AUTOMATYKA • 2006 • Tom 10 • Zeszyt 3

Paweł Skrzyński*

Język UML 2.0 w modelowaniu relacyjnych baz danych

1. Wprowadzenie

Proces projektowania systemów relacyjnych zwyczajowo przebiega przez trzy etapy:

1) modelowanie konceptualne,
2) modelowanie logiczne,
3) modelowanie fizyczne.

Większość klasycznych metod modelowania dostarcza wsparcia głównie w pierwszych dwóch etapach. Model konceptualny stanowi specyfikację modelu logicznego i opisuje dziedzinę systemu oraz jego funkcjonalność. Model logiczny, który rozważamy w tym artykule, jest modelem relacyjnym. Model konceptualny jest zazwyczaj przedstawiany w postaci diagramu ERD, podczas gdy na model logiczny składają się relacje – czyli tabele systemu relacyjnego. Transformacja pomiędzy tymi dwoma metodami jest dosyć oczywista i szybka, jednak właściwie na tym kończy się jakiekolwiek wsparcie dla projektantów.

Z drugiej strony diagram klas UML jest używany właściwie we wszystkich etapach pracy nad modelem – od fazy analizy, poprzez projektowanie, szczegółowe projektowanie itd. Oprócz tego istnieje możliwość generowania nawet działającego kodu z modelu UML, a w literaturze można się spotkać z terminem „executable UML". Żeby móc utworzyć wykonywalny model należy oczywiście oprócz zamodelowania statycznej perspektywy systemu dostarczyć specyfikacji zachowania. Specyfikacja zachowania może odbywać się poprzez definiowanie stanów – maszyny stanowe lub bez nich – poprzez modelowanie ciała operacji/metody.

W obydwu przypadkach zachowanie można opisywać poprzez:

– maszynę stanową na diagramie stanów,
– tekstowy opis na diagramie tekstowym,
– diagram czynności.

Diagramy czynności pokazują przepływ sterowania od czynności do czynności i są bardzo przydatne w modelowaniu procesów biznesowych lub modelowaniu ciała metod. Należą do grupy dynamicznych diagramów UML. Ich zadanie polega na ogół na przedsta-

* Katedra Automatyki, Akademia Górniczo-Hutnicza w Krakowie

wieniu sekwencyjnych (rzadziej współbieżnych) kroków procesu obliczeniowego – i to ich zastosowanie jest najciekawsze z punktu widzenia tego artykułu. Można na nim przedstawić również zmiany zachodzące w obiekcie, gdy przechodzi z jednego stanu do drugiego [1].

Jak widać, mogą one stanowić doskonałe uzupełnienie diagramu klas – diagram klas określa atrybuty klasy/obiektu oraz, poprzez metody, zachowania, jakich możemy oczekiwać od obiektów danej klasy, jednak nie mówi nic o sposobie implementacji metod, co natomiast może być modelowane na diagramach czynności. Metody stanowią specyfikację usług jakie klasa udostępnia innym klasom, więc w pewnym sensie stanowią definicję protokołu komunikacji w systemie. Jednak informacja ta jest całkowicie gubiona w prostej transformacji Encja-Klasa.

