienionych danych. Programista może ingerować w ustawione przez mechanizm śledzenia zmian statusy wierszy (niezmieniony, zmieniony, usunięty) co daję duża elastyczność w decydowaniu jakiego typu zmiany powinny zostać odzwierciedlone w źródle.

Odpowiednie skonfigurowanie obiektów Command obiektu DataAdapter pozwala na dostosowanie praktycznie każdego źródła z zaimplementowanymi obiektami Connection, Command i DataReader do współpracy z obiektem DataSet.

Obiekt DataSet jest centralnym obiektem ADO.NET jeśli chodzi o podejście przetwarzania danych w trybie disconnected. Obiekt ten powinien być jednak wykorzystywany tylko w przypadku, gdy dostęp do źródła danych jest kosztowny i odległy (np. potrzeba transmisji danych z odległego źródła poprzez Web Services) lub gdy istnieje potrzeba przeprowadzenia stosukowo czasochłonnych obliczeń i zatrzymywanie na ten czas aktywnego połączenia ze źródłem danych przełożyłoby się na zmniejszenie skalowalności systemu. Utworzenie obiektu DataSet i zapełnienie go danymi jest obliczeniowo kosztowne gdyż stanowi w pewnym stopniu mechanizm Object Relational Mapping, który opakowuje dane z bazy w odpowiednie obiekty typu DataRow i DataTable, dlatego nie powinien być wykorzystywany w przypadku prostych mechanizmów wyświetlania danych w aplikacjach sieciowych o dużym obciążeniu równoczesnymi połączeniami.

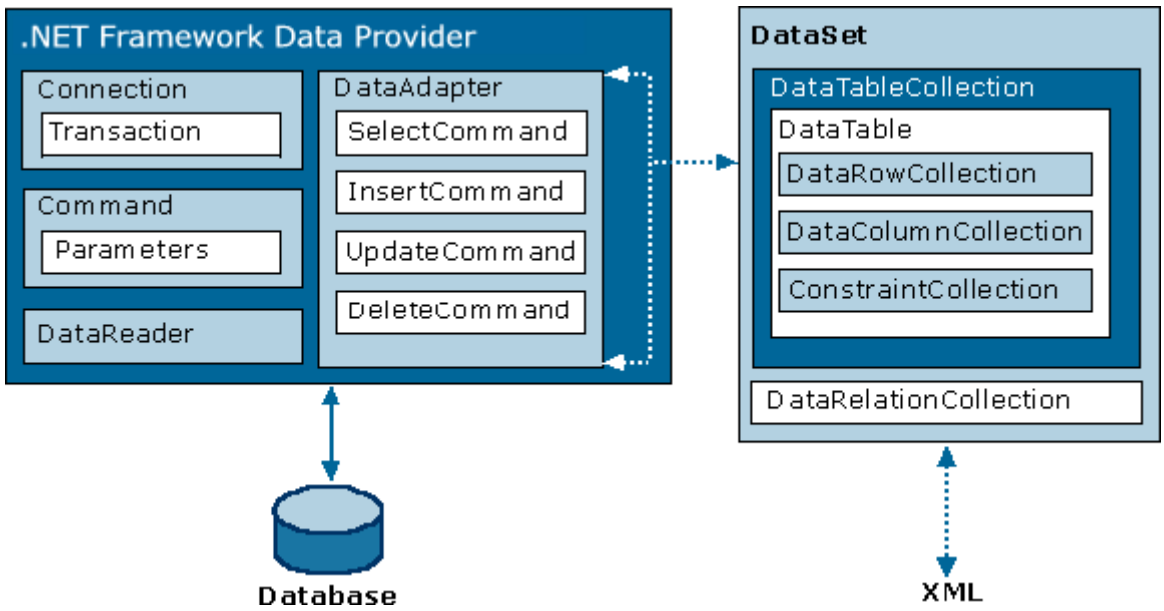
Obiekt DataSet jest w stanie odzwierciedlić strukturę całej odległej relacyjnej bazy danych (bądź innego źródła odpowiednio zmapowanego przez Data Provider) za pomocą następujących obiektów wchodzących w jego skład:

- DataTable – obiekt reprezentujący tabelę (lub obiekt w przypadku mapowania obiektowego źródła danych). Od wersji ADO.NET 2.0 może być również wykorzystywany niezależnie od DataSet (może być m.in. samodzielnie wypełniany danymi i serializowany). Zawiera wewnątrz kolekcję obiektów typu DataColumn, DataRow oraz Constraint.
- DataColumn – reprezentuje kolumnę tabeli występującej w źródle, m.in. zawiera informacje o nazwie i typie przechowywanej danej
- DataRow – reprezentuje rekord bazy danych. Przechowuje również informacje o stanie danych w stosunku do źródła (statusy: niezmieniony, zmieniony, usunięty) ustawiane automatycznie przez mechanizm śledzenia zmian i wykorzystywane w momencie wprowadzania zmian do źródła danych.
- Constraint – reprezentuje ograniczenia wartości danej kolumny występujące w bazie danych takie jak unikalność wartości, klucz główny.

- DataRelation – reprezentuje relację między tabelami źródła danych. DataRelation może być odpowiednikiem klucza obcego w relacyjnej bazie danych. Dzięki obiektom DataRelation możliwa jest nawigacja po danych – np. zwrócenie zbioru obiektów DataRow w wyniku wywołania metody GetChildRows() na danym DataRow co zwróci rekordy ze strony „wiele”.

DataSet może występować w dwóch postaciach:

- Generycznej/nietypowanej (untyped DataSet) – w tym przypadku do pól rekordów odwołujemy się podając ich nazwy, jako argument typu string; podobnie do tabel zawartych w DataSet. Tego typu DataSet to obiekt będący instancją System.Data.DataSet.
- Jawnie typowanej (typed DataSet) – jest to klasa dziedzicząca z klasy System.Data.DataSet jednakże złożona z tabel o nazwach i typach/nazwach kolumn odpowiadających strukturze konkretnej bazy danych.



Rys. 4.5.6.1 Obiekty występujące w ADO.NET [Źródło: msdn.microsoft.com]

Pomimo uniwersalności i elastyczności w podejściu ADO.NET do odmiennych źródeł danych, model obiektowy przyjęty w tej technologii ukierunkowany jest na źródła danych zorganizowane w sposób relacyjny. Z tego powodu interakcja z kompatybilnymi z ADO.NET obiektowymi bazami danych w dużym stopniu prowadzi do sztucznego odwzorowania

obiektów w struktury .NET przystosowane do danych relacyjnych, co miało też miejsce w przypadku napisanego w ramach niniejszej pracy ODRA Data Provider-a.

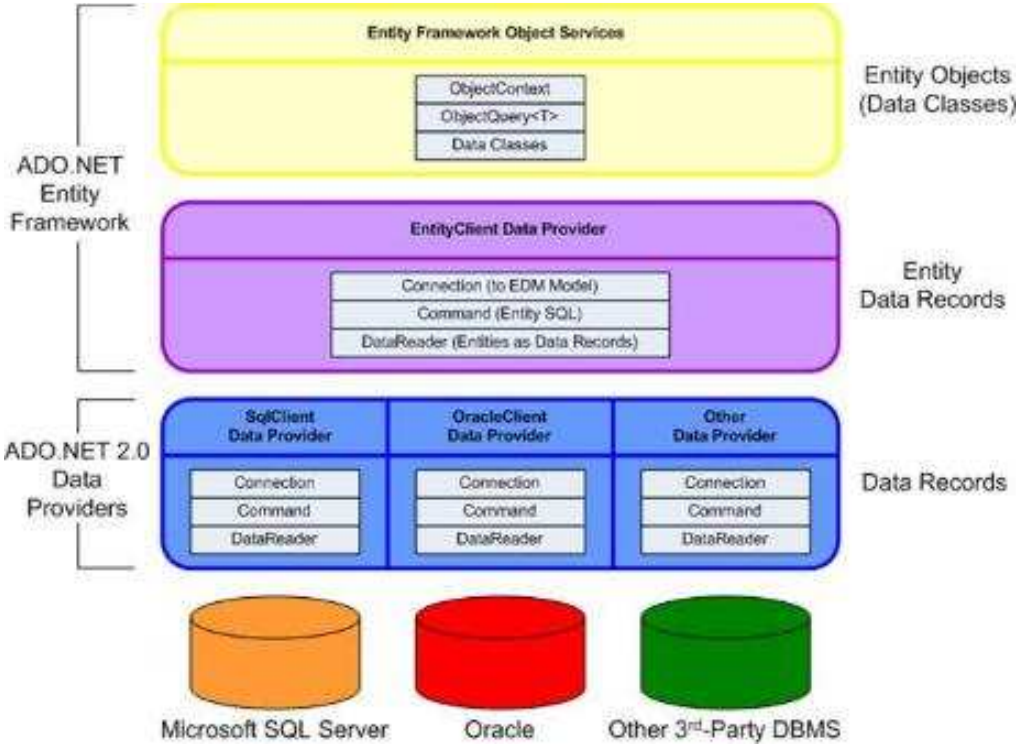
Zorientowanie na relacyjne źródła danych ADO.NET i jego poprzedników odzwierciedla koncepcję tworzenia aplikacji dla systemu Windows za pomocą narzędzi Microsoft. Narzędzia te wyrosły z proceduralnych środowisk programowania, dla których podstawowym źródłem były relacyjne bazy danych a także ze względu na fakt, iż rozwijanie aplikacji dla Windows miało być maksymalnie przystępne dla programistów. Z tego powodu technologie Microsoft nie są zorientowane na obiektowość w stopniu porównywalnym z technologiami Java i platformą J2EE, która od początku lansuje obiektowe odwzorowania danych w języku programowania (tzw. Entity Beans). Technologia Java została zaprojektowana od podstaw wokół obiektowości, podczas gdy technologie Microsoft, aby pozwolić wykorzystywać programistom zdobyte wcześniej doświadczenie w programowaniu proceduralnym, bardzo powoli i stopniowo wprowadzały pojęcia obiektowości. Nawet technologia .NET, która była swojego rodzaju rewolucją wśród technologii Microsoft (zrywała z wieloma przyjętymi wcześniej koncepcjami) nie była w pełni obiektowa. Wiele funkcjonalności (szczególnie związanych z wykorzystaniem Win32 API) zostało udostępnionych w .NET jedynie w postaci obiektowych „wrapperów", które będąc ograniczonymi przez proceduralną naturę funkcji systemowych które przykrywały, nie mogły w pełni implementować obiektowych mechanizmów. Podobnie stało się z technologiami dostępu do danych – pomimo, że ADO.NET ma swój model obiektów odzwierciedlający źródło danych to wciąż są to jedynie wrappery oparte o relacyjne pojęcia tabeli, rekordów i kluczy zewnętrznych a nie o typy i instancje obiektów oraz ich dowolne referencje jak to ma miejsce w przypadku obiektowych baz danych.

