STUDIA INFORMATICA 2006
Volume 27 Number 4 (69)

Jacek SZEDEL, Michał KOLANO, Jarosław CHOJNACKI
Politechnika Śląska, Instytut Informatyki

INTEGRACJA METODYKI OBIEKTOWEJ OPARTEJ NA ZASTOSOWANIU JĘZYKA UML ZE ŚRODOWISKIEM PROGRAMISTYCZNYM BORLAND C++ BUILDER

Streszczenie. W artykule podjęto próbę rozwiązania problemu integracji modelu obiektowego systemu informatycznego z częścią realizującą funkcje interfejsu dla relacyjnej bazy danych, a także częścią odpowiedzialną za obsługę graficznego interfejsu użytkownika. Opisywane rozwiązanie dotyczy platformy Borland C++ Builder. Opracowane zostały projekt i implementacja biblioteki komponentów nazwanej skrótem BMU (od ang. Business Model Utility).

Słowa kluczowe: analiza i projektowanie systemów informatycznych, metodyki obiektowe, język UML, metodyka RUP, programowanie obiektowe, zintegrowane środowiska programistyczne

INTEGRATION OF THE UML BASED MODELING METHODOLOGY WITH, THE BORLAND C++ BUILDER IDE

Summary. This article presents the description of the concepts, design and implementation of the component library created by the authors of the article for the Borland C++ IDE. The library enables programmer to separate the business functions from the database integration and the GUI functions.

Keywords: software analysis and design, object oriented methodologies, Unified Modelling Lanuage, Rational Unified Process, object oriented programming, integrated development environments

1. Wprowadzenie

1.1. Geneza problemu

Bardzo istotną niedogodnością zintegrowanych środowisk programistycznych, takich jak wymienione w tytule artykułu, środowisko C++ Builder firmy Borland Software Corporation[1], jest nadmierne znaczenie, jakie projektanci tych środowisk nadali zagadnieniom związanym z dostępem do relacyjnych baz danych. Dotyczy to zarówno komponentów integrujących aplikację z bazami danych, jak i komponentów realizujących obsługę odpowiednich elementów graficznego interfejsu użytkownika – komponentów, które umożliwiają użytkownikowi wizualną edycję danych. Sytuacja ta, niezależnie od zalet tego typu rozwiązań, ogranicza możliwości metodycznego i systematycznego podejścia do budowania aplikacji. Zmusza ona bowiem programistów do łączenia funkcji, związanych z przetwarzaniem danych, z funkcjami obsługującymi interfejs użytkownika. Dotyczy to najczęściej tych fragmentów kodu źródłowego programu, które są odpowiedzialne za obsługę okien formularzy edycyjnych. Kod źródłowy programu jest wówczas dosyć nieczytelny, przez co uruchamianie i późniejsza konserwacja oprogramowania są w znacznym stopniu utrudnione.

Środowisko programistyczne C++ Builder, podobnie zresztą jak opracowane wcześniej przez firmę Borland środowisko Delphi, jest bardzo popularne wśród programistów i firm informatycznych, zajmujących się wytwarzaniem oprogramowania. Oba środowiska programistyczne posługują się rozbudowaną biblioteką komponentów VCL (ang. *Visual Component Library*), która znacznie przyspiesza i ułatwia tworzenie aplikacji działających w środowiskach graficznych systemów operacyjnych. Trzeba obiektywnie stwierdzić, że wspomniana wcześniej funkcjonalność środowisk zintegrowanych odpowiedzialna za komunikację z bazami danych została bardzo dobrze zaprojektowana i to ona właśnie zadecydowała o ich dużej popularności. Niezależnie od tego faktu, narzucony przez twórców środowiska model aplikacji jest bardzo odległy od rozwijanych i wdrażanych w ostatnich latach obiektowych metod analizy i projektowania systemów informatycznych. Jest istotą niedogodności, o której tu mowa.

Powstaje zatem poważna przeszkoda w stosowaniu środowiska zintegrowanego C++ Builder w tych projektach, w których oprogramowanie wytwarza się z zastosowaniem metodyk obiektowych. Problem polega w głównej mierze na braku możliwości sprzężenia obiektowego modelu systemu informatycznego z jego implementacją, a właściwe z tą częścią implementacji, która realizuje funkcje komunikacji z bazą danych i edycją jej zawartości. Problem ten najbardziej uwidacznia się w przypadku, w którym w procesie wytwarzania

[1] W Polsce wyłącznym przedstawicielem firmy Borland Software Corporation jest BSC Polska sp. z o. o.

systemów informatycznych korzysta się z oprogramowania narzędziowego typu CASE (ang. *Computer Aided Software Engineering*), szczególnie zaś w przypadku posługiwania się udostępnianymi przez takie oprogramowanie generatorami kodu źródłowego.

W niniejszym artykule podjęto próbę rozwiązania opisanego powyżej problemu. Przedstawione rozwiązanie dotyczy platformy zintegrowanego środowiska programistycznego C++ Builder. W ramach prac nad artykułem opracowany został projekt i implementacja biblioteki komponentów nazwanej przez autorów skrótem BMU (od ang. *Business Model Utility*). Biblioteka ta umożliwia zastosowanie środowiska firmy Borland w procesie tworzenia oprogramowania, którego projekt powstał z zastosowaniem metodyki obiektowej.

1.2. Metodyka oparta na zastosowaniu języka UML

Obiektowe metodyki wytwarzania oprogramowania są rozwijane od dłuższego czasu, chociaż ze względu na ograniczenia wynikające z niewystarczającej mocy obliczeniowej komputerów osobistych ich praktyczne zastosowanie było początkowo bardzo ograniczone. Obecnie oprogramowanie zrealizowane zgodnie z taką koncepcją coraz częściej znajduje się w powszechnym użytku. Należy przy tym zaznaczyć, że obiektowe cechy oprogramowania mogą dotyczyć różnych jego aspektów i różnych etapów procesu powstawania oprogramowania. Implikacje metodyki obiektowej mogą rozciągać się na cały cykl wytwórczy i obejmować etap formułowania wymagań, analizę, projektowanie i implementację. Mogą też dotyczyć jedynie wybranych aspektów, takich jak obiektowe podejście do programowania czy też zastosowanie obiektowych baz danych.

