Antywzorce

Piotr Szwed

pszwed@ia.agh.edu.pl

Katedra Automatyki AGH

Celem dokumentu jest podanie listy typowych błędów, które zaobserwowano podczas realizacji projektów z wykorzystaniem języka modelowania UML.

Lista obejmuje typowe błędy pojawiające się w:

- specyfikacji przypadków użycia
- strukturze klas zidentyfikowanych podczas analizy dziedziny
- strukturze klas dla architektury MVC
- diagramach sekwencji

Przypadki użycia

Przypadki użycia i diagramy przypadków użycia to zazwyczaj osobny rozdział w podręczniku UML [BRJ,WMW]. Dostępne są także dedykowane podręczniki [COCK,SW].

Charakterystyczną cechą podręczników jest chęć przedstawienia składni diagramów PU. Zamieszczone diagramy nie są więc zazwyczaj kompletne z punktu widzenia opisywanego systemu, natomiast zawierają wszelkie możliwe relacje pomiędzy przypadkami użycia.

Zbyt wiele powiązań

W podręcznikach nie ma zbyt wiele wskazówek, w jaki sposób zarządzać przypadkami użycia. Prezentacja gotowych złożonych diagramów skłania przyszłych analityków do budowy ich w podobnej postaci. Diagramy nie zawierające relacji *extend* czy *include* są postrzegane, jako mało zaawansowane. Dlatego, zwłaszcza wstępne propozycje, są przepełnione relacjami, które w żaden sposób nie są odzwierciedlane w opisie scenariuszy. Występuje zależność *include*, pomiędzy dwoma przypadkami użycia: A *include* B, ale ten włączany B nigdzie nie wołany w scenariuszach A. Mamy relację C *extend* B, ale scenariusz C nie pojawia się (studenci mają szczegółowo opisać tylko część przypadków użycia).

Relacje pomiędzy przypadkami użycia powinny być narzędziem zarządzania treścią scenariusza.

- Jeżeli jakiś fragment scenariusza miałby powtarzać się wielokrotnie, wydzielamy go jako PU i wywołujemy, na diagramie pojawia się relacja *include*
- Dotyczy to zwłaszcza fragmentu, który realizuje jakąś dobrze zdefiniowaną i identyfikowaną funkcjonalność, np.: logowanie, rejestracja.
- Jeśli jakiś przebieg alternatywny jest ważny, powinien być widoczny, ma jakąś wartość biznesową, wyłączamy go jako osobny PU, np.: PrzyjęciePłatnościKartą

Proponuje się raczej jednak zacząć od przypadków użycia niepowiązanych relacjami, a potem dokonać refaktoringu. Wydzielamy fragmenty scenariusza w osobny PU i dodajemy relacje.

Poziom i aktorzy

A.Cocburn [COCK] zaproponował rozróżnienie poziomów przypadków użycia (PU). Najbardziej ewidentny jest podział na biznesowe i systemowe PU. Przypadki biznesowe opisują interakcje aktora biznesowego (np.: klienta wypożyczalni DVD) z organizacją (czyli wypożyczalnią razem z jej komputerami, płytami na półkach, pracownikami). Przypadki biznesowe zazwyczaj zawierają także operacje manualne – klient bierze płytę z półki, klient wręcza ją pracownikowi. Pracownik pobiera opłatę.

Kontynuując przykład wypożyczalni, Klient nie jest aktorem systemowego przypadku użycia, jeżeli założymy, że nie ma dostępu do systemu IT wypożyczalni.

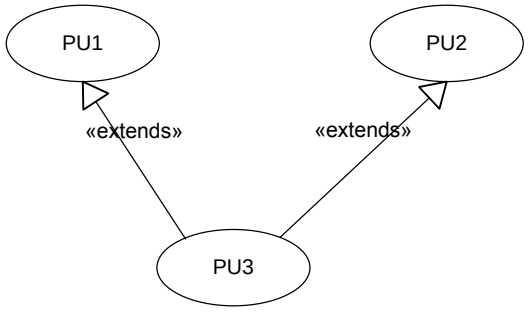
Charakterystyczną cechą systemowych przypadków użycia (czyli realizowanych przez system IT) jest to, że *stosunkowo rzadko występują w nich dwaj aktorzy* typu użytkownik (człowiek). Kolejne kroki powinny być wykonywane w trakcie jednej sesji z systemem IT kończącej się jakimś zdefiniowanym stanem (postwarunkiem). Np.: rejestracja w systemie i aktywacja konta mogą być odległe w czasie, więc raczej wymaga wprowadzenia dwóch systemowych przypadków użycia (a może nawet trzech – automatyczne czyszczenie danych użytkowników, którzy nie dokonali aktywacji konta).