Praca w trybie odłączonym od źródła danych (disconnected) lansowana przez architekturę ADO.NET, ze względu na aplikacje o architekturze klient – serwer i związanego z tym wymogu możliwie krótkiego angażowania serwerowych zasobów takich jak aktywne połączenia, znacznie ogranicza zastosowanie obiektowych źródeł danych. W tego typu systemach wspierających relatywizm obiektowy, obiekty mogą mieć dowolnie rozbudowaną hierarchię, niemożliwą do przeniesienia na warstwę ADO.NET chyba, że ryzykując przeniesienie dużej porcji nie potrzebnych dla obsługi danego klienckiego żądania danych (co dodatkowo łączyłoby się ze znacznym kosztem obliczeniowym konwersji obiektów ze źródła danych do obiektów kompatybilnych z ADO.NET).

4.5.7 Entity Framework

Wersja platformy .NET 3.5 jako pierwsza zawiera technologię obiektowego dostępu do danych podobną do Entity Beans z Javy. Jest to tzw. Entity Framework będący nadbudówką na ADO.NET Data Provider i korzystający z niego na takiej samej zasadzie jak robi to DataSet. Ponieważ jednak obiekty generowane przez Entity Framework są tylko odzwierciedleniem danych występujących w konkretnym źródle, a nie istnieje sposób pełnego przeniesienia obiektów ze źródła danych do aplikacji .NET, w niniejszej pracy technologia ta została pominięta w wyniku czego opracowano i zaimplementowano projekt integrujący system bazodanowy ODRA z .NET na poziomie Data Providera bez włączenia wyższej warstwy, którą reprezentuje Entity Framework.

Należy zauważyć, że istnieje możliwość pełnej integracji platformy .NET ze środowiskiem obiektowej bazy danych bez jakichkolwiek mechanizmów pośredniczących, ale ogranicza się jedynie do produktu Microsoft SQL Server 2005/2008. Ten silnik bazodanowy ma rozbudowane możliwości przechowywania tabel z kolumnami których typem danych jest klasa .NET – wówczas możemy mówić o obiektowej bazie danych, pomimo że SQL Server 2005/2008 pozostaje silnikiem bazodanowym wspierającym przede wszystkim relacyjny model danych. Takie podejście eliminuje m.in. koszt obliczeniowy mapowania danych relacyjnych na obiekty oraz koszt utraty części szczegółów środowiska bazodanowego i obiektowego na skutek mapowania przenoszonych pomiędzy nimi danych. Pomimo wymienionych znacznych zalet w przypadku przyjęcia takiego rozwiązania programiści skazani są na jednego dostawcę gdyż nie istnieją na rynku inne technologie pozwalające na tak ścisłą integrację z platformą .NET.



Rys. 4.5.7.1 Entity Framework jako kolejna warstwa ADO.NET korzystająca z Data Providerów.

4.5.8 Schemat działania aplikacji .NET wykorzystującej ADO.NET

Aplikacje pisane na platformę .NET wykorzystują ujednolicony schemat komunikacji ze źródłami danych udostępnianymi przez ADO.NET, który w większości przypadków zawiera następujące kroki:

1. Utworzenie obiektu Connection reprezentującego połączenie z konkretnym źródłem danych i zainicjalizowanie jego atrybutu ConnectionString parametrami połączenia z konkretną instancją źródła danych.
2. Utworzenie obiektu Command reprezentującego zapytanie/polecenie języka obsługiwanego przez źródło danych. Obiektowi Command należy przypisać tekst zapytania/polecenia oraz obiekt Connection, który zostanie wykorzystany w celu połączenie i wykonania zapytania.

3. Wywołanie metody Open() na obiekcie Connection, która tworzy aktywne połączenie. Niektóre obiekty ADO.NET (np. DataAdapter) mogą wywołać metodę Open() automatycznie.
4. Wywołanie na obiekcie Command jednej z metod: ExecuteNonQuery(), ExecuteScalar(), ExecuteReader().
5. W przypadku wywołania we wcześniejszym kroku metody ExecuteReader(), pobranie zwracanego obiektu DataReader i przegląd kolejnych wierszy danych za pomocą wywołań metody Read(). Aplikacje .NET a także komponenty takie jak DataAdapter czy kontrolki wizualne do prezentowania/interakcji z danymi, wykorzystują ten krok do pobrania listy zwróconych rekordów danych.
6. Wywołanie DataReader.Close() w celu zamknięcia obiektu i zwolnienia zasobów.
7. Wywołanie Connection.Close() w celu zamknięcia połączenia do źródła danych.

Ostatni krok jest szczególnie ważny ze względu na wspomnianą wcześniej kwestię skalowalności w architekturze klient – serwer. Aby w maksymalnym stopniu wykorzystać zasoby serwerowe w postaci dostępnych połączeń ze źródłem danych, należy zamykać aktywne połączenia w celu udostępnienia ich do obsługi kolejnych żądań klienta.

4.6 Microsoft Visual Studio 2008

Podstawowym narzędziem wykorzystanym w niniejszej pracy jest zintegrowane środowisko wytwarzania aplikacji (Integrated Development Evironment) – Microsoft Visual Studio 2008. Jest to edytor kodu źródłowego dla języków kompatybilnych z platformą .NET ze zintegrowanym kompilatorem, debuggerem oraz mechanizmami kontroli poprawności kodu i podpowiedzi działającego w trakcie pisania programu.

Visual Studio 2008 wyposażone jest w szereg narzędzi graficznych (tzw. designers), które wspomagają programistę w tworzeniu graficznych interfejsów do aplikacji Windows oraz ASP.NET. Produkt zawiera także rozbudowane narzędzia graficzne do modelowania danych w tym m.in. do tworzenia typowanych obiektów DataSet oraz diagramów encji najnowszego rozwinięcia ADO.NET – Entity Framework.

Visual Studio 2008 zawiera także rozbudowany debugger pozwalający na bieżąco podglądać a także zmieniać stan zmiennych; podgląd stosu czy też okienko Immediate, które na bieżąco kompiluje i uruchamia wpisywany kod programu. Narzędzia te okazały się

szczególnie pomocne w trakcie pisania niskopoziomowego ODRA Driver, ponieważ budowa bezpośredniego interfejsu do systemu ODRA za pomocą niskopoziomowych gniazd i strumieni wymagała szukania odpowiedniego mechanizmu transmisji danych. To zaś wymagało przeprowadzenia serii prób i zbadania, w jakim formacie bajtowym odpowiada ODRA i jak można go skonwertować do postaci zrozumiałej dla .NET oraz na odwrót.

4.7 ASP.NET oraz Winforms i DataBinding

ASP.NET to technologia budowy dynamicznie generowanych stron WWW dostępna na platformie Microsoft.NET. Wywodzi się ona z technologii Active Server Pages zbudowanej głównie w oparciu o model komponentowy COM. ASP.NET jest podstawową technologią do budowania aplikacji wykorzystujących, jako interfejs przeglądarkę stron internetowych WWW jak i oprogramowywania i udostępniania XML Web Services. W technologii tej interfejs użytkownika podzielony jest na widoki w postaci stron .aspx. Na funkcjonalność konkretnej strony składa się kod programu napisany w którymkolwiek z języków obsługiwanych przez platformę .NET oraz skrypt ASP będący wzbogaceniem języka HTML o dodatkowe elementy pozwalające na dynamiczne generowanie zawartości. Te dwa elementy – kod języka programowania i skrypt mogą występować w jednym pliku lub zostać rozdzielone na plik aspx i tzw. „code – behind".

Windows Forms („formatki Windows") jest technologią budowy aplikacji klienckich dla systemu Windows w oparciu o platformę .NET. Zawiera zestaw klas i interfejsów do obsługi wyświetlania aktywnych elementów GUI (tzw. kontrolek) jak np. listy elementów, przyciski itp. oraz mechanizmy do obsługi zdarzeń.