Znaczącym krokiem w rozwoju metodyk obiektowych było opracowanie ujednoliconego sposobu zapisu modeli, jakim jest język UML (ang. Unified Modelling Language) 2. Prace nad UML rozpoczęto w 1995 roku i kontynuowano je intensywnie do roku 1999, kiedy to opublikowano wersję języka oznaczoną symbolem 1.3. Dalsze prace doprowadziły do opublikowania w roku 2002 wersji 2.0, która jest wersją języka obowiązującą obecnie. Prace nad językiem UML prowadzone były głównie w ramach grupy OMG (ang. Object Management Group), organizacji, która opracowała wiele standardów związanych z metodykami obiektowymi. Do najważniejszych, obok UML, specyfikacji opracowanych przez OMG należą m.in. takie znane standardy, jak CORBA (ang. Common Object Request Broker) czy MDA (ang. Model Driven Architecture). W pracach nad językiem UML uczestniczyło wielu specjalistów wywodzących się zarówno ze środowisk naukowych, jak i tych reprezentujących zespoły badawcze dużych firm (IBM, AT&T, Microsoft). Jakkolwiek lista uczestników prac nad UML jest bardzo długa, na czoło wysuwają się trzy nazwiska: Grady Booch, Ivar Jacobson i James Rambaugh. Tym trzem specjalistom, których przed

powstaniem UML kojarzono głównie z opracowanymi przez nich metodykami obiektowymi, przypisuje się autorstwo UML, a także autorstwo większości metod związanych z jego praktycznymi zastosowaniami. Intencja, jaka przyświecała autorom języka UML, jest bardzo czytelna i oczywista. Chodziło mianowicie o połączenie rozwijanych niezależnie, często bardzo odmiennych, spojrzeń na problematykę analizy i projektowania systemów informatycznych, zwłaszcza zaś na problematykę notacji służących do zapisu wiedzy analitycznej i projektowej przyjmujących formę różnego rodzaju diagramów. Takie ujednolicenie było bardzo ważne, gdyż istniejąca przed powstaniem UML niespójność podejść hamowała w znacznym stopniu proces transferu rozwiązań w kierunku zastosowań praktycznych.

Podkreślając ważną rolę notacji, jaką pełni ona w modelowaniu, które towarzyszy wszelkim procesom związanym z analizą i projektowaniem oprogramowania, nie należy zapominać o tym, że notacja nie jest jedynym elementem metodyki. Metodyka w rozumieniu właściwym dla procesów związanych z tworzeniem oprogramowania – to pojęcie o znacznie szerszym znaczeniu, notacja zaś jest tylko jednym z jej elementów. Metodyka określa bowiem nie tylko notację, ale również precyzuje spojrzenie na wiele innych aspektów, związanych z procesem powstawania oprogramowania. Wśród tych aspektów wyróżnić należy sposób usystematyzowania przebiegu prac nad oprogramowaniem, czyli koncepcję tzw. cyklu życiowego oprogramowania (zwanego też cyklem wytwórczym), oraz zestaw narzędzi, najczęściej w postaci określonych narzędziowych programów komputerowych, które wspomagają procesy analityczne i projektowe.

Mówiąc zatem o modelowaniu z zastosowaniem języka UML, należy mieć na uwadze to, że notacja UML jest elementem szerszej koncepcji, którą określono tutaj mianem metodyki obiektowej opartej na zastosowaniu języka UML. Metodyka ta, podobnie jak sam język UML, opracowana została pod kierownictwem wspomnianej wcześniej trójki specjalistów z zakresu metodologii, działającej głównie w ramach założonej w 1980 roku firmy Rational Software (dzisiaj IBM Rational Software). Do najważniejszych elementów wspomnianej metodyki, należą oprócz języka UML, bardzo szczegółowy opis metodyczny dotyczący koncepcji cyklu życiowego oprogramowania prezentowany pod nazwą Rational Unified Process (RUP) oraz rozległy pakiet oprogramowania narzędziowego CASE – Rational Suite. Ze względu na to, że opis metodyki, o której tu mowa, zawarto właśnie w specyfikacji RUP, metodykę tę nazywa się często metodyką Rational Unified Process (metodyką RUP).

1.3. Problem trwałości danych w aplikacjach obiektowych

Paradygmat obiektowy jest obecnie bardzo powszechnie stosowany w procesie wytwarzania oprogramowania. Zastosowanie modelu obiektowego daje wiele możliwości, takich

jak użycie hermetyzacji, dziedziczenia, polimorfizmu, hierarchizacji typów, referencji, tożsamości obiektów itp. Zdecydowana większość aplikacji wymaga mechanizmu zapewniającego trwałość klasom, które są odwzorowaniem tzw. obiektów biznesowych. Przez obiekt biznesowy rozumieć należy przedmiot lub pojęcie pochodzące z dziedziny problemowej, tj. dziedziny działania organizacji, która jest odbiorcą projektowanego oprogramowania. Naturalnym rozwiązaniem zapewniającym zgodność modelu danych aplikacji z modelem bazy danych byłoby zapewnienie trwałości poprzez wykorzystanie obiektowych baz danych. Dodatkowe zalety obiektowych baz danych to szybki dostęp nawigacyjny, lepsze wsparcie dla złożonych typów danych, łatwe zarządzanie. Niestety, obiektowe bazy danych nie uzyskały na rynku tak znaczącej pozycji jak relacyjne i obiektowo-relacyjne bazy danych, mające obecnie pozycję dominującą. Zastosowanie obiektowych baz danych wiąże się wiec z ryzykiem. Dlatego też obecnie najczęściej stosowanym rozwiązaniem jest wykorzystanie relacyjnych lub obiektowo-relacyjnych baz danych. Relacyjne bazy danych są technologią dojrzałą, stosowaną od lat. Posiadają wsparcie dużych dostawców i zalety, takie jak optymalizacja zapytań i bezpieczeństwo. Obiektowo-relacyjne bazy danych są komercyjnym kompromisem pomiędzy modelem relacyjnym i obiektowym. Rozszerzają model relacyjny i język zapytań o wsparcie dla danych obiektowych. Pomimo możliwości zastosowania obiektów złożonych i tzw. składowanych procedur (ang. stored procedures) nie rozwiązują problemu niezgodności modelu danych aplikacji i modelu przechowywania danych aplikacji w bazie danych.

Sprzężenie dwóch odmiennych modeli prowadzi do tak zwanego problemu niezgodności impedancji. Oba modele posiadają znaczące różnice. Model relacyjny oparty jest na dwuwymiarowych tabelach. Związki pomiędzy danymi odwzorowane są poprzez złączenia wierszy w różnych tabelach. Z kolei, w modelu obiektowym przejść pomiędzy obiektami dokonuje się za pomocą referencji. Efektem problemu niezgodności impedancji jest obniżenie jakości oprogramowania, zwiększenie koniecznego nakładu pracy, zmniejszenie poziomu abstrakcji, zmniejszenie spójności. Problem ten został szeroko opisany w literaturze [9÷19].