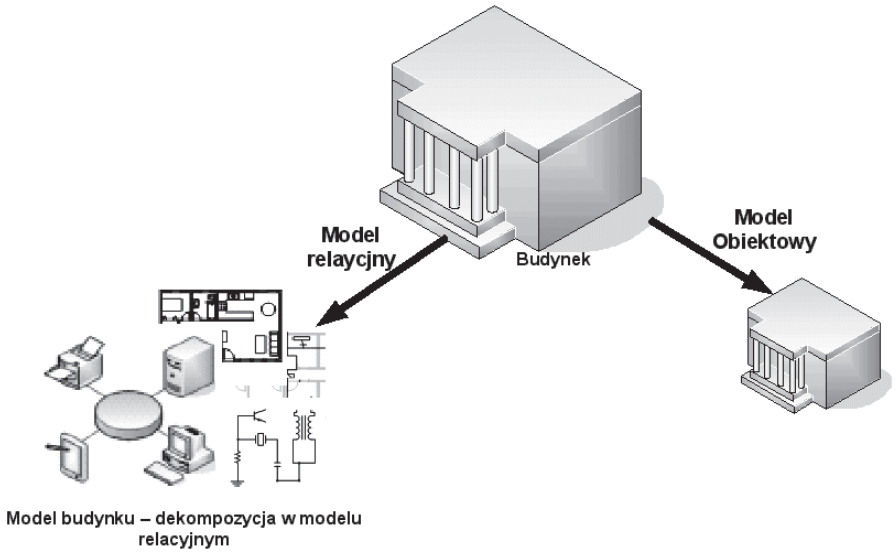
Jak wiadomo, w systemach relacyjnych, w języku SQL, możemy implementować nie tylko statyczną część systemu w postaci relacyjnych tabel, ale również operacje, które zachodzą na danych. Odbywa się to poprzez procedury wbudowane (*stored procedures*) i funkcje, które są również przechowywane w bazie danych. Pomimo iż używa się ich od kilku lat i są bardzo popularne, wydajne oraz pozwalają na lepszą modularyzację systemów, właściwie nie ma dla nich żadnego wsparcia na etapie modelowania. Artykuł przedstawia pewne cechy języka UML, które czynią go przydatnym do modelowania relacyjnych baz danych – nie tylko na etapie konceptualnym i logicznym ale również fizycznym.

Artykuł jest zorganizowany następująco: rozdział 2 przedstawia „state of art" w łączeniu UML z systemami relacyjnymi, rozdział 3 przedstawia możliwości użycia diagramów czynności do modelowania procedur wbudowanych, rozdział 4 prezentuje zastosowania proponowanego podejścia na przykładzie prostego forum dyskusyjnego. Ostatnia część stanowi podsumowanie.

2. State of art

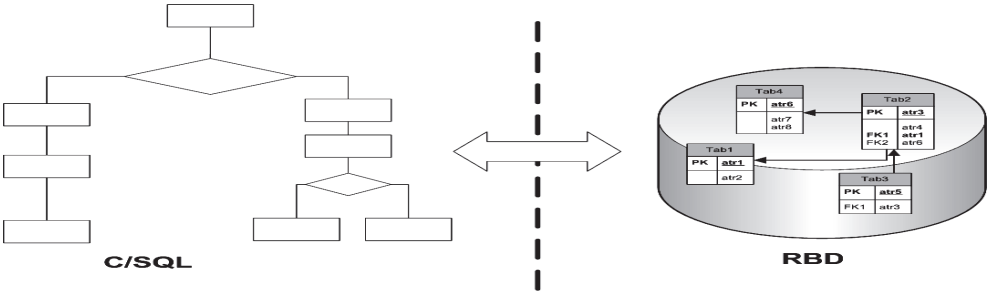
Relacyjny modela danych jest w dalszym ciągu najpopularniejszym logicznym modelem służącym do reprezentowania rzeczywistość – a konkretniej jej fragmentów – zależnych od dziedziny problemu modelowanego systemu. Pomimo że model jest popularny, istnieje mnóstwo dostawców systemów relacyjnych, dowiódł swojej przydatności w tysiącach aplikacji, to jednak stwarza on pewne problemy. Największy z nich wiąże się z reprezentowaniem rzeczywistych obiektów – w tym formalizmie każdy obiekt, taki jak samochód, osoba, budynek itp., nie jest przechowywany w jednej relacji, lecz jest rozproszony na kilka lub kilkanaście relacji, które są ze sobą powiązane. Sytuację tę opisuje rysunek 1.

Podejście takie jest kłopotliwe, ponieważ ludzki umysł postrzega rzeczywistość taką, jaka jest a nie poprzez pryzmat jej składowych. Większość ludzi myśli o budynku, samochodzie, fabryce jako całości, a nie rozważa ich jako połączonych okien, drzwi itd. Konsekwencją tego jest to, że projektant musi na początku dokonać mapowania rzeczywistych obiektów na zbiór tabel, co może nie tylko sprawić kłopot początkującym projektantom, lecz co ważniejsze, powodować zaburzenie integralności danych.