ASP.NET oraz Windows Forms wykorzystują mechanizm deklaratywnego łączenia elementów graficznego interfejsu użytkownika (w postaci kontrolek stron WWW czy okienek Windows) z danymi z heterogenicznych źródeł danych. Główną ideą przyświecającą opracowaniu DataBindingu było wyposażenie platformy .NET w uniwersalny mechanizm interakcji komponentów GUI ze źródłami danych oraz zwolnienie programisty z obowiązku pisania kodu do obsługi tej interakcji (jest to ogólna tendencja w technologii .NET, aby jak najwięcej funkcjonalności mogło być włączonych i skonfigurowanych w sposób

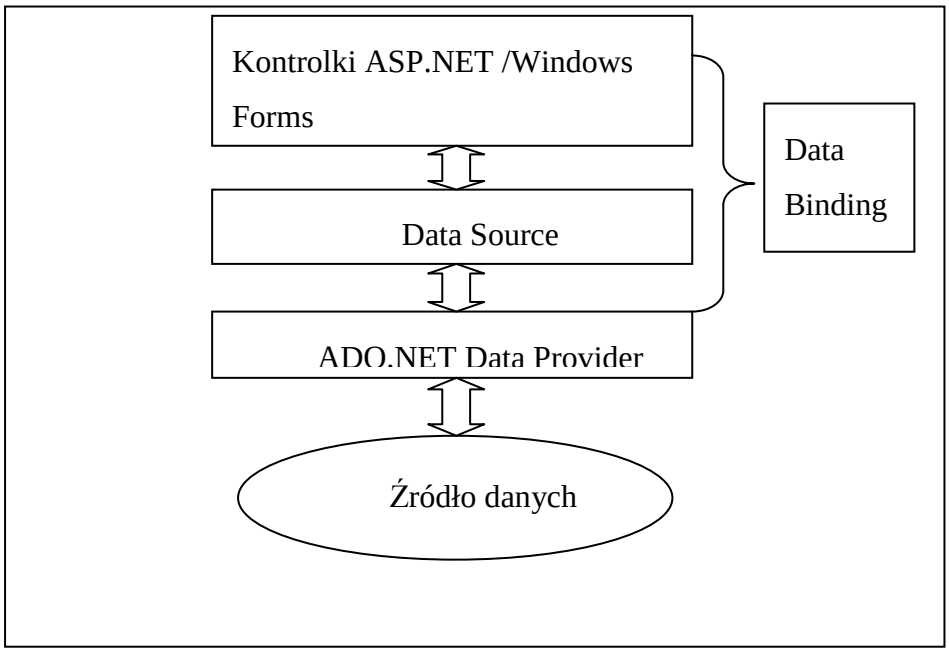
deklaratywny). Implementacja mechanizmu DataBinding dla konkretnej kontrolki sprowadza się do takiego jej oprogramowania, aby potrafiła wykorzystywać obiekty i interfejsy ADO.NET w stopniu pozwalającym na wyświetlenie danych oraz meta danych. Szczególną kontrolką jest DataGrid (w wersji .NET 2.0 i późniejszych dostępna jest także uproszczona wersja tej kontrolki pod nazwą GridView) która występuje zarówno w postaci kontrolki Windows Forms jak i ASP.NET (w obu przypadkach posiada te same atrybuty i zachowania, co jest bardzo wygodne dla programistów .NET). DataGrid nie tylko wyświetla dane ale udostępnia również funkcjonalność modyfikacji danych i przesyłania zmian do obiektów Data Providera ADO.NET w celu wprowadzenia ich do konkretnego źródła danych. Mechanizm DataBinding można podzielić na dwa elementy:

- Imperatywny DataBinding czyli podłączanie kontrolek ASP.NET/Windows Forms do danych w kodzie programu poprzez przypisanie atrybutowi DataSource (jest to standardowy atrybut, który posiadają wszystkie kontrolki obsługujące DataBinding) obiektu implementującego odpowiedni interfejs ADO.NET. Zazwyczaj przypisywany jest obiekt typu DataSet lub DataReader chociaż można wykorzystać jakikolwiek komponent implementujący metody interfejsu ADO.NET IDataSource. W przypadku projektów opracowanych w ramach niniejszej pracy, do kontrolki obsługującej DataBinding można przypisać ODRADataReader
- Deklaratywny – połączenia kontrolek interfejsu użytkownika z danymi jest jedynie deklarowane w części skryptowej strony ASP.NET lub w atrybutach foremki Windows Forms a wewnętrzne mechanizmy.NET odpowiedzialne są za faktyczne wypełnienie kontrolki danymi. W takim przypadku najczęściej stosuje się zadeklarowany w skrypcie obiekt typu DataSource (z platformą .NET dostarczone są standardowe DataSource: SqlDataSource, ObjectDataSource, XMLDataSource, AccessDataSource). W ramach deklaracji DataSource podaje się namiary źródła danych (connection string) oraz polecenia i parametry niezbędne do wykorzystania operacji CRUD. W samej kontrolce obsługującej DataBinding (np. DataGrid/GridView) należy za pomocą atrybutu DataSourceId przypisać zadeklarowany wpierw obiekt DataSource a także przypisać atrybutowi DataKey nazwę atrybutu/kolumny będącego unikalnym identyfikatorem/ kluczem głównym obiektów/rekordów wyświetlanych przez kontrolkę. W chwili wywołania danej akcji (update/delete) przez użytkownika kontrolka wywołuje odpowiednią metodę

DataSource przekazując jako argument DataKey oraz ewentualnie zmienione dane. Datasource natomiast wywołuje metody Data Providera dla konkretnego źródła danych.

Mechanizm DataBindingu jest podstawowym mechanizmem interakcji między interfejsem użytkownika a źródłami danych na platformie .NET. Jest napisany w sposób modularny pozwalający na podłączenie do kontrolek ASP.NET/Windows Forms bardzo odmiennych źródeł danych. Dzięki implementacji ODRA Data Provider możliwe jest również podpięcie pod interfejs użytkownika aplikacji ASP.NET /Windows Forms danych pochodzących z systemu ODRA.

DataBinding jest technologią cały czas rozwijaną i wspieraną nie tylko przez ASP.NET i Windows Forms ale także przez najnowsze technologie wchodzące w skład .NET Framework (i popularne szczególnie w najnowszym systemie Microsoft – Windows Vista) takie jak Windows Presentation Foundation (WPF) i Silverlight. Z tego względu, dzięki ODRA Data Provider, dane z systemu ODRA mogą być pobierane także przez komponenty WPF i Silverlight bez poważniejszych modyfikacji.



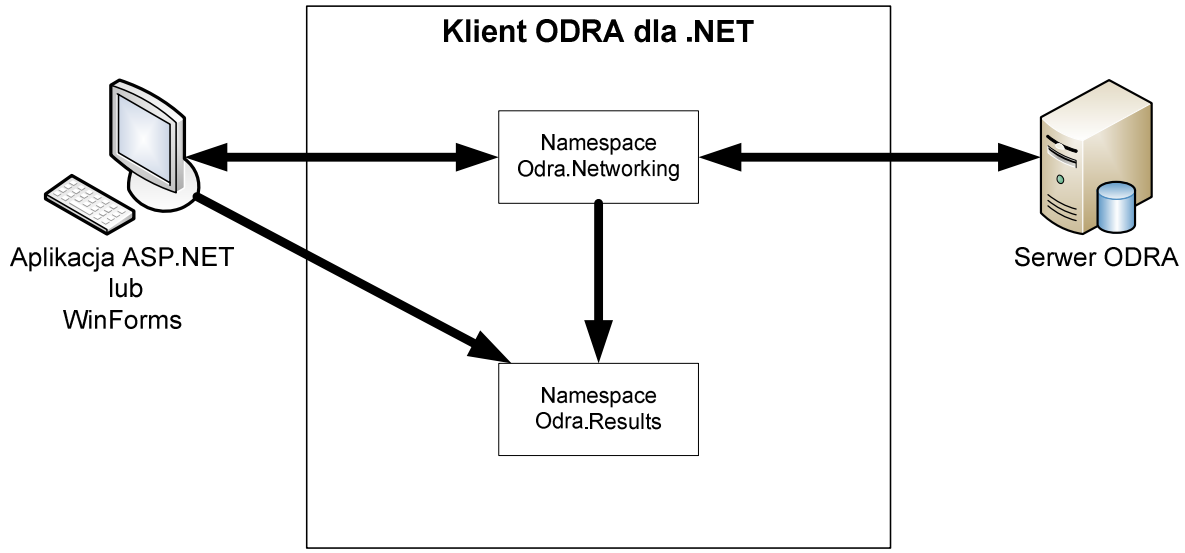
Rys. 4.7.1 Umiejscowienie Data Binding w architekturze .NET.

5 Sterownik ODRA dla Środowiska .NET (ODRADriver)

Rozdział 5 opisuje interfejs sieciowy systemu ODRA oraz w jaki sposób jest on wykorzystany przez pierwszy element .Net Data Provider'a dla ODRA, który odpowiada za komunikację sieciową z DBMS. Tym elementem jest sterownik ODRA dla .NET (ODRADriver).

5.1 Budowa sterownika ODRA dla .NET

Sterownik ODRA dla .NET został zbudowany na podstawie implementacji jego odpowiednika w języku Java. Składa się z dwóch przestrzeni nazw, co ilustruje poniższy diagram:



Rys. 6.1.1 Schemat aplikacji wykorzystującej klienta ODRA dla .NET [Źródło: Opracowanie własne]

- Odra.Networking – klasy znajdujące się w tej przestrzeni nazw są odpowiedzialne za połączenie z bazą danych oraz przesyłanie żądań i rezultatów. Aplikacja klienta wykorzystuje te klasy do uzyskania połączenia oraz wysyłania żądań
- Odra.Results – klasy znajdujące się w tej przestrzeni nazw służą do odkodowywania rezultatów otrzymanych z bazy danych oraz opakowania ich w odpowiednie typy rezultatów. Obiekty tej klasy wykorzystywane są zarówno

przez klasy z przestrzeni Odra.Networking (do odkodowywania rezultatów otrzymanych z bazy) oraz przez aplikację klienta (do dalszego przetwarzania otrzymanych rezultatów)

5.2 Protokół sieciowy

System ODRA udostępnia interfejs sieciowy, na bazie którego można tworzyć mechanizmy komunikacyjne z bazami danych w ODRA, a w szczególności interfejsy dostępu do danych takie, jak JDBC, czy ADO.NET provider (jak w przypadku niniejszej pracy).

Połączenie z systemem ODRA można uzyskać poprzez gniazdo TCP/IP. Żądania wysyłane do systemu ODRA i otrzymywane rezultaty przekazywane są w postaci ciągu (tablicy) bajtów. Dalej przedstawione są formaty żądania i odpowiedzi wraz z opisem poszczególnych elementów.