Powstało wiele rozwiązań łagodzących skutki problemu niezgodności impedancji oraz strategii zapewnienia mechanizmu trwałości obiektów opartych na relacyjnych bazach danych. Najprostszym przypadkiem jest zapewnienie obiektom biznesowym bezpośredniego dostępu do bazy danych poprzez wbudowane zapytania SQL. Rozwiązanie to charakteryzuje się prostą implementacją i efektywnością. Jego wadą jest zbytnie uzależnienie tzw. logiki biznesowej od schematu danych. Prosta zmiana w schemacie danych może wiązać się z potrzebą przebudowy modelu. Utrudnia to pielęgnację i wprowadzanie zmian. Lepszym rozwiązaniem jest zastosowanie obiektów dostępu do danych (DAOs) oddzielających obiekty biznesowe od bezpośredniej interakcji z bazą danych. Obiektom biznesowym przypisuje się odpowiadające im obiekty dostępu do danych. Zazwyczaj jest to odwzorowanie jeden do

jednego implementujące mechanizm wymiany danych ze źródłem danych. Takie podejście pozwala na uniezależnienie implementacji obiektów biznesowych od schematu danych, jednak i tak klasy dostępowe wciąż są silnie powiązane z bazą danych. Rozwinięciem tej koncepcji dla języka Java jest standard JDO. Najbardziej wyszukanym rozwiązaniem jest zastosowanie obiektowo relacyjnej warstwy dostępu odpowiedzialnej za odwzorowanie obiektowego modelu danych do modelu relacyjnego. Warstwa taka w pełni oddziela obiekty biznesowe od części dostępowej do bazy danych. Prowadzi to do zwiększenia poziomu abstrakcji i spójności. Projektanci aplikacji mogą skoncentrować się na logice biznesowej zamiast na szczegółach mechanizmu zapewniania trwałości. Kod źródłowy staje się łatwiejszy w pielęgnacji i rozwoju. Podstawowe zadania warstwy dostępu to odwzorowanie atrybutów obiektów w wiersze w bazie danych, odwzorowanie klas w tabele, odwzorowanie związków pomiędzy klasami oraz wymiana danych pomiędzy obiektami biznesowymi a warstwą bazodanową.

W literaturze opisano wiele technik służących odwzorowaniu obiektowo-relacyjnemu. Jedną z nich opisuje wzorzec CRUD (akronim słów Create, Read, Update, Delete). W czasie wykonywania programu generowane są funkcje wykonujące operacje związane z trwałością obiektów [18]. Inne wzorce to np. Proxy, Basic Relationship Patterns, Active Record oraz wzorzec Data Mapper, będący warstwą oprogramowania izolującą obiekty w pamięci od bazy danych [12]. Podstawą dla wielu rozwiązań stał się szkielet trwałych obiektów (Object Persistence Framework) [11] oparty na architekturze warstwowej, rozdzielający reguły biznesowe od warstwy zajmującej się szczegółami zapewnienia trwałości. Jedną z implementacji szkieletu jest TechInsite Object Persistence Framework (tiOPF) (status open source) stworzony w środowisku Delphi. Inną komercyjną implementacją jest Borland ECO. Ponieważ własna implementacja tych rozwiązań nie jest trywialnym zadaniem, warto skorzystać z któregoś z gotowych narzędzi.

Najwięcej narzędzi powstało dla języka Java, między innymi niekomercyjne narzędzia wymienione poniżej:

- Hibernate – obiektowo-relacyjna usługa zapewniająca mechanizm trwałości i zapytań.
- Cayenne – szkielet obiektowo relacyjny.
- ObJectRelationalBridge (OJB) – mechanizm zapewniający przeźroczystą trwałość obiektom Java, stanowi część projektu Jakarta.

Dla języka C++ powstał zgodny z biblioteką MFC system Persistent Data Objects będący uboższą obiektową bazą danych rezydującą pomiędzy modelem obiektowym a końcową bazą danych.

Dla platformy .NET powstało między innymi komercyjne narzędzie Genome zapewniające trwałość klasom .NET w relacyjnej bazie danych. Narzędzie służy generacji warstwy dostępu do danych na podstawie modelu klas i słownika dostarczonych przez projektanta.

2. Proponowane podejście

2.1. Analiza problemu

Podstawowe pytanie, jakie należy zadać rozważając możliwości rozszerzenia środowiska Borland C++ Builder o elementy integrujące jego koncepcję z koncepcją metodyki obiektowej, dotyczy ustalenia sposobu implementacji, który zagwarantuje pełne wykorzystanie sporych przecież możliwości biblioteki VCL. Szczególnie dotyczy to tej funkcjonalności biblioteki, która znajduje się w warstwie komunikacji aplikacji z bazą danych. Aby móc odpowiedzieć na tak postawione pytanie, należy zastanowić się nad kilkoma aspektami rozważanej integracji.

Pierwszy aspekt dotyczy pokonania różnic koncepcyjnych pomiędzy podejściem obiektowym i strukturalnym, które to podejście *de facto* zostało narzucone programistom przez twórców środowiska, pomimo iż sama biblioteka VCL, stanowiąca jego trzon, zrealizowana została z zastosowaniem technik obiektowych. Sięgając głębiej, należy zauważyć, że przyczyna tych różnic leży w rozbieżnościach pomiędzy schematem obiektowym, który jest istotą samego podejścia obiektowego, a schematem relacyjnym, na którym oparto koncepcję stosowanych nadal bardzo powszechnie relacyjnych baz danych.

Drugi aspekt problemu związany jest z zaprojektowanymi przez twórców biblioteki VCL mechanizmami integracji aplikacji z bazami danych i potencjalnymi możliwościami wykorzystania ich funkcjonalności. Innymi słowy, chodzi o to, by nie powtarzać wykonanej już raz pracy, a w szczególności o to, by móc uchwycić precyzyjnie to miejsce w strukturze klas biblioteki VCL, które pozwala na pobranie danych niezbędnych do dokładnego odtworzenia schematu obiektowego. Odtworzenie schematu obiektowego klas realizujących zasadnicze funkcje aplikacji jest bowiem najważniejszym elementem integracji środowiska programistycznego ukierunkowanego na współpracę z bazami relacyjnymi, z metodyką obiektową.

Trzecim elementem rozważań podjętych przez autorów niniejszej pacy jest techniczny aspekt realizacji wspomnianego mechanizmu odtwarzania struktury klas, których stan wyrażony jest przez wartości atrybutów zapisane w relacyjnej bazie danych. W warstwie bardziej technologicznej oczywiste jest wybranie przewidzianej przez twórców środowiska C++ Builder drogi wprowadzania doń własnych rozszerzeń poprzez implementację dodatkowych komponentów. W warstwie koncepcyjnej należy ustalić, jakie to powinny być komponenty i jakie funkcje powinny one realizować.

2.2. Koncepcja rozwiązania

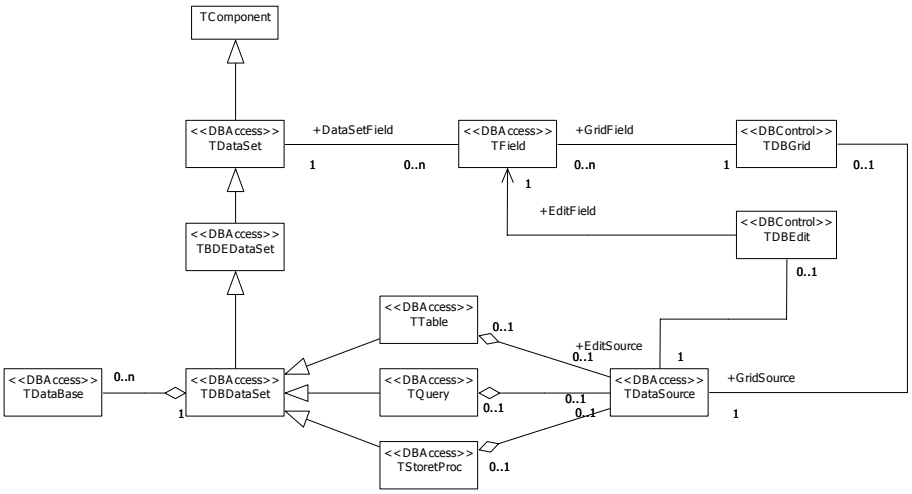
Przyjęta koncepcja rozwiązania problemu dotyczącego rozbieżności pomiędzy schematami: relacyjnym i obiektowym opiera się na trzech założeniach:

1. W kwestii odzwierciedlania relacji uogólnienia można pozostawić projektantowi aplikacji możliwość zastosowania dowolnego podejścia: bądź to spłaszczenia kraty dziedziczenia, bądź jej dokładnego odwzorowania w bazie danych. Możliwe jest także zastosowanie podejścia pośredniego.
2. Odwzorowanie typu klasy powinno nastąpić poprzez jawne skojarzenie typu klasy z nazwą tabeli lub zbiorem krotek relacji będących wynikiem określonej projekcji i (lub) selekcji, względnie złączenia.
3. Problem gromadzenia instancji klas najlepiej rozwiązać poprzez zastosowanie uniwersalnych agregatorów, tj. kolekcji klas pozwalających na gromadzenie odnośników do abstrakcyjnej klasy obiektu bazowego – „klasy-korzenia" – mogącej być przodkiem wszystkich innych klas w budowanej strukturze.

Aby najpełniej wykorzystać możliwości biblioteki VCL ułatwiające programiście konstruowanie warstwy komunikacji z bazą danych, trzeba skorzystać z komponentu, który w strukturze klas usytuowany jest możliwie jak najwyżej, a jednocześnie posiada odpowiednie funkcje. Chodzi o to, żeby zachowując pełną funkcjonalność nie uzależniać jednocześnie implementacji przyszłej biblioteki od szczegółów związanych z obsługą konkretnych standardów komunikowania się aplikacji z bazami danych. Rolę takiego uogólnienia w bibliotece VCL pełni abstrakcyjna klasa *TDataSet*. Klasa ta reprezentuje dowolny zbiór krotek relacji i jest całkowicie niezależna od źródła, z którego pochodzą dane. Wszystkie wyspecjalizowane klasy reprezentujące tabele, zapytania, wyniki wykonania procedur SQL składowanych w bazie lub zbiory rekordów rezydujące w pamięci są potomkami tej właśnie klasy. Z tego względu to właśnie z poziomu klasy *TDataSet* najlepiej pobierać wartości atrybutów podczas odtwarzania schematu obiektowego utrwalonego w bazie danych. Fragment struktury klas komponentów biblioteki VCL związanych z interfejsem aplikacji odpowiedzialnym za komunikację z bazą danych przedstawiono na rys. 1. Dla jasności, przedstawiono jedynie ten fragment struktury klas potomnych klasy *TDataSet*, który dotyczy obsługi standardu BDE (ang. Borland Database Engine).

Mechanizmy utrwalania i odtwarzania obiektów na podstawie danych zapisanych w różnej postaci w plikach lub bazach danych – generalnie chodzi o zapis na nośnikach pamięci masowej, który to zapis różni się o zapisu w pamięci operacyjnej – są już dobrze znane. Część z tych mechanizmów utrwalono w ogólnie przyjętych i stosowanych szablonach projektowania i implementacji, czyli w tzw. wzorcach projektowych (ang. *design patterns*). Jednym z takich mechanizmów jest mechanizm tzw. fabryki klas (ang. *class factory*), który również utrwalono we wspomnianych wzorcach projektowych. Wzorzec ten posiada funkcje pozwalające na wywoływanie odpowiednich konstruktorów klas w zależności od danych zapisanych w pliku lub bazie danych. Jedną z odmian tego wzorca jest tzw. abstrakcyjna fabryka klas (ang. *abstract class factory*). Klasa ta potrafi sama odtwarzać klasy, przy czym

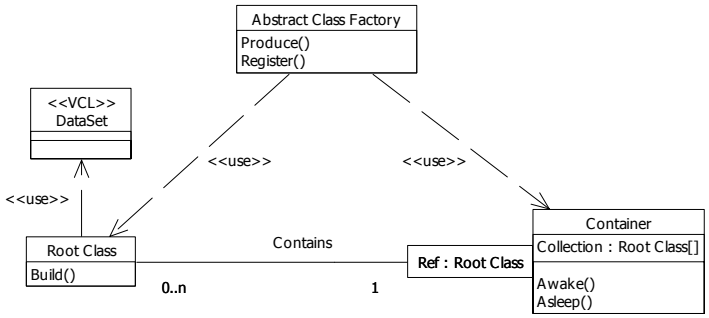
na poziomie kodu źródłowego klasy nie są zawarte żadne informacje na temat tego, jakie klasy mogą być przez nią konstruowane. Innymi słowy, w kodzie tym nie wymienia się jawnie konstruktorów określonych klas. Konieczne jest jednak wcześniejsze zarejestrowanie w odpowiedniej strukturze danych, identyfikatora klasy, a w językach o silnych typach, takich jak C++, konieczne jest skojarzenie z tym identyfikatorem adresu funkcji, najczęściej statycznej, która zawiera wywołanie określonego konstruktora i zwraca w wyniku adres będący wskazaniem na obiekt abstrakcyjnej klasy-korzenia (ang. *root class*). Warunkiem jest jednak to, aby klasa, którą chcemy odtworzyć, była pochodną wspomnianej klasy-korzenia. W opracowanej w ramach prac nad artykułem bibliotece BMU zastosowano takie właśnie rozwiązanie.



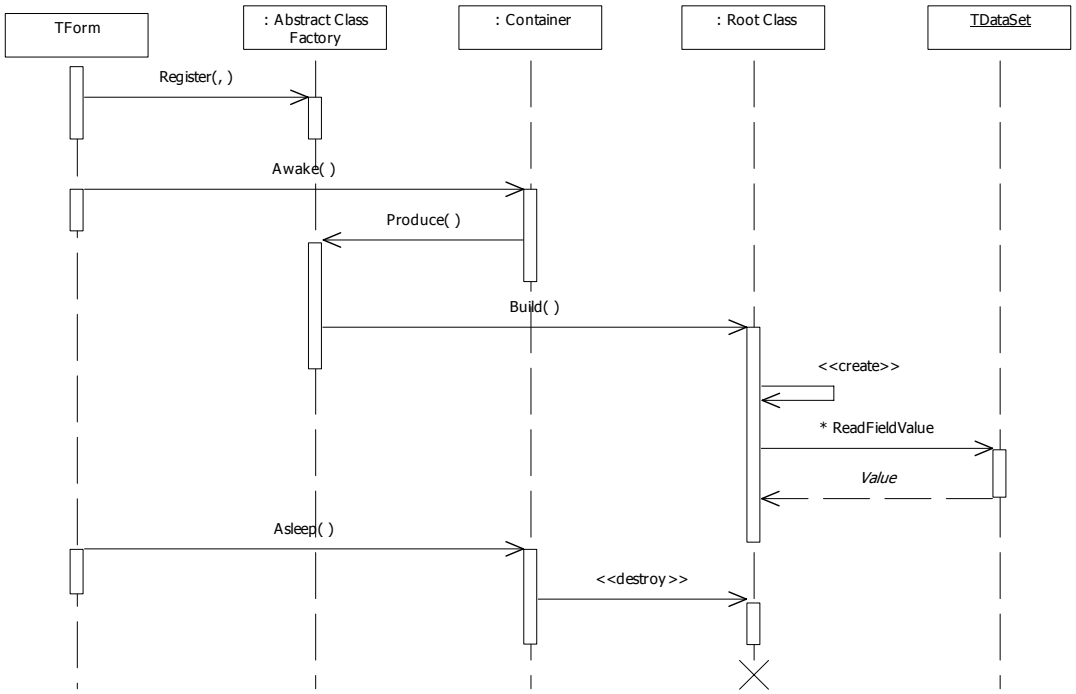
Rys. 1. Fragment struktury klas biblioteki VCL odpowiedzialnych za komunikację aplikacji z bazą danych
Fig. 1. The part of the VCL class hierarchy containing selected classes responsible for the access to databases