Rys. 1. Model relacyjny i model obiektowy

Na podobne problemy natrafia się również podczas implementacji. Przez ostatnie lata głównym paradygmatem był paradygmat obiektowy. Zatem warstwa biznesowa i prezentacji posługują się często paradygmatem obiektowym, natomiast warstwa danych – relacyjnym. Prowadzi to do pewnych rozbieżności w modelu – część obiektowa była modelowana z użyciem UML, natomiast warstwa danych przy użyciu ERD. Często zachodzi potrzeba prezentacji w warstwie biznesowej danych z relacyjnego modelu – potrzebne jest zatem dostarczenie odpowiedniego mapowania (rys. 2), co powoduje oczywiście zwiększenie nakładu, czasu oraz jest potencjalnym źródłem błędów.



Rys. 2. Mapowanie pomiędzy językiem programowania a systemem relacyjnym

Wprawdzie mapowanie przedstawione na rysunku 2 jest częściowo robione przez platformy developerskie, jednak nie likwiduje to potrzeby tworzenia dwóch modeli – obiektowego dla aplikacji i relacyjnego dla danych. Konsekwencją tego jest to, że w dalszym ciągu

w warstwie aplikacji jednemu obiektowi odpowiada kilka połączonych tabel z warstwy danych.

Częściowym rozwiązaniem takich problemów może być posługiwanie się w warstwie danych obiektową bazą danych lub systemami hybrydowymi – jednak jak wspomniano, nie są one tak popularne i efektywne jak systemy relacyjne. Przewaga systemów relacyjnych jest osiągana dzięki:

- silnym fundamentom matematycznym (logika);
- językowi SQL, który jest prosty, potężny i bardzo popularny;
- OLTP – przetwarzaniu transakcji on-line.

Dodatkowo istnieją czynniki ekonomiczne, które sprawiają, że jeszcze przez długie lata systemy relacyjne będą tworzone. Wszystko to sprawia, że jest silna potrzeba tworzenia metod pozwalających na generację danych relacyjnych z modelu aplikacyjnego.

Ponieważ UML jest właściwie przemysłowym standardem, oczywiste zatem jest, że duża część badań powinna koncentrować się na transformacji modelu UML do postaci systemu relacyjnego. Jednym z najważniejszych diagramów UML jest diagram klas – zawiera on klasy, dzięki którym później tworzone są obiekty reprezentujące rzeczywistość – one z kolei powinny być mapowane do postaci tabel relacyjnych. W związku z tym, dotychczasowe badania [3, 5] skupiają się na metodach generowania tabel na podstawie informacji dostarczanych przez klasy. Generalne zasady, jakie wynikają z tych badań, dotyczące transformacji klasy z modelu konceptualnego do tabeli z modelu logicznego są następujące:

R1. Każda klasa z modelu konceptualnego jest transformowana do tabeli relacyjnej o tej samej nazwie.

R2. Atrybut klasy jest transformowany do kolumny tabeli o tej samej nazwie (jeśli jest możliwe mapowanie od typu atrybutu do typu kolumny).

R3. Tabeli zostaje przypisany klucz główny ze zbioru kluczy kandydujących, które powinny być brane pod uwagę przy modelowaniu konceptualnym. Klucze niewybrane jako klucz główny powinny spełniać więzy SQL: UNIQUE.

R4. Binarna asocjacja typu wiele-do-wielu jest transformowana do relacyjnej tabeli. Klucz główny tej tabeli składa się z kluczy głównych tabel powstałych z transformacji klas wchodzących w skład asocjacji.

R5. Generalizacja jest reprezentowana jako klucz obcy (który jest równocześnie kluczem głównym – jego wartość identyfikuje jednoznacznie rekord w tabeli reprezentującej nadklasę).

Istnieją jednak jeszcze alternatywne sposoby odwzorowania generalizacji:

R5b. Tworzenie osobnych tabeli dla każdej nadklasy i klasy potomnej.

R5c. Tworzenie osobnej tabeli tylko dla klas potomnych.

R5d. Tworzenie jednej tabeli dla całej hierarchii klas.

Każde z tych podejść ma pewne wady polegające między innymi na wprowadzaniu redundantnych danych do tabel.