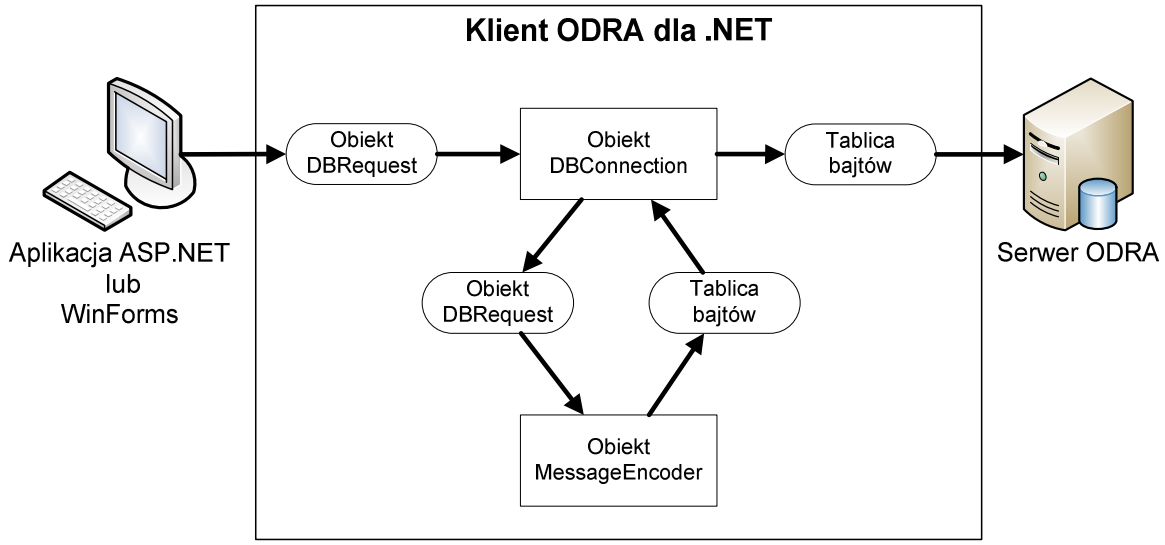
Format żądania jest następujący:

'RQST' (4 bajty)	# żądań (4 bajty)	id żądania (1 bajt)	# parametrów (4 bajty)	długość parametru (4 bajty)	parametr (n bajtów)	długość (4 bajty)	parametr (n bajtów)	...	id żądania (4 bajty)	...

Tabela 6.2.1 Format żądania do systemu ODRA [Źródło: Opracowanie własne]

- Pierwszym polem żądania – 4 bajty reprezentujące ciąg znaków *RQST* – jest nagłówek wiadomości
- Pole *# żądań* informuje ile żądań użytkownik przesyła za jednym razem
  Dalej zakodowane są żądania
  - *id żądania* to jego kod – oznacza on operację, jaką użytkownik chce wykonać (np. kod 4 – LOGIN _RQST – oznacza żądanie uwierzytelnienia)
  - Pole *# parametrów* określa liczbę parametrów żądania.
    Dalej zakodowane są parametry
    - Pole *długość parametru* oznacza jakiej długości będzie następne pole, czyli parametr
    - Pole *parametr* to treść parametru zakodowana przy użyciu kodowania UTF-8

Poniższy diagram prezentuje w jaki sposób tworzone i wysyłane jest żądanie do systemu ODRA za pomocą klienta dla .NET:



Rys. 6.2.1 Wysyłanie żądania do serwera ODRA [Źródło: Opracowani własne]

Format odpowiedzi otrzymywanej od serwera jest następujący:

'RPLY' (4 bajty)	# odpowiedzi (4 bajty)	status odpowiedzi (1 bajt)	długość rezultatu (4 bajty)	rezultat (n bajtów)	długość komunikatu (4 bajty)	komunikat (n bajtów)	...

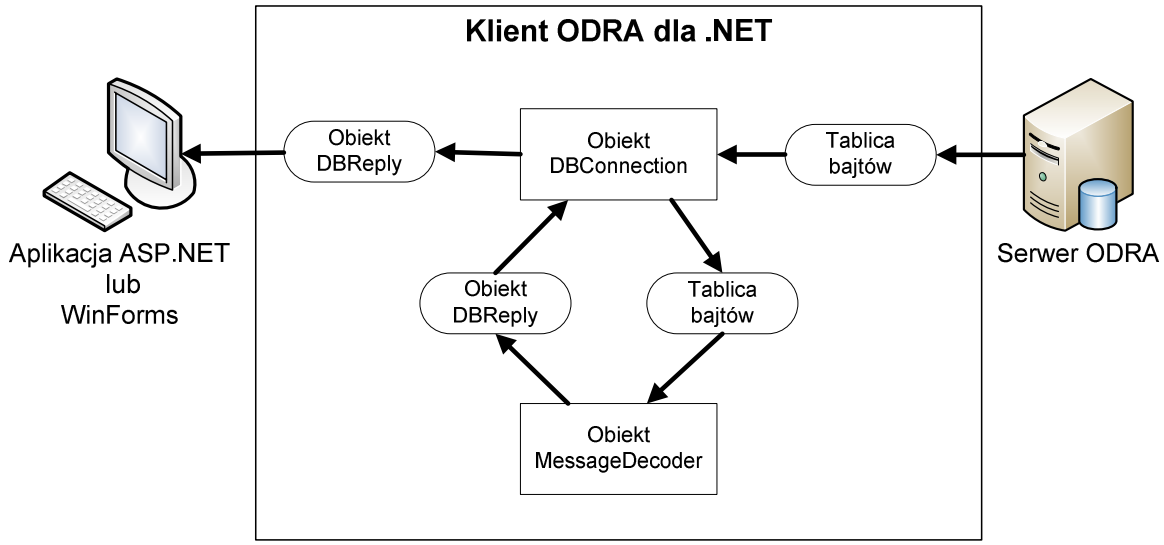
Tabela 6.2.2 Format odpowiedzi od systemu ODRA [Źródło: Opracowanie własne]

- Pierwszym polem żądania – 4 bajty reprezentujące ciąg znaków *RPLY* – jest nagłówek wiadomości
- Pole *# odpowiedzi* informuje ile odpowiedzi przyszło od serwera na zadane żądania

  Dalej zakodowane są odpowiedzi na żądania wysłane do serwera:
  - Pole *status odpowiedzi* określa status odpowiedzi, czyli czy z wiadomością wszystko w porządku, czy może wystąpił jakiś błąd w przesłanym wcześniej żądaniu
  - Pole *długość rezultatu* określa jaki długi będzie rezultat, czyli ile kolejnych bajtów składa się na odpowiedź serwera
  - Pole *rezultat* to jest zakodowany rezultat na żądanie wysłane do serwera
  - Pole *długość komunikatu* określa jak długi jest komunikat o błędzie otrzymanym od serwera

- Pole *komunikat* to ciąg bajtów reprezentujący opis błędu zakodowany w UTF-8

Poniższy diagram prezentuje jak przebiega proces odbierania odpowiedzi od serwera ODRA:



Rys. 6.2.2 Odbieranie odpowiedzi od serwera ODRA [Źródło: Opracowani własne]

Jak widać z powyższych diagramów podstawowymi klasami biorącymi udział w komunikacji klient-serwer są:

```
DBConnection // łączy się z bazą oraz wysyła wiadomości w postaci ciągów bajtów
DBRequest // obiekty tej klasy stanowią wrapper żądań dla aplikacji klienta
DBReply // obiekty tej klasy stanowią wrapper odpowiedzi dla aplikacji klienta
MessageEncoder // zakodowuje obiekty DBRequest do opisanego formatu żądania
MessageDecoder // odkodowuje ciąg bajtów w opisanym formacie odpowiedzi i opakowuje w obiekty klasy DBReply
```

5.3 Obsługa rezultatów

Na przedstawionym w tabeli 6.2.2 formacie odpowiedzi widać, że otrzymany rezultat jest nadal w postaci nieczytelnej dla programisty – jest po prostu ciągiem bajtów, dlatego w

przestrzeni nazw Odra.Results znajduje się klasa QueryResultDecoder, która ma za zadanie odkodować ten rezultat i opakować go w klasy reprezentujące rezultaty zwracane przez system ODRA.

Poniżej jest lista konkretnych rezultatów, jakie może otrzymać klient w odpowiedzi na swoje zapytanie:

Kolekcja:

`BagResult // jest to jedyny rodzaj kolekcji, który można otrzymać w odpowiedzi na zapytanie do serwera ODRA`

Pojedynczy rezultat (klasy dziedziczące z `SingleResult`):

`RemoteReferenceResult // jeśli programista otrzymał rezultat tego typu oznacza, że wynikiem (lub częściowym wynikiem) zapytania jest referencja do obiektu. Z tego rezultatu można wyciągnąć wewnętrzny OID (Object ID) obiektu zlokalizowanym na serwerze ODRA`

`BinderResult // rezultat będący wiązaniem nazwy do obiektu`

W poniższych rezultatach same nazwy wskazują na tym zwracanych danych, dlatego pominięte zostały ich dokładniejsze opisy:

`BooleanResult`

`DateResult`

`DoubleResult`

`IntegerResult`

`StringResult`

Tabela 6.3.1 Typy rezultatów zwracanych w odpowiedzi na zapytania [Źródło: Opracowanie własne]

Po odkodowaniu rezultatu z otrzymanego wcześniej ciągu bajtów użytkownik może go w dowolny sposób skonsumować – np. wypisać go na stronę w sieci Web.

5.4 Sposób użycia sterownika ODRADriver

Zanim programista będzie wysyłał zapytania do bazy danych najpierw musi uzyskać połączenie i zalogować się do systemu ODRA. Dokonać tego może w poniższy sposób:

```
DBConnection db = new DBConnection(server, port, timeout);
db.Connect();
DBRequest req = new DBRequest(DBRequest.LOGIN_RQST, new string[] { user, password });
```

```
DBReply rep = db.SendRequest(req);
```

Tabela 6.4.1 Uzyskanie połączenia z serwerem ODRA [Źródło: Opracowanie własne]

Warto zwrócić uwagę na sposób konstrukcji żądania (`DBRequest`). Wszystkie żądania do systemu ODRA konstruuje się w analogiczny sposób:

- jako pierwszy parametr w konstruktorze żądania zawsze należy podać typ żądania (w tym przypadku `DBRequest.LOGIN_RQST`)
- kolejnym parametrem jest tablica typu `string`, która zawiera parametry żądania. W zależności od typu żądania liczba parametrów różni się, a podanie niepoprawnej ilości lub formatu parametrów zwróci w odpowiedzi błąd systemu ODRA, a klient .NET wywoła wyjątek, typu `OdraException`, który powinniśmy obsłużyć.