Ostatnim problemem, jaki pozostaje do rozwiązania, jest problem gromadzenia instancji odtwarzanych klas. Ponieważ jednak wszystkie odtwarzane obiekty mają wspólną klasę bazową, wystarczy w odpowiedniej uniwersalnej strukturze zapisać referencje lub wskazania na utworzone obiekty.

Na rys. 2 przedstawiono analityczną strukturę klas zaimplementowanych w bibliotece BMU. Struktura ta jest oczywiście jedynie zarysem dalszego projektu i implementacji, pozwala jednak na zobrazowanie przebiegu procesu odtwarzania instancji klasy. Proces ten zobrazowano dokładniej na rys. 3. w postaci diagramu przebiegu. Warto w tym momencie zauważyć, że jedyna znaczeniowa zależność pomiędzy klasami bibliotek VCL i przedstawionych na diagramie klas biblioteki BMU dotyczy wyłącznie klasy *TDataSet*, co pozwala na całkowite uniezależnienie działania biblioteki od szczegółów implementacyjnych związanych z konkretnym standardem komunikowania się aplikacji z bazą danych.



Rys. 2. Analityczna struktura klas biblioteki BMU
Fig. 2. The BMU library class hierarchy – the analysis model

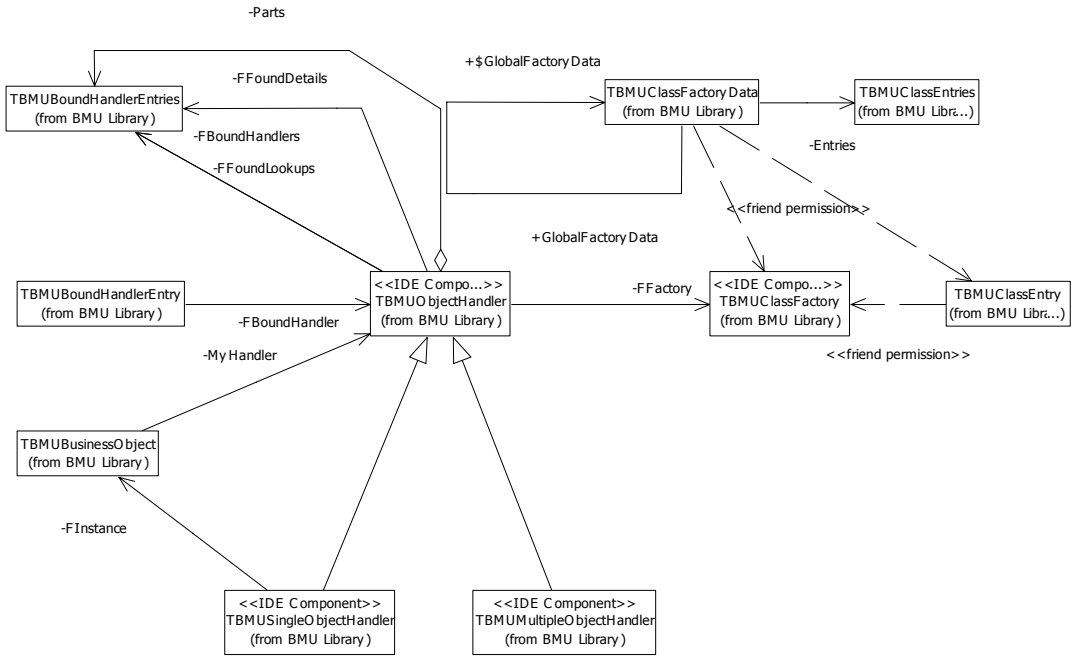


Rys. 3. Diagram przebiegu interakcji związanej z odtwarzaniem i kasowaniem obiektu
Fig. 3. The sequence diagram showing the interaction that creates and destroys an object

2.3. Projekt i implementacja

Zasadniczą część modelu projektowego biblioteki BMU stanowi pięć klas (rys. 4). Trzy z nich odpowiadają dokładnie klasom modelu analitycznego, a więc klasom *Root Class*, *Abstract Class Factory* oraz klasie *Container*. Są to klasy abstrakcyjne, którymi są odpowiednio: *TBMUBusinessObject*, *TBMUClassFactory* i *TBMUObjectHandler*. Dwie pozostałe, tj. *TBMUSingleObjectHandler* i *TBMUMultipleObjectHandler*, precyzują określony sposób odtwarzania schematu. Pierwsza klasa, którą nazwać można zarządcą pojedynczej

instancji, odtwarza w pamięci operacyjnej tylko jedną instancję odtwarzanej klasy, zawsze tą, która związana jest z bieżącym rekordem zbioru danych. Rekord bieżący to ten, na którym ustawiony jest wskaźnik pozycji. Wartości pól tegoż rekordu zapisywane są już na poziomie biblioteki VCL w odpowiedniej strukturze (właściwość *Fields* klasy *TDataSet*), co ułatwia znacznie ich odczytywanie i zapisywanie. Druga z wymienionych, tzn. *TBMUMultiple-ObjectHandler*, pozwala na zarządzanie wieloma instancjami. Dlatego też nazwano ją zarządcą wielu instancji. Klasa ta tworzy instancje dla wszystkich rekordów znajdujących się w zbiorze danych z uwzględnieniem bieżących ustawień filtrowania.