Sposoby generowania tabel relacyjnych z klas modelu obiektowym były już zdefiniowane dla UML w wersji 1.x. W kontekście nowej wersji języka zaproponowane kilka ciekawych rozszerzeń przedstawionego sposobu transformacji. Rozszerzenia te wykorzystują nowe cechy języka takie jak power-types (nie ma jeszcze przyjętego odpowiednika polskiego tego pojęcia), nowe specyfikatory końców asocjacji ({set},{bag},{list})) oraz rozszerzenie pojęcia generalizacji o jej typy [6]:

{disjoint} – dowolny obiekt nadklasy może być instancją co najwyżej jednej podklasy,

{overlapping} – dowolny obiekt nadklasy może być instancją dwóch lub więcej podklas,

{complete} – każdy obiekt nadklasy musi być instancją przynajmniej jednej podklasy,

{incomplete} – dowolny obiekt nadklasy może nie być instancją dowolnej podklasy.

Bazując na tych właściwościach zaproponowane zostały nowe reguły transformacji [3]. Rozważając typ asocjacji {bag}, regułę R4 można zmodyfikować następująco:

S1. Asocjacja binarna wiele-do-wielu jest transformowana do tabeli relacyjnej. Kluczem głównym tej tabeli jest sztuczny klucz a kluczami obcymi są klucze główne tabel reprezentujących klasy z końców asocjacji.

Transformacja generalizacji zależy teraz od modyfikatorów. Najciekawsze są właściwości: {complete}, {incomplete, overlapping}, {incomplete, disjoint}, {complete, overlapping}, {complete, disjoint}, a reguły transformacji dla nich są następujące:

S2. Nadklasa z jedną podklasą i właściwością {complete} jest reprezentowana przez jedną tabelę, której kolumny stanowią atrybuty obydwu klas.

S3. Nadklasa z wieloma podklasami i właściwością {incomplete, overlapping} jest transformowana zgodnie z regułą R5 – osobna tabela dla każdej podklasy. Wszystkie te tabele powinny mieć taki sam klucz główny.

S4. Nadklasa z wieloma podklasami i właściwością {incomplete, disjoint}, jak wyżej z dodatkową właściwością: zbiór wartości kluczy głównych tabel dla różnych podklas musi być rozłączny.

S5. Transformacja nadklasy z wieloma klasami potomnymi i właściwością {complete, disjoint}zależy od tego czy podklasa jest powiązana z innymi klasami. Jeśli nie jest – wtedy stosowana jest reguła S2 z dodatkowym warunkiem: zbiór wartości kluczy głównych reprezentujących różne podklasy musi być rozłączny. W innych przypadkach stosuje się regułę S4 z dodatkowym warunkiem: każdy wiersz w tabeli reprezentujący nadklasę jest referencjonowany dokładnie przez jeden klucz obcy jednej z tabel reprezentujących podklasę [3].

Jak widać, pewne reguły transformacji zostały zaproponowane już dla UML 1.x (reguły R1–R4), a następnie dodane zostały pewne szczegóły bazujące na nowych cechach UML 2.0 (reguły S1–S5). W rozdziale 3 autor proponuje kolejne reguły bazujące na diagramach czynności, które biorą pod uwagę dynamiczne aspekty działania systemu.

3. Modelowanie zachowania

3.1. Symbole na diagramach czynności

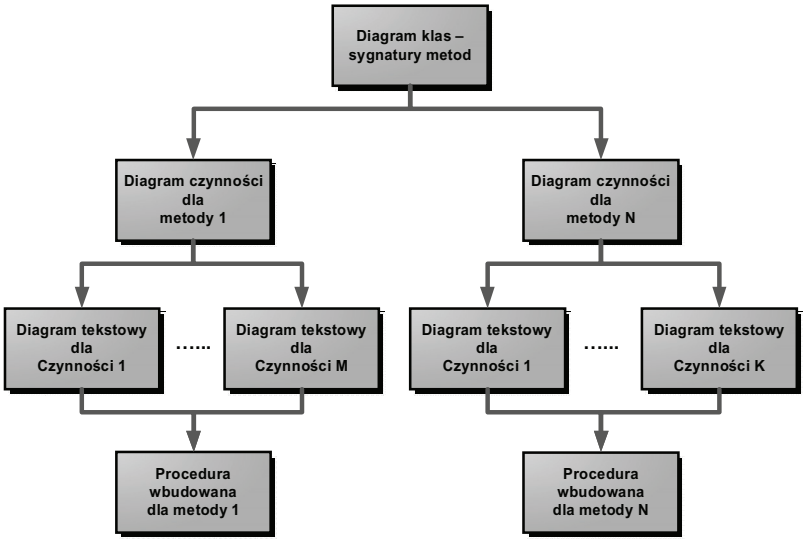
Diagramy czynności posiadają symbol decyzyjny, który umożliwia modelowania złożonych algorytmów za ich pomocą. Inne symbole, które pojawiają się na tych diagramach to:

- symbol czynności reprezentowany przez „prostokąt" z zaokrąglonymi rogami;
- symbol sygnału – kiedy czynność wiąże się z wysłaniem lub odebraniem sygnału, wtedy nazywana jest sygnałem (signal);
- rozwidlenie i złączenie – do modelowania czynności współbieżnych;
- aktywność początkowa i końcowa

3.2. Reguły transformacji

Jak wspomniano w rozdziale 2, klasy są transformowane do relacyjnych tabel natomiast atrybuty do kolumn. Jednak klasy zawierają oprócz atrybutów tez metody, dla których nie dostarczono żadnej transformacji. Najbardziej intuicyjną transformacją jest transformacja ich do procedur wbudowanych i/lub funkcji, ponieważ zarówno procedury wbudowane, jak i metody służą do wykonywania operacji na danych. Jedynym problemem jest ich zakres: metody są związane z klasą, natomiast procedury wbudowane z całym systemem, a nie z pojedynczą tabelą. Aby zapobiec zatem konfliktom nazw (metody różnych klas mogą się nazywać tak samo), autor proponuje prefiksowanie nazwy procedury nazwą tabeli/klasy. Zatem nowa reguła mogła by wyglądać następująco:

R6. Metody klasy są transformowane do procedur wbudowanych przy czym nazwa procedury to nazwa metody poprzedzona nazwą klasy ze znakiem podkreślenia.



Rys. 3. Generowanie kodu SQL z diagramów czynności

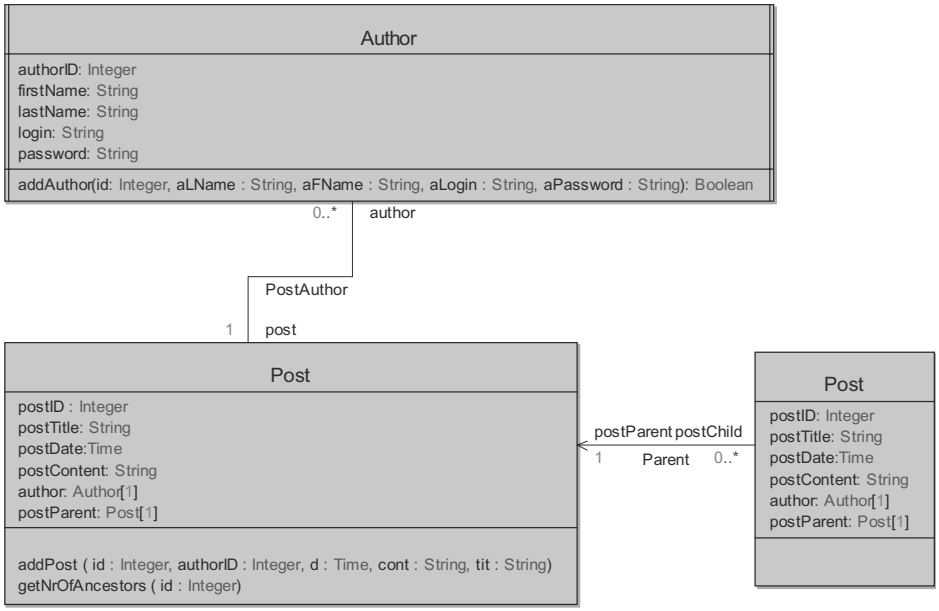
Aby modelować ciało operacji, można się posłużyć diagramami czynności, na podstawie których później może zostać wygenerowany kod SQL. Innymi słowy, można wprowadzić dodatkową warstwę abstrakcji pomiędzy sygnaturę operacji z diagramu klas a wykonywalny kod aplikacji, który bazuje na diagramach czynności. Dzięki temu model staje się bardziej czytelny i ułatwione jest przejście od projektu konceptualnego do fizycznej implementacji.