Jeżeli w bazie nie ma jeszcze żadnego modułu zanim zacznie się korzystanie z bazy i zadawanie zapytań trzeba taki moduł stworzyć (typ żądania to `DBRequest.ADD_MODULE_RQST`). Poniższy kod tworzy, a później kompiluje nowy moduł.:

```
DBRequest req = new DBRequest(DBRequest.ADD_MODULE_RQST, new String[] {
module_code, currmod });
DBReply rpl = db.SendRequest(req);
req = new DBRequest(DBRequest.COMPILE_RQST, new String[] { currmod + "." +
moduleName });
rep = db.SendRequest(req);
```

Tabela 6.4.2 Tworzenie i kompilowanie modułu [Źródło: Opracowanie własne]

Jeżeli programista nie jest pewien czy moduł już sinieje może to sprawdzić sprawdzić wysyłając żądanie typu `DBRequest.EXISTS_RQST`, podając jaki obiekt chcemy sprawdzić (w tym wypadku „module”):

```
DBRequest req = new DBRequest(DBRequest.EXISTS_RQST, new String[] {
"module", module, currmod});
DBReply rpl = db.SendRequest(req);
bool exists = ((BooleanResult)rpl.GetResult()).value;
```

Tabela 6.4.3 Sprawdzanie, czy moduł istnieje [Źródło: Opracowanie własne]

Parametr `currmod` w powyższych zapytaniach to nazwa modułu, który jest aktualnie używany przez programistę. Po uzyskaniu połączenia moduł to nazwa użytkownika (np. *admin*). Aby zmienić używany moduł należy wysłać żądanie typu `DBRequest.EXISTS_MODULE_RQST` (powinno się zapamiętać nazwę nowego modułu, aby podawać ją przy kolejnych żądaniach jako `currmod`):

```
DBRequest req = new DBRequest(DBRequest.EXISTS_MODULE_RQST, new String[] {
currmod + "." + newModule });
DBReply rep = db.SendRequest(req);
```

Tabela 6.4.4 Zmiana aktualnego modułu [Źródło: Opracowanie własne]

Stworzony wcześniej moduł, lub jakikolwiek istniejący moduł stworzony wcześniej przy użyciu tego czy innego klienta, można usunąć przy pomocy żądania typu `DBRequest.REMOVE_MODULE_RQST`:

```
DBRequest req = new DBRequest(DBRequest.REMOVE_MODULE_RQST, new String[] {
module});
DBReply rpl = db.SendRequest(req);
```

Tabela 6.4.5 Usunięcie modułu [Źródło: Opracowanie własne]

Po stworzeniu modułu (lub zmianę na istniejący) można zacząć zadawać zapytania w składni SBQL i obsłużyć otrzymany rezultat.

```
DBRequest req = new DBRequest(DBRequest.EXECUTE_SBQL_RQST, new string[] {
sbql, currmod, "on", "off" }); // automatyczna dereferencja zapytań "on"
                               // wyniki w postaci XML "off"
DBReply rep = db.SendRequest(req);
Result res = rep.GetResult(); // uzyskanie rezultatu z obiektu DBReply
```

Tabela 6.4.6 Zadawanie zapytania w SBQL [Źródło: Opracowanie własne]

Jak już wcześniej wspomniano wywołanie `db.SendRequest(req);` może spowodować błąd na serwerze ODRA, a co za tym idzie zwrócić wyjątek typu `OdraException`. Wyjątek ten może informować o następujących błedach (klasa posiada atrybut `Type` typu enumeracyjnego `ErrorType`, a poniższa lista przedstawia możliwe wartości tego atrybutu):

- Błąd bazy danych (`DBError`)
- Wewnętrzny błąd systemu ODRA (`InternalError`)

- Błąd bezpieczeństwa (`SecurityError`)
- Błąd czasu wykonania (`RuntimeError`)
- Błąd sieci (`NetworkError`)
- Błąd kompilacji (`CompilationError`)
- Błąd optymalizacji (`OptimizationError`)
- Błąd nakładki (`WrapperError`)
- Błąd metabazy (`StaleMetabaseError`)
- Niepoprawna odpowiedź (`InvalidReply`)

6 Omówienie ODRAClient

6.1 Budowa i działanie ODRAClient

Opracowany w ramach niniejszej pracy ODRAClient jest elementem wyższego poziomu dostosowującym obiekty udostępniane przez ODRADriver do standardów obowiązujących w ADO.NET. Celem implementacji ODRAClient było umożliwienie korzystania z systemu ODRA z poziomu platformy .NET w sposób analogiczny do sposobu w jaki programiści posługujący się .NET korzystają z innych źródeł danych jak Microsoft SQL Server czy Oracle. Aby to osiągnąć zostały zaimplementowane interfejsy IDBConnection, IDBCommand, IDataParameter oraz IDataParameterCollection. Zaimplementowanie wymienionych interfejsów pozwoliło osiągnąć identyczny kod dostępu do danych jak w przypadku komercyjnych Data Providers oraz możliwość wykorzystywania mechanizmów Data Binding.

Projekt ODRAClient składa się z następujących klas:

- ODRAConnection – klasa implementująca interfejs IDbConnection a konkretnie następujące jej atrybuty i metody:
    - atrybut ConnectionString – jest to zmienna przechowująca dane konfiguracyjne połączenia z konkretną instancją bazy danych ODRA w postaci tekstu (typ .NET System.String) zawierającego zbiór powiązań klucz = wartość. Na potrzeby projektu ODRAClient opracowano własny zestaw kluczy definiujący namiary na bazę danych, należą do nich:
        - Server – definiuje adres w formacie URI stacji roboczej na której uruchomiony jest proces ODRA.
        - Database – definiuje nazwę konkretnej bazy danych, z której programista chce skorzystać w aplikacji .NET
        - Port – numer portu, na którym proces systemu ODRA nasłuchuje w oczekiwaniu na połączenia klientów
        - User – definiuje nazwę użytkownika
        - Password – definiuje hasło

- atrybut State - definiuje stan połączenia za pomocą wartości z enumeracji ConnectionState. ODRAClient korzysta z dwóch wartości z ConnectionState: Open i Close.
- Metoda Open() – obsługuje proces ustanawiania połączenia z bazą danych. Parsuje dane zawarte w ConnectionString i zgodnie z nimi stara się ustanowić połączenie za pomocą wewnętrznego obiektu DBConnection należącego do projektu ODRADriver.
- Metoda Close() – zamyka otwarte połączenie do bazy danych ODRA
- Metoda CreateCommand() – tworzy i zwraca nowy obiekt typu ODRACommand
- Metoda Dispose() – Metoda interfejsu IDisposable szczególnie ważna w przypadku obiektów reprezentujących połączenia z bazą danych gdyż powinny być one możliwie szybko zamykane i zwalniane. Metoda ta zamyka połączenie w przypadku gdy jest aktywne.

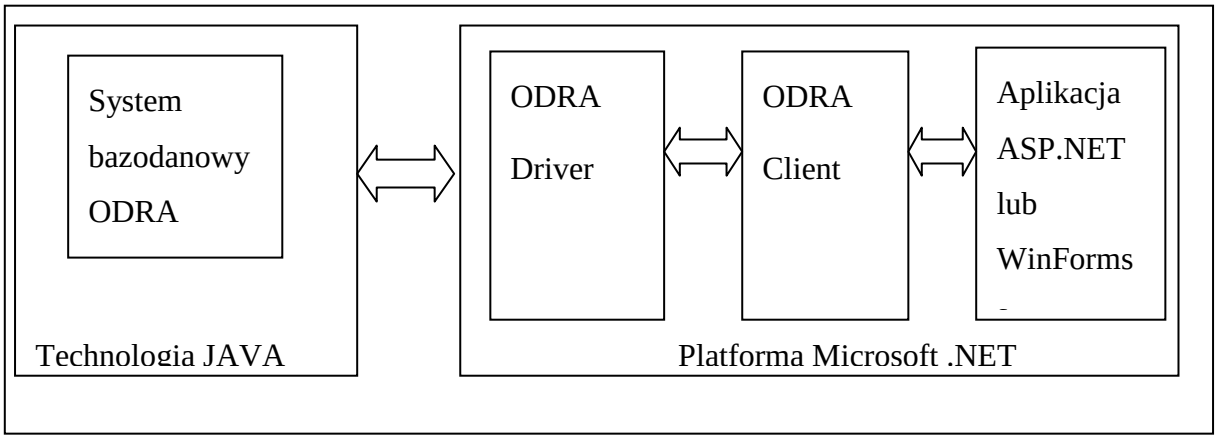
ODRAConnection implementuje także dwie metody niezawarte w interfejsie IDBConnection a wprowadzone w celu umożliwienia interakcji z bazą danych ODRA:

- ExecSBQL() – pobiera jako parametr zapytanie SBQL i zwraca obiekt klasy Result zdefiniowanej w projekcie ODRADriver który reprezentuje wynik zapytania SBQL. Metoda korzysta z niskopoziomowych klas ODRADriver: DBRequest, DBReplay oraz QueryResultDecoder.
- ExecSBQLReturnResult() – jest to najważniejsza metoda projektu ODRAClient ze względu na dostosowanie interakcji z ODRĄ do standardów ADO.NET. Metoda ta podobnie do ExecSBQL() pobiera jako parametr zapytanie SBQL i korzysta z niskopoziomowych obiektów ODRADriver w celu realizacji zapytania. Metoda przegląda zwrócone z systemu ODRA dane i zapełnia nimi obiekt ODRAResultSet a konkretnie zawartą w nim wielowymiarową tablicę, dzięki czemu obiektowe dane zostają „spłaszczone” i dostosowane do pobierania przez obiekty ADO.NET, przede wszystkim przez obiekt DataReader. Metoda wypełnia obiekt ODRAResultSet również metadanymi zawierającymi nazwy atrybutów.