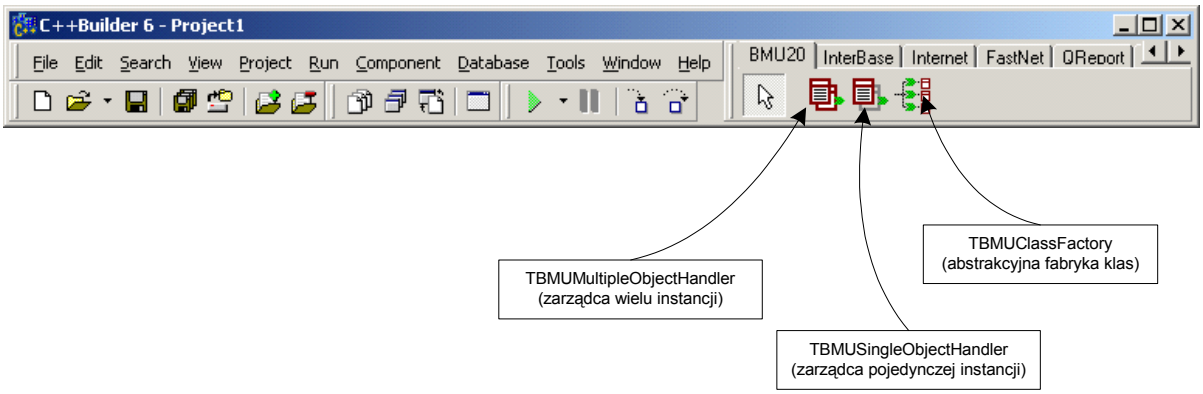


Rys. 4. Projektowy diagram klas biblioteki BMU
Fig. 4. The BMU class diagram – design model

Istotną cechą zaprojektowanej biblioteki jest możliwość odtwarzania struktur obiektów na podstawie struktur powiązań pomiędzy zbiorami danych. Dotyczy to zbiorów reprezentujących powiązane ze sobą tabele bazy danych. Jeśli tak zadecyduje projektant aplikacji, zarządca instancji może sam podążać śladem powiązań typu master-detail pomiędzy zbiorami danych i odtwarzać kolejne elementy schematu zapisanego w bazie danych (właściwości *AutoFindDetails* i *BuildWithPart*s ustawione na wartość *true*). Możliwe jest także jawne wskazanie zbiorów, które chcemy w danym momencie odtworzyć w budowanym schemacie obiektowym, co można osiągnąć podając nazwy zarządców dla tych zbiorów danych, które tworzą określoną strukturę (właściwość *BoundHandlers*). Dzięki temu mechanizmowi możliwe stało się odtwarzanie schematu obiektowego dla bardziej złożonych sytuacji, np. dla formularzy edycyjnych operujących na kilku tabelach bazy danych.

Jak już wspomniano, podczas implementacji biblioteki posłużono się standardowymi mechanizmami rozszerzania środowisk IDE firmy Borland. Ich istota polega na umożliwie-

niu użytkownikowi środowiska implementacji własnych wyspecjalizowanych komponentów. Implementacja własnych komponentów jest stosunkowo prosta i sprowadza się do utworzenia odrębnego projektu tzw. pakietu (ang. *package*). Pakiet zawiera zazwyczaj kod źródłowy kilku komponentów, składających się na określoną bibliotekę. Po wykonaniu kompilacji pakietu i jego zainstalowaniu użytkownik środowiska może korzystać z nowych komponentów na takich samych zasadach, na jakich korzysta z komponentów dostarczonych wraz ze środowiskiem. Interfejs biblioteki BMU stanowią trzy komponenty, które reprezentują fabrykę klas oraz omówione wcześniej klasy zarządców instancji. Widok palety komponentów biblioteki BMU w obecnej wersji (wersja 2.0) przedstawiono na rys. 5.



Rys. 5. Paleta komponentów biblioteki BMU
Fig. 5. The BMU package palette

3. Tworzenie oprogramowania z wykorzystaniem biblioteki BMU

3.1. Ogólne zasady korzystania z biblioteki

Komponent fabryki klas *TBMUClassFactory* odpowiedzialny jest za tworzenie instancji zarejestrowanych klas będących potomkami klasy korzenia.

Na żądanie któregoś z powiązanych komponentów zarządzających wywoływane są metody budujące odtwarzanych klas. Cała konfiguracja komponentu sprowadza się do nadania mu preferowanej nazwy.

Komponenty *TBMUSingleObjectHandler* i *TBMUMultipleObjectHandler* posiadają właściwość *ClassName* określającą nazwę klasy odtwarzanej w pamięci operacyjnej.

Określając adres funkcji obsługującej zdarzenie *OnGetBuildFunct*, projektant wskazuje funkcję budującą odtwarzanej klasy. Zdarzenie *OnGetBuildFunct* powinno zawierać wywołanie metody *SetBuildFunction*, parametrem której jest wskaźnik na statyczną funkcję budującą.

Właściwość *Factory* to nazwa, która została nadana komponentowi fabryki klas. Komponent zarządcy instancji przez właściwość *DataSet*. można powiązać z komponentem dostępowym reprezentującym zbiór wierszy (jest on zawsze potomkiem klasy *TDataSet*). Projektant aplikacji może odwzorować strukturę powiązań klas biznesowych, wykorzystując właściwość *BoundHandlers*. Służy ona do określenia listy klas zarządców powiązanych z komponentem zarządców innych obiektów biznesowych.

Ustawienie właściwości *AutoFindDetails* oraz *BuildWithParts* na wartość *true* pozwala na automatyczne odtworzenie struktury schematu na podstawie powiązań master-detail.

Odtwarzane klasy muszą być zaprojektowane według następujących wytycznych:

1. Muszą być potomkami klasy korzenia *TBMUBusinessObject*.
2. Wymagany jest konstruktor postaci:

```
explicit <TUserClass> ( TBMUObjectHandler * pMyHandler ) : TBMUBusinessObject(
pMyHandler )
```

3. Klasa musi mieć zdefiniowaną statyczną funkcję budującą, odpowiedzialną za utworzenie instancji klasy w pamięci operacyjnej na podstawie danych z bieżącego wiersza zbioru danych. Poniżej podano sygnaturę funkcji budującej:

```
static TBMUBusinessObject* Build ( TBMUObjectHandler * pHandler )
```

4. Implementacja funkcji budującej powinna zawierać:
   - wywołanie konstruktora,
   - wypełnienie atrybutów nowo utworzonego obiektu klasy danymi z bieżącego wiersza zbioru danych. Dostęp do tych danych można uzyskać za pośrednictwem skojarzonego z budowaną klasą komponentu zarządcy instancji, ustawiając odpowiednio właściwość *DataSet*,
   - aby zapewnić dostęp z poziomu odtworzonej instancji klasy do instancji klas powiązanych, należy skorzystać z metody *GetPart*, która zwraca wektor wskaźników na obiekty podanej klasy powiązanej.

Odwołanie się do właściwości *Instance* komponentu zarządcy uruchamia mechanizm odtwarzania klas ze zbiorów danych. W przypadku komponentu *TBMUSingleObjectHandler* tworzona jest jedynie instancja odpowiadająca aktualnemu wierszowi zbioru danych oraz ewentualnie instancje klas powiązanych. Inaczej wygląda to w przypadku komponentu *TBMUMultipleObjectHandler*, gdzie instancje klasy budowane są w oparciu o wszystkie wiersze z uwzględnieniem bieżących ustawień filtrowania.