Include Logowanie

Taka konstrukcja pojawia się dosyć często. Ale równocześnie w scenariuszach PU nie ma odwołań do Logowania, natomiast pojawia się prewarunek „użytkownik musi być zalogowany”. Relacja nie jest więc odzwierciedlona w scenariuszu.

Wielokrotne extend

PU nie powinien rozszerzać 2 lub więcej PU Rys. 1, bo semantyka relacji extend jest następująca: jeżeli jest spełniony warunek, zastąp kroki bazowego scenariusza krokami scenariusza rozszerzającego. Należy oczywiście podać warunki (kiedy jest stosowany rozszerzający PU).

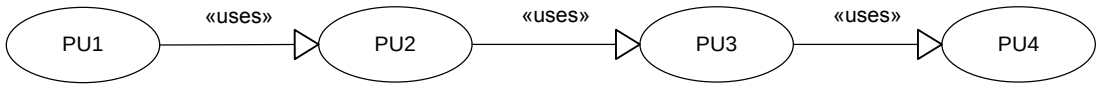


Rys. 1 Wielokrotne extend

Rozszerzenie może być, więc traktowane, jako przebieg alternatywny, trudno wyobrazić sobie, aby jeden przebieg alternatywny dotyczył dwóch scenariuszy głównych różnych przypadków użycia.

Łańcuch include

Kroki scenariusza zastąpione łańcuchem include.



Rys. 2 Łańcuch include

Analiza scenariuszy pokazywała z reguły, że były bardzo krótkie, w zasadzie składały się z 1-2 kroków. Jest to oczywiście błędne rozwiązanie, należy scalić je w jeden przypadek użycia, bo dopiero wykonanie całego łańcuch prowadzi do osiągnięcia jakiegoś celu istotnego z punktu widzenia aktora.

Przypadki użycia niespecyfikujące wymagań

Wyobraźmy sobie następująco opisany przypadek użycia

Scenariusz główny

1. System wyświetla formatkę wprowadzania *danych*
2. Aktor wprowadza *dane*
3. System weryfikuje *dane*
4. System wyświetla potwierdzenia

Scenariusz alternatywny:

3.a. Wprowadzono błędne *dane*

3.a.1. System wyświetla informacje o błędzie

3.a.2. Następuje przejście do punktu 2 scenariusza głównego

Co tu jest błędem? W taki sposób można opisać niemal każdą akcję użytkownika w systemie. W zasadzie opis jest poprawny. Wystarczy go przekopiować i mamy 10 przypadków użycia.

Przypadki użycia mają opisywać wymagania oraz być źródłem przypadków testowych. Jak zaimplementować formatkę wprowadzania danych, jeśli ich zawartość nie jest znana? Musimy określić, *jakie dane* są wprowadzane.

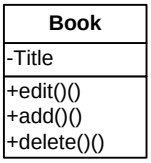
Jeżeli już zdefiniowaliśmy postać i rodzaj danych, to należy uściślić: *jakie dane lub kombinacje danych są poprawne*, jakie warunki muszą być spełnione, aby pomyślnie przejść punkt 3 lub wybrać przebieg alternatywny. Właśnie to będzie przedmiotem testów akceptacyjnych dla systemu. Jeżeli wymagane jest, aby system reagował na określone błędy w danych, np.: błąd sumy kontrolnej numeru konta bankowego, to testujemy gotowy produkt podając błędny numer i obserwujemy, ja system zachowa się.

Analiza dziedziny

Celem analizy dziedziny jest identyfikacja struktury klas występujących w dziedzinie problemu bez ich uwarunkowań technologicznych.