Autorowi nie są znane żadne nowoczesne narzędzia CASE, które umożliwiałyby generowanie kodu proceduralnego w SQL (większość narzędzi ma możliwość generowania kodu w popularnych językach programowania takich jak Java czy C++) – oczywiście transformacja taka jest bardziej złożona od transformacji do języka obiektowego. W związku z tym można zaproponować ominięcie tego problemu bazujące na diagramach tekstowych. Diagram taki może zawierać dowolny tekst, w tym kod SQL i powinien być kojarzony z każdą czynnością na diagramie (rys. 3).

4. Przykład

Omawiana metodologia zostanie zastosowana do prostego systemu forum dyskusyjnego dla zarejestrowanych użytkowników. Każdy użytkownik jest opisany poprzez następujące atrybuty: imię, nazwisko, login, hasło. Natomiast każdy post poprzez: tytuł, datę wysłania, autora, zawartość oraz informację o tym, na jaki post jest odpowiedzią, jeżeli nie jest nowym wątkiem (każdy post może mieć wiele odpowiedzi natomiast sam może być oczywiście odpowiedzią na tylko jeden post macierzysty).

Przykładowy system może zostać zamodelowany na diagramie klas przedstawionym na rysunku 4.



Rys. 4. Forum dyskusyjne – diagram klas

Stosując dla tego diagramu omówione reguły transformacji, można wygenerować następujący kod SQL tworzący schemat bazy danych:

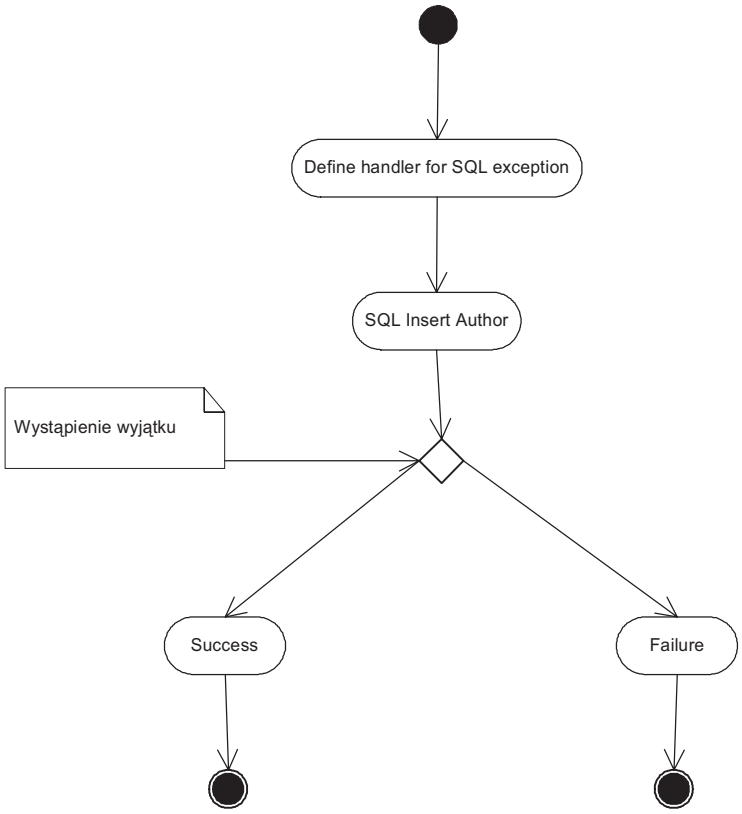
```
CREATE TABLE Author (
   authorID INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
   firstName VARCHAR(45),
   lastName VARCHAR(45),
   login VARCHAR(45),
   password VARCHAR(45),
   PRIMARY KEY(authorID)
);

CREATE TABLE Post (
   postID INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
   parentID INTEGER UNSIGNED,
   authorID INTEGER UNSIGNED,
   postTitle VARCHAR(255),
   postContent TEXT,
   PRIMARY KEY(postID),
   INDEX Post_FKIndex1(authorID),
   INDEX Post_FKIndex2(parentID),
   FOREIGN KEY(authorID)
          REFERENCES Author(authorID)
          ON DELETE SET NULL
          ON UPDATE SET NULL,
   FOREIGN KEY(parentID)
          REFERENCES Post(postID)
          ON DELETE CASCADE
          ON UPDATE CASCADE
);
```