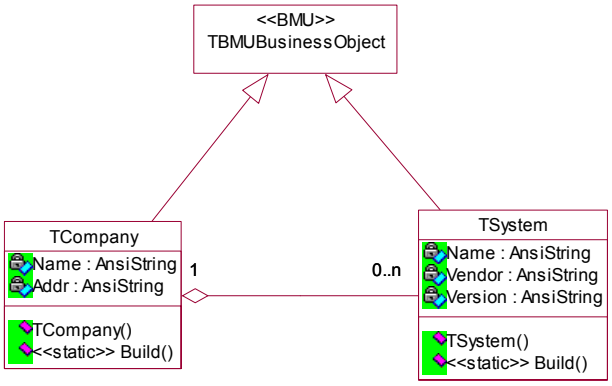
Operacja *First* klasy *TBMUSingleObjectHandler* zwraca obiekt klasy odpowiadającej pierwszemu wierszowi tabeli. Wskaźnik bieżącego wiersza w zbiorze danych ustawiany jest na pierwszy rekord. Operacja *Next* ustawia wskaźnik bieżącego wiersza na następny wiersz

i zwraca odpowiadający mu obiekt klasy. Operacje *First* i *Next* klasy *TBMUMultipleObject-Handler* zwracają odpowiednio pierwszy i kolejny obiekt kolekcji odtworzonych obiektów.

Komponenty zarządców poza odtwarzaniem obiektów biznesowych umożliwiają także zapis bieżącego stanu obiektów. Do tego celu przeznaczona jest metoda *Save* oraz operatory zapisu: operator += dodanie obiektu do bazy danych oraz utworzenie nowego obiektu oraz operator << zapisanie stanu obiektu do bazy danych.

3.2. Przykład wykorzystania biblioteki BMU do odtworzenia fragmentu schematu obiektowego

Na rys. 6 przedstawiony został prosty schemat stanowiący część modelu przykładowej aplikacji służącej do zarządzania systemami informatycznymi. Pomiędzy klasami *TCompany* i *TSystem* występuje relacja powiązania. Ponieważ obiekty tych klas wymagają trwałości, zostały odwzorowane w odpowiadających im tabelach. Do danych pochodzących z tych tabel można uzyskać dostęp poprzez preferowane komponenty bazodanowe.



Rys. 6. Część modelu logiki biznesowej przykładowej aplikacji
Fig. 6. A part of business object model of a sample application

Pomiędzy komponentami bazodanowymi mającymi dostęp do tabel odpowiadających klasom *TCompany* i *TSystem* należy ustalić relację typu master-detail.

Przykładowa deklaracja klasy *TCompany*:

```
class TCompany : public TBMUBusinessObject
{
private:
   AnsiString Name;
   AnsiString Addr;
   ObjectVectorType Systems;
protected:
   virtual AnsiString Body( int pLevel );
public:
   explicit TCompany ( TBMUObjectHandler * pMyHandler ) : TBMUBusinessObject(
pMyHandler ) {};

   static TBMUBusinessObject* Build( TBMUObjectHandler * pHandler );
   virtual AnsiString ClassName( void ) { return "TCompany"; };
   virtual AnsiString ObjectName( void ){ return Name; };
};
```

Atrybut *Systems* typu *ObjecVectorType* jest dynamiczną tablicą wskaźników na powiązane instancje klasy *TSystem*. Zdefiniowany w bibliotece typ *ObjectIteratorType* pozwala na zadeklarowanie iteratora dla wektora obiektów odtwarzanej klasy. Poniżej przedstawiono przykład funkcji budującej obiekty klasy *TCompany*.

```
TBMUBusinessObject* TCompany::Build( TBMUObjectHandler * pHandler )
{
   TCompany * f;

   f = new TCompany(pHandler);
   f->Name = pHandler->DataSet->FieldByName("Name")->AsString;
   f->Systems = pHandler->GetPart( "TSystem" );

   return (TBMUBusinessObject *)f;
};
```

Atrybut *pHandler* klasy *TBMUBusinessObject* umożliwia odwołanie się z klasy odtwarzanej do przypisanej do niej klasy komponentu zarządcy. Za pośrednictwem właściwości *Instance* komponentu zarządcy (lub właściwości *Instances* w przypadku komponentu zarządcy wielu instancji) programista ma dostęp do odtworzonego obiektu lub do kolekcji odtworzonych obiektów.

Analogicznie można zadeklarować klasę *TSystem.* Deklaracja różniłaby się brakiem atrybutów typu *ObjecVectorType*, gdyż w rozważanym przykładzie klasa *TSystem* nie jest w związku całość-część z inną klasą biznesową.

Zakładając, że w danej chwili użytkownika interesują dane dotyczące tylko jednej firmy, do zarządzania instancjami klasy *TCompany* można zastosować komponent *TBMUSingleObjectHandler*. Odtwarzanie wszystkich instancji w pamięci nie jest w tym przypadku konieczne. Do formularza należy dodać komponenty *TBMUClassFactory*, *TBMUSingleObjectHandler* oraz *TBMUMultipleObjectHandler.* Kolejnym krokiem jest określenie właściwości komponentu zarządcy pojedynczej instancji. Przykładowe wartości właściwości komponentu:

- *Name*: hCompany,
- *Factory* : classFactory,
- *DataSet*: nazwa komponentu dostępowego reprezentującego tabelę odwzorowującą klasę *Tcompany,*
- *ClassName*: “TCompany”,

Właściwościom *BuildWithParts* oraz *AutoFindDetails* należy przypisać wartość *true*, tak aby komponenty zarządców zostały powiązane automatycznie na podstawie relacji pomiędzy podlegającymi im zbiorami danych.

Należy określić również adres funkcji obsługującej zdarzenie *OnGetBuildFunct* tak, aby móc wskazać funkcję budującą dla klasy *TCompany.*

```
void __fastcall TForm1::hCompanyGetBuildFunct(TObject *Sender)
```

```
{
   hCompany->SetBuildFunction( TCompany::Build );
}
```

Przykładowa konfiguracja komponentu *TBMUMultipleObjectHandeler* jest następująca:

- *Name*: hSystems,
- *Factory* : BMUClassFactory1,
- *DataSet*: Nazwa komponentu reprezentującego zbiór wierszy tabeli System,
- *ClassName*: "TSystem".

Dla właściwości *BuildWithParts* i *AutoFindDetails* należy pozostawić domyślną wartość *false*. W kodzie funkcji obsługującej zdarzenie *OnGetBuildFunct* należy wskazać metodę budującą klasy *TSystem*.