- ODRACommand – jest to obiekt reprezentujący zapytanie SBQL powiązane z konkretnym połączeniem (obiekt ODRAConnection). ODRACommand implementuje m.in. następujące atrybuty i metody:
    - CommandText – atrybut przechowujący tekst zapytania/polecenia SBQL które ODRACommand reprezentuje. Zapytanie przechowywane jest w postaci tekstu i nie jest poddawane żadnej obróbce poza ewentualnym dołączeniem parametrów. Tekst zapytania przekazywany jest bezpośrednio do systemu ODRA, co odpowiada technice „zanurzonego SQLa" (w tym przypadku można mówić o „zanurzonym SBQLu").
    - Connection – atrybut przechowywujący obiekt ODRAConnection, który zostanie użyty do nawiązania połączenia z bazą ODRA i wykonania zapytania.
    - ExecuteNonQuery() – metoda wywołuje zapytania SBQL bez zwracania wyników, przez co jest odpowiednia przede wszystkim do wydawania bazie danych pojedynczych poleceń.
    - ExecuteScalar() – metoda wywołuje zapytanie SBQL i zwraca pojedynczy rezultat lub pierwszy rezultat z serii rezultatów.
    - ExecuteReader() – kluczowa metoda dla mechanizmu DataBinding (jej wynik jest często przypisywany do atrybutu DataSource kontrolek obsługujących wyświetlanie danych takich jak DataGrid/GridView). Metoda wywołuje metodę ExecSbqlReturnResultSet na skojarzonym obiekcie Connection a uzyskany z niej obiekt ODRAResultSet, reprezentujący rezultat zapytania, przekazuje do konstruktora obiektu DataReader, który jest następnie zwracany. Metoda posiada również wariant pobierający jako argument zmienną typu CommandBehavior definiujący czy skojarzony obiekt Connection powinien być zamykany po wykonaniu zapytania.
    - PrepareParameters() – metoda wywoływana przed wysłaniem zapytania SBQL do bazy danych ODRA. Służy do scalenia wartości parametrów z kolekcji ODRAParameterCollection z tekstem samego zapytania.
- ODRAParameter – obiekt reprezentujący parametr w przypadku zapytań, które wymagają sparametryzowania danymi pobranymi np. z interfejsu użytkownika. Przed wykonaniem zapytania parametry są scalane z tekstem zapytania za pomocą prostych operacji na typie System.String. Budowanie sparametryzowanych zapytań w

aplikacjach ASP.NET i Windows Forms jest zalecaną praktyką ze względu na odporność na ataki SQL Injection dzięki sprawdzaniu poprawności parametrów (co nie ma miejsca w przypadku gdy programista tworząc zapytanie po prostu dokonuje konkatenacji). Z tego powodu zaimplementowanie obsługi parametrów dla zapytań w SBQL jest ważne dla programistów ASP.NET.

- ODRAParameterCollection – obiekt reprezentujący kolekcję parametrów ODRAParameter potrzebnych do wykonania danego zapytania SBQL.
- ODRAResultSet – obiekt reprezentujący zbiór rezultatów zwróconych przez ODRACommand. Zawiera wewnętrzną klasę MetaData, której tablica zawiera nazwy atrybutów/kolumn zwróconych przez zapytanie SBQL oraz dwuwymiarową tablicę wyników przechowującą wartości wszystkich atrybutów wszystkich zwróconych obiektów. Tablica wyników tworzona jest w metodzie ExecSbqlReturnResultset obiektu ODRAConnection i ma podstawowe znaczenie dla dostosowania obiektowego modelu systemu ODRA do zorientowanych relacyjnie konstrukcji ADO.NET
- ODRAException – obiekt reprezentujący błąd przetwarzania w obrębie ODRADataProvider. Dostawcy DataProviders dla konkretnych baz danych definiują własną klasę reprezentującą błąd, często dodatkowo definiując pola reprezentujące wewnętrzne numery błędów w źródle danych (przykładowo dla bazy danych SQL Server została zdefiniowana klasa SqlException).



Rys. 6.1.1 Schemat komunikacji ODRA - aplikacja .NET i ODRAClient [Źródło: Opracowanie własne]

6.2 Wykorzystanie ODRAClient z poziomu aplikacji .NET

Aplikacje .NET, korzystające z systemu bazodanowego ODRA za pomocą ODRAClient, korzystają z identycznego schematu działania jak w przypadku korzystania z innych Data Provider dla ADO.NET. Kolejne kroki interakcji ze źródłem danych zgodne są z krokami przedstawionymi w sekcji 4.5.8. W praktyce kod korzystający z ODRAClient ma w języku C#.NET następującą postać:

```
using (ODRAConnection conn = new ODRAConnection
  ("server=localhost;database=admin.phonebook;port=1521;
  user=admin;password=admin"))
{
  ODRACommand cmd = new ODRACommand("Person where name =
  @name;", conn);
     cmd.Parameters.Add(new ODRAParameter("@name",
"Ludwik"));
  conn.Open();
     ODRADataReader reader =
  (ODRADataReader)cmd.ExecuteReader();
     repeaterPerson.DataSource = reader;
     repeaterPerson.DataBind();
}
```

Przedstawiony kod wykonuje następujące operacje:

1. Utworzenie obiektu ODRAConnection o nazwie conn zainicjalizowanego parametrami połączenia z konkretną instancją bazy danych ODRA. Obiekt tworzony jest w specjalnej konstrukcji języka C#.NET using, która gwarantuje, że po wyjściu z zakresu oznaczonego nawiasami klamrowymi na obiekcie conn zostanie wywołana metoda Dispose przygotowująca obiekt do wykasowania z pamięci. W przypadku obiektów Command Data Providerów dla ADO.NET, metoda Dispose wywołuje metodę Close() w celu zamknięcia aktywnego połączenia ze źródłem danych (o ile takie ma miejsce). Konstrukcja using gwarantuje wywołanie Dispose() (a tym samym zamknięcie połączenia) nawet w przypadku zgłoszenia wyjątku i przerwania

normalnego wykonania programu. Ponieważ użycie using jest zalecaną przez Microsoft praktyką pisania wydajnych/skalowalnych aplikacji, zdecydowano się umożliwić jej wykorzystanie we współpracy z bazą danych ODRA.

2. Otwarcie połączenia. Obiekt conn korzysta z obiektu niższego poziomu (projekt ODRADriver), który za pomocą gniazd łączy się z instancją ODRA.
3. Utworzenie obiektu ODRACommand reprezentującego polecenie/zapytanie SBQL i zainicjalizowanie go tekstem zapytania SBQL oraz obiektem typu ODRAConnection, które zostanie wykorzystane podczas łączenia się z instancją ODRA.
4. Utworzenie obiektu ODRAParameter reprezentującego parametr podany w tekście zapytania. Podanie nazwy i wartości parametru.
5. Otwarcie połączenia do instancji bazy danych ODRA. Wywoływana metoda Open() wykorzystuje zawarty w klasie ODRAConnection obiekt typu DBConnection (projekt ODRADriver) w celu nawiązania niskopoziomowego połączenia.
6. Wywołanie metody ExecuteReader(), która przekazuje tekst zapytania SBQL do instancji bazy danych ODRA gdzie następuje jego wykonanie. Wyniki zwracane są do wywołującego w postaci obiektu ODRADataReader, który obsługuje przeglądanie zwróconych rezultatów w sposób sekwencyjny do przodu (forward only)
7. Obiekt reader przypisywany jest atrybutowi DataSource obietku repeater. Obiekt repeter (System.Web.UI.WebControls.Repeater) jest kontrolką wizualną ASP.NET obsługującą szablonowe wyświetlanie zbioru danych (wartość każdej danej ze zbioru scalana jest z szablonem i dopisywana do treści strony w HTML).
8. W chwili zamknięcia nawiasu klamrowego określającego zakres działania konstrukcji using, wywoływana jest metoda ODRAConnection.Dispose(), która z kolei, poprzez wywołanie metody Close(), zamyka połączenie z instancją bazy danych ODRA.

Dla porównania, analogiczny kod posługujący się Data Provider dla Microsoft SQL Server wygląda następująco:

```
   using (SqlConnection conn2 = new SqlConnection
("Server=localhost; Initial Catalog=phonebook;User
Id=admin;Password=admin;"))
   {
     conn.Open();
        SqlCommand cmd2 = new SqlCommand("select * from Person
     where name=@name", conn2);
```

```
        cmd2.Parameters.Add( new SqlParameter("@name",
"Ludwik"));
        SqlDataReader reader = cmd2.ExecuteReader();
        repeaterPerson.DataSource = reader;
        repeaterPerson.DataBind();
    }
```

Jak widać kod aplikacji .NET korzystającej z ODRAClient nie różni się znacząco od kodu SqlClient dostarczanego razem z platformą .NET. Dzięki temu programista nie powinien mieć trudności w obsługiwaniu danych pochodzących z systemu bazodanowego ODRA z poziomu aplikacji .NET.

7 Przykładowa aplikacja na bazie ODRAClient

Rozdział 7 przedstawia praktyczne zastosowanie ODRAClient na przykładzie prostej aplikacji ASP.NET – baza książek. Aplikacja ta przechowuje Autorów, Książki oraz Opinie o książkach. Jest to minimalistyczna implementacja mająca na celu jedynie zaprezentowanie co można osiągnąć dzięki użyciu ODRAClient w połączeniu z systemem ODRA. Aplikacja ta będzie dalej nazywana BookList.

7.1 Wymagania sysemowo-sprzętowe

Do uruchomienia aplikacji BookList niezbędne jest uruchomienie serwera ODRA na lokalnym (ewentualnie zdalnym) komputerze. Komputer ten musi zatem spełniać niezbędne wymagania do uruchomienia serwera ODRA, czyli mieć zainstalowane środowisko JRE w wersji 1.6+.

Serwer, na którym działać będzie aplikacjia BookList musi mieć zainstalowany system Windows XP Professional/Windows Server 2003 lub nowszy z IIS w wersji 6.0 lub wyższej.

7.2 Instalacja aplikacji

Na załączonej do pracy płycie CD znajdują się dwa katalogi:

- odra
- dotNetProviderForODRA

Żeby zainstalować serwer ODRA należy przegrać katalog *odra* do dowolnej lokalizacji na dysku twardym – np. „c:\odra”. Pierwszego uruchomienia najlepiej wykonać przy pomocy pliku wsadowego easyStart.bat. Później można uruchamiać już poprzez odraserver.bat.