Transformacja ta jest dosyć prosta i szeroko omawiana w literaturze. Jednak diagram klas z rysunku 4 niesie ze sobą więcej informacji. – definiuje przynajmniej trzy operacje: dwie proste do dodawania autora i posta oraz jedną bardziej złożoną do pobierania liczby odpowiedzi na podanego posta. Operacja addAuthor odpowiada za dodanie nowego autora do systemu i zwraca true w przypadku powodzenia i false w przypadku przeciwnym. Sekwencyjne kroki tego procesu można opisać tak:

1. zdefiniuj obsługę jakichkolwiek wyjątków, które mogą wystąpić;
2. wykonaj insert na tabeli relacyjnej;
3. zwróć true, jeśli insert się udał, i false, jeśli nie lub wystąpiły wyjątki.

Przykładowy diagram czynności dla tego procesu przedstawia rysunek 5.



Rys. 5. Diagram czynności dla dodania nowego autora

Rysunek ten jest czytelny i łatwy dorozumienia. Chcielibyśmy jednak mieć możliwość generowania kodu SQL z tego diagramu. Opisana przez autora metodologia polega na dodaniu do każdej czynności diagramu tekstowego z kodem SQL. Zatem diagram dla czynności „Define_handler_for_sql_exception" mógłby zawierać taki kod:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
select «Can not add author»;
SET success = false;
END;
```

Natomiast dla aktywności „activity SQL_Insert_Author" następujący:

```
INSERT INTO Author(authorID, firstName, lastName, login, password)
VALUES
(id, fname, lname, login, pass);
```

W końcu kod dla aktywności “Success”:

```
SET success = true;
select «Author added»;
```

Aby wygenerować z tego diagramu pełny kod procedury powinniśmy przetwarzać czynności w kolejności, w jakiej pojawiają się na diagramie – przetwarzanie takie polega na konkatenowaniu kodu z kolejnych diagramów tekstowych. Zatem pełen kod procedury (sygnatura jest generowana na podstawie sygnatury metody z diagramu klas – dodany zostaje tylko prefiks z nazwą tabeli) może wyglądać następująco (dialekt DB2):

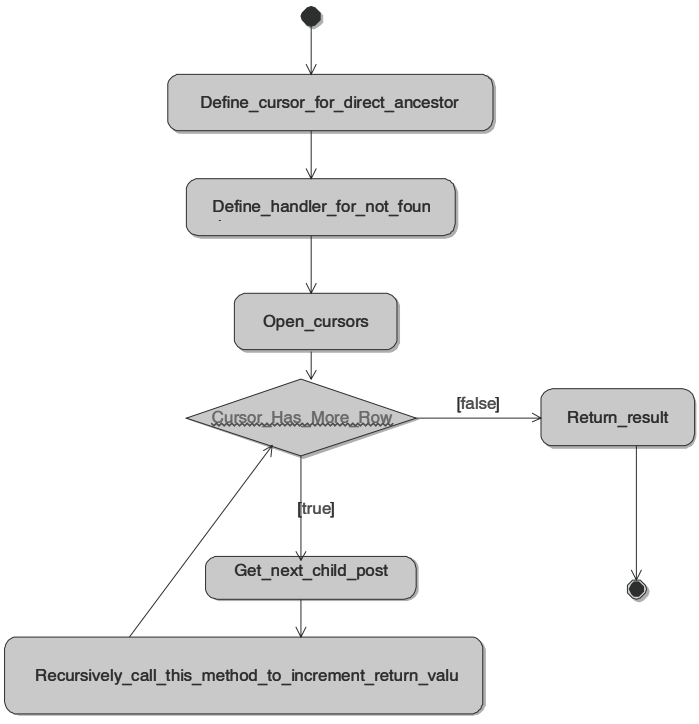
```
CREATE PROCEDURE Author_addAuthor (id INT, fname VARCHAR(45),
lname VARCHAR(45), login VARCHAR(45), pass VARCHAR(45), OUT
success Boolean)
BEGIN
  #Activity: DECLARE EXIT HANDLER FOR SQLEXCEPTION
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
   BEGIN
     select «Can not add author»;
     SET success = false;
   END;
  #Acitivity END
  #Activity: SQL_Insert_Author
  INSERT INTO Author(authorID, firstName, lastName, login,
password) VALUES (id, fname, lname, login, pass);
  #Acitivity END
  #Activity: Success
  SET success = true;
  select «Author added»;
  #Acitivity END
END;
```

Jak widać, dodatkowym wyjściem, jakie powinno być generowane przez narzędzie CASE, są komentarze informujące o tym gdzie się zaczyna, a gdzie kończy dana aktywność/czynność. Przedstawiona metoda jest elegancka i pozwala szybko zrozumieć kod SQL reprezentujący nawet bardzo złożone algorytmy. Kolejny przykład reprezentuje właśnie bardziej złożony algorytm (z rekursją) pobierania liczby odpowiedzi na dany post (rys. 6).

Kod, który może zostać wygenerowany z takiego diagramu (jeżeli zostaną z każdą czynnością skojarzone diagramy czynności) może wyglądać następująco:

```
CREATE PROCEDURE getNrOfAncestors (Post_ID INT, OUT nr INT)
BEGIN
  #Activity: Define_cursor_for_direct_ancestors
  DECLARE id, noMoreRows, children INT;
  DECLARE cur_1 CURSOR FOR
    SELECT postID, parentID FROM Post WHERE parentID=Post_ID;
  #Activity END
  #Activity: Define_handler_for_not_found
```

```
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET noMoreRows = 1;
  #Activity END
  #Activity: Open_Cursors
  OPEN cur_1;
  set nr = 0;
  #Activity END
  #LOOP
  WHILE noMoreRows = 0 DO
    #Activity: Get_Next_Child_Post
    FETCH cur_1 INTO id;
    set nr = nr + 1; #increment for current child
    #Activity END
    #Activity: Recursively_call_this_method…
    call getNrOfAncestors (id, children);
   set nr = nr + children; #increment for children of current child
    #Activity END
  END WHILE;
  #LOOP END
  #Activity: Return_result
  CLOSE cur_1;
  #Activity END
END;
```



Rys. 6. Pobieranie liczby odpowiedzi na post – diagram czynności

4.1. Inżynieria wstecz

Jak wspomniano wcześniej, jeżeli narzędzie CASE generuje odpowiednie komentarze z informacją o początku i końcu czynności, to umożliwia to inżynierię wstecz.

5. Podsumowanie

W artykule zostało zaprezentowane podejście integracji języka UML z systemami relacyjnych baz danych. Zaprezentowana metoda pozwala na generowanie kodu SQL z diagramów UML (klas i czynności) definiującego nie tylko schemat bazy danych, ale również zachowanie systemu: algorytmy i operacje na danych.

Dalsze badania powinny się skupić na:

- dodaniu do istniejących wybranych narzędzi CASE funkcjonalności pozwalającej na generowanie kodu SQL bezpośrednio z diagramów czynności,
- użyciu innych typów diagramów do transformacji – np. diagramów stanów wspomagających diagramy czynności przy modelowaniu operacji.

Literatura

[1] Booch G., Rumbaugh J., Jacobson I.: *The Unified Modeling Language User Guide*. Addison-Wesley, 1998
[2] Douglass B.P.: *Real-Time UML. Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998
[3] Hnatkowska B., Huzar Z., Tuzinkiewicz L.: *Data Modelling with UML 2.0*. IOS Press, 2005
[4] Klimek R., Skrzyński P., Turek M.: *UML i Telelogic Tau Generation2.* Materiały dydaktyczne 2005
[5] Naiburg E., Maksimchuk R.: *UML for Database Design*. Addison-Wesley, 2001
[6] OMG: *Unified Modeling Language: Superstructure version 2.0.* 2004