Poniżej podano prosty przykład odwołania się do obiektów klasy *TCompany* za pomocą komponentu zarządcy pojedynczej instancji.

```
AnsiString xml;
 TCompany *bo = (TCompany*)(hCompany->First());
    while ( bo )
    {
       xml += bo->XML();
       bo = (TCompany*)(hCompany->Next());
    }
    re1->Text=xml;
```

4. Podsumowanie

Ponowne spotkanie się obu zasadniczych nurtów metodologicznych, jakie wyłoniły się na przestrzeni ostatnich kilku dekad rozwoju inżynierii oprogramowania, tzn. historycznie wcześniej rozpoczętego nurtu strukturalnego i późniejszego, obiektowego, jest niewątpliwie zjawiskiem ciekawym. Nie spodziewali się tego z pewnością stojący w opozycji do zasad modelowania strukturalnego pomysłodawcy metod obiektowych. Zaistniała sytuacja ma jednak swoje głębokie przesłanki praktyczne. Wynika ona bowiem z faktu utrwalenia się tych elementów obu podejść, które zyskały, ze względu na swoją przydatność, szerokie praktyczne zastosowanie. W przypadku modelu strukturalnego utrwalił się przyjęty relacyjny model danych oraz oprogramowanie systemów zarządzania bazami danych, które realizuje jego postulaty. Sama strukturalna metodyka projektowania, być może poza jej elementami odnoszącymi się do formułowania wymagań, nie przyjęła się. Została ona wyparta przez podejście obiektowe, szczególnie takie, w którym bazuje się na zastosowaniu notacji języka UML. Metodyka obiektowa pozwoliła na przeniesienie do struktur programów komputerowych elementów rzeczywistego świata i znacznie ułatwiła analitykom i projektantom dekompozycję bardziej złożonych zagadnień. Jej słabym elementem były jednak mechanizmy składowania danych, a w szczególności niemożność ugruntowania się i upow-

szechnienia określonych standardów dla tzw. obiektowych baz danych. Stąd wyniknęła konieczność ponownego odwołania się do mechanizmów składowania danych przyjętych w koncepcji strukturalnej.

Opracowane przez autorów artykułu rozwiązanie problemu integracji modelu obiektowego ze środowiskiem programistycznym pracującym z bazami relacyjnymi jest z pewnością sporym uproszczeniem zarysowanego w artykule problemu. W porównaniu z innymi rozwiązaniami, szczególnie tworzonymi komercyjnie, jego funkcjonalność jest pewnością ograniczona. Jak jednak wynika z praktycznych testów, zastosowanie biblioteki BMU pozwala na tworzenie znacznie bardziej czytelnych i poprawnych metodycznie aplikacji obiektowych i ułatwia pracę projektanta. Ponadto, zastosowanie biblioteki wymusza na projektancie wcześniejsze zrozumienie dziedziny problemowej programu, co wpływa bardzo pozytywnie na jakość powstającej aplikacji.

Nakreślając plan dalszych prac nad rozwiązaniem opisanym w artykule, należy uwzględnić fakt, że firma Borland wprowadziła do najnowszej wersji swojego środowiska IDE, a mianowicie w ukazującym się na rynku w momencie powstawania tego artykułu Borland Developer Studio 2006, rozwiązanie Borland ECO będące komercyjnym rozwiązaniem dla problemu integracji obiektowo-relacyjnej. Należy jednak podkreślić, że pełna wersja platformy ECO dostępna jest jedynie w wersji Enterprise Architect środowiska, która jest wersją najdroższą, co z pewnością uniemożliwi jej wykorzystanie w mniejszych firmach oraz dla celów edukacyjnych. Tak więc prace nad zagadnieniem będą kontynuowane. Te najbliższe dotyczyć będą sprzężenia podążającego w odwrotnym kierunku, tzn. konstruowania schematu relacyjnego na podstawie struktur tworzonych przez obiekty.

LITERATURA

1. Cattell R. G. G.: Object Data Management: Object Oriented and Extended Relational Database Systems. Addison-Wesley, 1994.
2. Booch G., Rumbaugh J., Jacobson I.: UML Przewodnik użytkownika. WNT, Warszawa 2002.
3. Vossen G.: Bibliography on Object Oriented Database Management. Technical Report No. 9301, Computer Science Group, Univ. of Gissen, Germany 1993.
4. Jaszkiewicz A.: Inżynieria oprogramowania. Helion, Gliwice 1997.
5. Cooper J.: Building an Object Persistence Framework. http://www.tabdee.ltd.uk/ Software-/Papers/BuildingAnOPF/BuildingAnOPF.html .
6. Fowler M.: Patterns of Enterprise Application Architecture. Addison-Wesley 2003.

7. Ambler S. W.: The Design of a Robust Persistence Layer for Relational Databases. http://www.ambysoft.com/scottAmbler.html .
8. Carter J.: Object Store. http://www.carterconsulting.org.uk/ .
9. Brown P.: An Object-Oriented Persistence Layer Design. http://codecentral.borland.com/.
10. Nock C.: Data Access Patterns: Database Interactions in Object-Oriented Applications. Addison-Wesley 2003.
11. Srinivasan V., Chang D. T.: Object persistence in object-oriented applications. IBM Systems Journal Volume 36, No. 1, 1997.
12. Keller W.: Persistence Options for Object-Oriented Programs. JAOO, 2003.
13. Ambler S. W.: Mapping Objects to Relational Databases: O/R Mapping In Detail. http://www.agiledata.org/essays/mappingObjects.html.
14. Keller W.: Mapping Objects to Tables, Proceedings EuroPLoP, 1997.
15. Philippi S.: Model driven generation and testing of object-relational mappings. Journal of Systems and Software Volume 77, No. 2, 2005.
16. Keller W.: Object Relational Access Layers - A Roadmap, Missing Links and more Patterns. Proceedings EuroPLoP, Germany 1998.
17. Fussel M. L.: Foundations of Object Relational Mapping. http://www.chimu.com/ publications/objectRelational/ .
18. Yoder J. W., Johnson R.E., Wilson Q.D.: Connecting Business Objects to Relational Databases. http://www.joeyoder.com//Research/objectmappings/Persista.pdf .
19. Ramanathan C.: Providing object-oriented access to existing relational databases. Mississippi State University, 1998.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 9 sierpnia 2006 r.

Abstract

The serious inconvenience of the modern integrated development environments (IDE's), such as Borland C++ Builder is, that in most cases the IDE's remain incoherent with object programming model and modern object methodology of system analysis, design and implementation. It is so, because in the IDEs' design too much attention has been paid to the concepts of their integration with relational databases, while object modelling concepts where fully omitted. In opinion of authors, this causes situation in with programmers create

programs ineffectively by merging business functions with functions being responsible for the communication with databases and for GUI.

The mentioned inconvenience creates a serious barrier for realizing in such development environments those software projects in witch object methodologies are being applied. The incoherence between IDE's and object modelling concepts can be seen especially in projects being supported with Computer Aided Software Engineering tools (CASE) and their so called Lower CASE functionality that enables the automatic generation of the certain parts of the programs' source code.

This article presents the description of the concepts, design – presented on figure (4) and implementation of the component library created by the authors of the article for the Borland C++ IDE. The library has been called Business Model Utility (BMU) because of the fact that it allows to transfer the entities and events met in business environment that the program is designed for, to the code of the program created using the C++ programming language. This makes the programmer a possibility to separate the business functions, that he can encapsulate in classes derived from the business, analysis and design model of the system, from the database integration and the GUI functions. The article also describes usage of library regarding development of exemplary application based on class hierarchy presented on figure (6).

Adresy

Jacek SZEDEL: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-101 Gliwice, Polska, jszedel@star.iinf.polsl.gliwice.pl.
Michał KOLANO: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-101 Gliwice, Polska, mkolano@star.iinf.polsl.gliwice.pl.
Jarosław CHOJNACKI: chojnacki.j@op.pl.