Instalacja aplikacji ASP.NET wymaga przegrania katalogu *dotNetProviderForODRA* do dowolnego katalogu na dysku twardym (np. „c:\Intepub\wwwroot”) oraz ustawić IIS tak, żeby domyślna strona (Default Web Site) wskazywała na ten katalog.

7.3 Przygotowanie bazy danych

Aplikacja zawiera przygotowany wcześniej przykładowy schemat bazy danych wraz z niewielką ilością danych. Po zainstalowaniu aplikacji wystarczy wejść na http://localhost/install/Default.aspx i odpali się skrypt inicjalizujący bazę danych. Również w momencie, gdy użytkownik chce wrócić do pierwotnego stanu bazy wystarczy ponownie wejść na tą stronę. Skrypt tworzy poniższy moduł w systemie ODRA i uruchamia procedurę init(), która wypełnia bazę przykładowymi danymi:

```
module BookStore{

     class  PersonClass {
          instance Person : {
             fName : string;
             lName : string;
          }

          getFullName():string{
             return fName + " " + lName;
          }
     }

     class AuthorClass extends PersonClass{
          instance Author : {
               books : ref BookClass[0..*] reverse author;
          }
     }

     class BookClass {
          instance Book : {
               title : string;
               imgUrl : string[0..1];
               author : ref AuthorClass reverse books;
               reviews : ref ReviewClass[0..*] reverse book;
          }
```

```
    }

    class ReviewClass {
        instance Review : {
            title: string;
            content : string;
            book : ref BookClass reverse reviews;
            author : string;
        }
    }

    Person : PersonClass[0..*];
    Author : AuthorClass[0..*];
    Book : BookClass[0..*];
    Review : ReviewClass[0..*];

    init(){
        create permanent Author("Adam" as fName, "Mickiewicz" as
lName);
        create permanent Author("Julian" as fName, "Tuwim" as
lName);
        create permanent Author("Ignacy" as fName, "Krasicki" as
lName);
        create permanent Author("Denis" as fName, "Diderot" as
lName);


        create permanent Book("Sonety krymskie" as title,
"~/images/sk.jpg" as imgUrl, ref(Author where lName = "Mickiewicz")
as author);
        create permanent Book("Konrad Wallenrod" as title,
"~/images/kw.jpg" as imgUrl, ref(Author where lName = "Mickiewicz")
as author);


        create permanent Book("Czyhanie na Boga" as title,
ref(Author where lName = "Tuwim") as author);
        create permanent Book("Czarna msza" as title, ref(Author
where lName = "Tuwim") as author);
        create permanent Book("Czary i czarty polskie" as title,
ref(Author where lName = "Tuwim") as author);
```

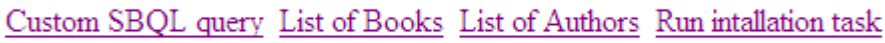
```
          create permanent Book("Monachomachia" as title,
"~/images/mn.jpg" as imgUrl, ref(Author where lName = "Krasicki") as
author);
          create permanent Book("Antymonachomachia" as title,
ref(Author where lName = "Krasicki") as author);


          create permanent Book("Kubus Fatalista i jego pan" as
title, "~/images/kb.jpg" as imgUrl, ref(Author where lName =
"Diderot") as author);
          create permanent Book("Kuzynek mistrza Rameau" as title,
ref(Author where lName = "Diderot") as author);


          create permanent Review("Kiepska" as title, "Tej ksiazki
nie da sie przeczytac!!" as content, "parak" as author, ref(Book
where title = "Czyhanie na Boga") as book);
          create permanent Review("WOW!!!" as title, "Super, w
ogole odjazd!!" as content, "dinek" as author, ref(Book where title
= "Konrad Wallenrod") as book);
          create permanent Review("Poruszajacy temat" as title,
"Czytalo sie naprawde fajnie." as content, "pietrek" as author,
ref(Book where title = "Kuba Fatalista i jego pan") as book);


          create permanent Review("Ekstra" as title, "Naprawde
niezly stuff" as content, "kulka" as author, ref(Book where title =
"Monachomachia") as book);
          create permanent Review("Straszne nudy..." as title,
"Zasnalem przy drugim rozdziale, fatal!" as content, "kulka" as
author, ref(Book where title = "Sonety krymskie") as book);
          create permanent Review("Intrygujaca!" as title, "Z
zapartym tchem parlem do konca ksiazki." as content, "milew" as
author, ref(Book where title = "Czarna msza") as book);
          create permanent Review("Intrygujaca!" as title, "Z
zapartym tchem parlem do konca ksiazki." as content, "grzybek" as
author, ref(Book where title = "Antymonachomachia") as book);
     }
}
```

Tabela 7.3.1 Przykładowe dane dla aplikacji BookList [Źródło: Opracowanie własne]

Powyższy schemat jest stworzony dla prostej bazy danych zawierającej informacje o Książkach, Autorach oraz Recenzjach.

7.4 Praca z aplikacją BookList

Po zainicjalizowaniu aplikacji można przejść do testowania aplikacji oraz zadawania własnych zapytań SBQL do systemu ODRA. Poniższy rysunek prezentuje menu nawigacyjne po aplikacji BookList:



Rys. 7.4.1 Nawigacja aplikacji BookList [Źródło: Opracowanie własne]

Powyższe odnośniki umożliwiają:

- Custom SBQL query – zadawanie własnych zapytań w SBQL
- List of Book – zarządzanie książkami (dodawanie, edycja, usuwanie) oraz dodawanie recenzji
- List of Authors – zarządzanie autorami (dodawanie, edycja, usuwanie)
- Run installation task – uruchomienie skryptu instalacyjnego (UWAGA: powoduje zresetowanie bazy danych i zainicjalizowanie danymi przykładowymi)

7.5 Realizacja operacji CRUD przy użyciu ODRAClient oraz SBQL

7.5.1 Wewnętrzyny identyfikator obiektów

Do identyfikacji obiektów aplikacja używa wewnętrznego identyfikatora bazy ODRA uzyskiwanego poprzez `serialize x`, gdzie x jest referencją do obiektu, którego identyfikator chcemy odzyskać. Identyfikator uzyskany w ten sposób może posłużyć do przypisywania referencji, żeby uniknąć porównywania wszystkich pól obiektów (co i tak jest nie gwarantuje, że uzyskamy planowany cel, ponieważ może być więcej niż 1 obiekt o takich samych wartościoch pól i wtedy użytkownik nie ma kontroli nad tym, co przypisuje).

7.5.2 Selekcja (READ)

Selekcja zostanie zilustrowana na dwóch przykładach:

1. Selekcja informacji o książkach:

```
(Book as x).((serialize x) as oid, x.title as title, ((x.imgUrl as imgUrl)
union (\"\" as imgUrl where not exists(x.imgUrl))), (serialize x.author)
```

```
as authorId, x.author.getFullName() as author);
```

Tabela 7.5.2.1 Zapytanie wyciągające informacje o książkach z bazy danych [Źródło: Opracowanie własne]

Warto zauważyć dwie interesujące metody w powyższym zapytaniu. Po pierwsze jak radzi sobie z ewentualnym brakiem wartości (w opozycji do wartości zerowych *NULL* w relacyjnych systemach baz danych) w atrybucie *imgUrl* za pomocą wyrażeń *union* oraz *exists*. Kolejną rzeczą jest korzystanie z metody *getFullName()* zdefiniowanej w type *PersonType*. Prezentuje to w jaki sposób można odwoływać się do metod obiektów w zapytaniach.

W tym zapytaniu zwracane obiekty będą typu *binder* (co uzyskuje się przy pomocy nadania nazwy – *as nazwa*), aby można było intuicyjnie wykorzystywać nazwy zwracanych pól w kodzie strony. Dla powyższego zapytania wykorzystane jest to w poniższy sposób:

```
<uc:Book BookID='<%# Eval("oid") %>' AuthorID='<%# Eval("authorId") %>'
Title='<%# Eval("title") %>' ImgUrl='<%# Eval("imgUrl") %>' Author='<%#
Eval("author") %>' ID="Book1" runat="server" />
```

Tabela 7.5.2.2 Przykład intuicyjnego wykorzystania nazw pól [Źródło: Opracowanie własne]

2. Selekcja informacji o recenzjach wystawionych dla danej książki (identyfikacja na postawie wewnętrznego identyfikaora)

```
deref(((Book as x).((serialize x) as oid, x.reviews as reviews) where
(string)oid = \"" + bookId + "\").reviews);
```

Tabela 7.5.2.3 Zapytanie SBQL wybierające recenzje dla danej książki [Źródło: Opracowanie własne]

W powyższym zapytaniu parametr *bookId* to wewnętrzny identyfikator książki, której recenzje chcemy pobrać. Widać w nim również, jak wygodne jest zastosowanie wyrażeń ścieżkowych do wykonywania złączeń (*join*).

Kolejny przykład zilustruje wykorzystanie wyniku przy użyciu ODRAClient na przykładzie selekcji autorów.

```
protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            rptAuthors.DataSource = GetAuthors();
            rptAuthors.DataBind();
        }
    }

protected IDataReader GetAuthors()
```

```
    {
        using (ODRAConnection conn = new ODRAConnection
("server=localhost;database=admin.BookStore;port=1521;user=admin;password=a
dmin;timeout=1000"))
        {
            try
            {
                conn.Open();
                string sbql = "(Author as x).( (serialize x) as oid,
x.fName as fName, x.lName as lName);";

                ODRACommand cmd = new ODRACommand(sbql, conn);

                return cmd.ExecuteReader();
            }
            catch (ODRAException exc)
            {
                Response.Write("Error: " + exc.Message.Replace("\n", "<br
/>"));
                return null;
            }
        }
    }
```

Tabela 7.5.2.4 Wykorzystanie ODRAClient do selekcji informacji o autorach

[Źródło: Opracowanie własne]

W przypadku selekcji wywołujemy metodę `cmd.ExecuteReader();`, która zwraca obiekt typu `IDataReader`. Ten obiekt można później bezpośrednio podpiąć jako źródło danych (DataSource) do kontrolki ASP.NET, która ma taką możliwość (np. Repeater, GridView, List, DropDownList, itd.)

7.5.3 Wstawianie (CREATE lub INSERT)

Tworzenie nowych trwałych obiektów w systemie odra jest bardzo łatwe. W tym wypadku również przedstawimy kilka przykładów ilustrujących różne możliwości tworzenia (lub tworzenia i wstawiania) prostych oraz bardziej złożonych obiektów, które zawierają referencje do innych obiektów:

1. Tworzenie prostego obiektu (autora):

```
create permanent Author(\"" + fName + "\" as fName, \"" + lName + "\" as
lName);
```

Tabela 7.5.3.1 Zapytanie tworzące nowy obiekt autora [Źródło: Opracowanie własne]

Warto zaznaczyć, że obiekt typu *AuthorType* zawiera listę pointerów do obiektów książek, które autor napisał.

2. Wstawienie nowej książki do obiektu autora:

```
((Author as x).( (serialize x) as oid, x as object) where
(string)oid = \" + oid + \").object :<< books( ref(create permanent
Book(\" + title + \" as title, ref( ((Author as x).( (serialize x)
as oid, x as object) where (string)oid=\" + oid + \").object ) as
author) ));
```

Tabela 7.5.3.2 Zapytanie wstawiające nową książkę do obiektu autora [Źródło: Opracowanie własne]

Powyższe zapytanie wykorzystuje operator wstawiania *:<<* do dodania stworzenia nowej książki i dodania jej do autora. Poniważ jednak typ *BookType* oraz *AuthorType* mają zdefiniowane dwukierunkowe pointery (bidirectional pointers) wystarczy stworzyć nową książkę dodając jej pointer do autora, a automatycznie zostanie stworzony odpowiadający tej książce pointer w obiekcie reprezentującym tego autora. Tę metodę przedstawia kolejny przykład

3. Tworzenie nowej książki:

```
create permanent Book(\" + title + \" as title, ref( ((Author as
x).( (serialize x) as oid, x as object) where (string)oid=\" + oid +
\").object ) as author);
```

Tabela 7.5.3.3 Zapytanie tworzące nową książkę [Źródło: Opracowanie własne]

Wyrażenie *create* zwraca referencję do stworzonego obiektu i ten wynik można skosumować w dowolny sposób (np. taki, jak w Tabeli 7.5.2.4), ale jeśli wynik ten nie będzie nam potrzebny można po prostu wywołać metodę `cmd.ExecuteScalar();` obiektu `ODRACommand`.

```
using (ODRAConnection conn = new ODRAConnection
("server=localhost;database=admin.BookStore;port=1521;user=admin;password=a
dmin;timeout=1000"))
{
    try
    {
        conn.Open();
        string sbql = "create permanent Review(\"" + txtReviewTitle.Text +
            "\" as title, \"" + txtReviewContent.Text + "\" as content, \""
+ txtReviewAuthor.Text +
            "\" as author, ref(( (Book as x).( (serialize x) as oid, x as
object) where (string)oid = \"" + BookID + "\").object) as book);";

        ODRACommand cmd = new ODRACommand(sbql, conn);
        cmd.ExecuteScalar();
        Response.Redirect(Request.Url.AbsolutePath);
```

```
    }
    catch (ODRAException exc)
    {
        Response.Write("Error: " + exc.Message.Replace("\n", "<br />"));
    }
}
```

Tabela 7.5.3.4 Wykorzystanie ODRAClient do stworzenia nowego obiektu [Źródło: Opracowanie własne]

7.5.4 Aktualizacja (UPDATE)

W SBQL dane aktualizuje się poprzez przypisywanie (*assignment*) realizowane wyrażeniem „*lQuery* := *rQuery*”. Ta operacja nie jest makroskopowa, dlatego zapytania muszą zwrócić pojedyncze wartości. Wynikiem lewego zapytania musi być referencją pod którą chcemy zapisać wynik prawego zapytania. Realizację aktualizacji przedstawia poniższy przykład:

```
"((Author as x).( (serialize x) as oid, x as object) where (string)oid =
\"" + authorID + "\").object.fName := \"" + firstName + "\";"
```

Tabela 7.5.4.1 Zapytanie aktualizujące obiekt autora

7.5.5 Usuwanie (DELETE)

Usuwanie obiektów w SBQL jest zrealizowane w bardzo wygodny sposób zostanie to pokazane na złożonym przykładzie usuwania autora (trzeba usunąć książki tego autora wraz z ich recenzjami).

```
try
{
    conn.Open();
    string sbql = "delete ((Author as x).( (serialize x) as oid, x as
object) where (string)oid = \"" + AuthorID +
"\").object.books.Book.reviews.Review;";

    ODRACommand cmd = new ODRACommand(sbql, conn);
    cmd.ExecuteScalar();

    sbql = "delete ((Author as x).( (serialize x) as oid, x as object)
where (string)oid = \"" + AuthorID + "\").object.books.Book;";
    cmd = new ODRACommand(sbql, conn);
    cmd.ExecuteScalar();

    sbql = "delete ((Author as x).( (serialize x) as oid, x as object)
where (string)oid = \"" + AuthorID + "\").object;";
    cmd = new ODRACommand(sbql, conn);
    cmd.ExecuteScalar();

    Response.Redirect(Request.Url.AbsolutePath);
```

```
}
catch (ODRAException exc)
{
    Response.Write("Error: " + exc.Message.Replace("\n", "<br />"));
}
```

Tabela 7.5.5.1 Usuwanie obiektu autora przy użyciu ODRAClient [Źródło: Opracowanie własne]

W powyższym przykładzie widać trzy kolejne zapytania, które wysyłane są do systemu ODRA. Pierwsze dwa służą do usunięcia Recenzji, a później książki. Dopiero po ich wykonaniu można usunąć obiekt autora.

W zapytaniach realizujących usuwanie Recenzji i Książek widać, jak wygodne w użyciu są zapytania ścieżkowe wykorzystujące dwukierunkowe pointery. Jest to bardzo intuicyjne podejście zbliżone do tego, w jaki sposób programiści nawigują po atrybutach złożonych obiektów.

8 Podsumowanie

Autorzy zrealizowali stawiany sobie cel i umożlwili wykorzystanie serwera ODRA do programowania aplikacji w środowisku .NET Framework (w szczególności ASP.NET) poprzez stworzenie wygodnego w użyciu sterownika dostępu do danych implementującego interfejs ADO.NET.

Na podstawie opracowanego rozwiązania ADO.NET ODRA Data Provider można stwierdzić, że programiści tworzący aplikacje dla platformy Microsoft.NET mogą bez problemu wykorzystywać, jako źródło danych, system bazodanowy ODRA. Wprawdzie zaimplementowany w ramach pracy ODRA Data Provider nie implementuje wszystkich mechanizmów ADO.NET Data Provider (jak np. transakcje), ale udowadnia, że współpraca systemu ODRA z platformą .NET jest w pełni możliwa zgodnie ze standardami ADO.NET. Przedstawione oprogramowanie składające się z projektu ODRADriver oraz ODRAClient jest fundamentem do dalszego rozwoju w kierunku integracji platformy .NET z bazą danych ODRA.

W chwili obecnej ODRAClient umożliwia zadawanie zapytań typu CRUD w języku SBQL. Przyszły rozwój interfejsu powinien uwzględnić obsługę zapytań administracyjnych (takich jak tworzenie klas, widoków, indeksów) oraz obsługę transakcji, jak umożliwiają to inni dostawcy np. SQLProvier. Wykonanie tych operacji jest możliwe za pomocą interfejsu ODRADriver, ale nie tak wygodne, jak w przypadku użycia ODRAClient.

Stworzenie możliwości korzystania z systemu ODRA w środowisku .NET Framework może przyczynić się popularyzacji podejścia stosowego wśród programistów, którzy programują w językach .NET.

9 Słownik terminów

- API – Application Programming Interface (interfejs programistyczny)
- DBMS – Database Management System (System Zarządzania Bazami Danych)
- JVM – Java Virtual Machine (Wirtualna Maszyna Javy) – środowisko, w którym wykonywane są programy napisane w języku Java
- COM – Component Object Model
- ODMG (Object Database Management Group) – organizacja, która za cel wzięła sobie opracowanie zbioru specyfikacji do tworzenia przenośnych aplikacji korzystających z obiektowych baz danych
- Operacje CRUD – create, read, update, delete

10 Źródła

10.1 Prace cytowane

1. **Subieta, Kazimierz.** *Teoria i konstrukcja obiektowych języków zapytań.* Warszawa : Wydawnictwo PJWSTK, 2004.

2. —. General Overview of SBA/SBQL. [Online] 2006. http://sbql.pl/overview/.

3. **Wikipedia.** *Call stack.* [Online] http://en.wikipedia.org/wiki/Call_stack .

4. **Kazimierz Subieta and the ODRA team.** ODRA Description and Programmer Manual. *ODRA Manual.* [Online] 2008. http://www.sbql.pl/various/ODRA/ODRA_manual.html.

10.2 Bibliografia

- Kazimierz Subieta. Teoria i konstrukcja obiektowych języków zapytań. Wydawnictwo PWSTK, Warszawa 2004.
- http://www.sbql.pl/various/ODRA/ODRA_manual.html
- http://www.sbql.pl/various/ODRA/ODRA_manual_Java.html
- http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch
- http://www.simba.com/odbc.htm
- http://en.wikipedia.org/wiki/ODBC
- Microsoft ODBC – http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx
- http://www.simba.com/jdbc.htm
- http://java.sun.com/products/jdbc/driverdesc.html
- http://msdn.microsoft.com

11 Dodatki

11.1 Dodatek A: System ODRA

Na załączonej płycie CD, w folderze „*odra*” znajduje się skompilowana wersja systemu ODRA wraz ze skryptami uruchamiającymi serwer w domyślnych ustawieniach.

11.2 Dodatek B: Kod źródłowy projektu

Kod źródłowy projektu .*Net Data Provider dla ODRA obsługujący zapytania sformułowane w SBQL* znajduje się w katalogu „dotNetDataProviderForODRA” na załączonej płycie CD.