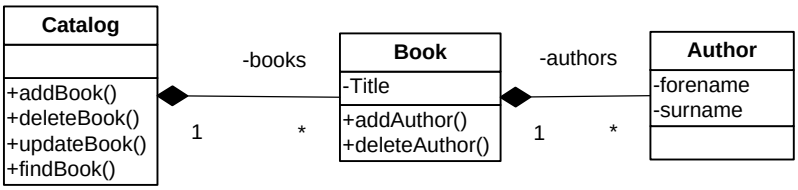
Metody CRUD w klasach dziedziny

W klasach dziedziny pojawiają się metody `dodaj(), edytuj(), usuń()`. Nie mają sensu, bo są to operacje na kontenerze obiektów, np.: bazie danych lub wymagają dodatkowych obiektów – np.: wyświetlenia formatki edycji, czyli obiektu klasy widoku.



Rys. 3 Metody CRUD w klasie. Jak to możliwe, czy książka dodaje się do siebie, czy umie się usunąć?

Identyfikując i rozmieszczając metody należy przyjąć, że metoda klasy pozwala na zmianę stanu obiektu lub wykonanie usługi, do której niezbędny jest dostęp do zawartości obiektu (lub obiektów składowych struktury całość-część). Książka może dodać autora, jeżeli przewidzimy tu agregację pomiędzy klasami Book i Author, natomiast katalog dodać książkę (Rys. 4).



Rys. 4 Prawidłowe rozmieszczenie metod, operacje dodawania i usuwania są metodami kontenera i działają na jego zawartości

Wydaje się, że źródłem tego typu błędów może być wcześniejszy przebieg kursu obejmujący diagramy ERD. Diagramy ERD specyfikują zależności pomiędzy encjami, nie rozróżniając jednak kontenerów i umieszczonych w nich obiektów. Równocześnie, mają podobną siłę wyrazu, jak diagramy klas: możliwe jest podanie atrybutów oraz zapisanie relacji asocjacji, agregacji i dziedziczenia. Przejście od diagramu ERD do diagramu klas jest stosunkowo prostym zabiegiem, wiele informacji przenosi się bezpośrednio z jednego modelu do drugiego; niestety przy okazji często błędnie rozmieszczane są metody.

Aktor vs. klasa

Bardzo często aktor systemu i klasa przechowująca informacje o aktorze (na przykład jego dane adresowe i dane logowania) mają nadane te same nazwy. Podczas modelowania związków pomiędzy klasami wprowadzane są asocjacje, które powinny być rzeczywiście przechowywane w systemie, ale także relacje odzwierciedlające *rolę aktora* podczas wykonywania pewnych czynności.

Czytelnik pożycza książkę. Jest to poprawna relacja zapisywana trwale w systemie. Chcemy wiedzieć, kto ma obecnie wypożyczoną książkę, a nawet przechować takie informacje w historii wypożyczeń.

Czytelnik oddaje książkę. To prawda, aktor biznesowego przypadku użycia oddaje książkę, ale najczęściej system nie przechowuje informacji o tym, kto oddał książkę. Raczej wystarcza informacja, że książka została zwrócona. Relacja jest więc błędna i nadmiarowa. Natomiast możemy chcieć zmienić stan wypożyczenia (książka zwrócona).

Systemy stosunkowo rzadko przechowują informację o tym, kto wykonał określoną czynność. Tego typu wymagania mogą dotyczyć podania wystawcy faktury, operacji bankowych, gdzie kontrolowany jest każdy dostęp do danych, natomiast nie występują w przeważającej większości aplikacji.

Warto zwrócić uwagę, że podobny efekt pojawia się podczas modelowania za pomocą diagramów ERD. Często wykorzystywany podręcznik

[http://yourdon.com/strucanalysis/wiki/index.php?title=Chapter_12] zawiera przykłady, w *których relacje są czasownikami*: Nabywca kupuje Przedmiot, Lekarz leczy Pacjenta i Lekarz obciążaRachunkiem Pacjenta. Te relacje istnieją tylko wtedy, jeżeli system powinien je trwale zapisywać.

Nadmiarowe asocjacje

Metoda Coad-Yourdon [CY] promowała stosowanie relacji asocjacji w każdym przypadku, kiedy klasy do realizacji zadań potrzebują informacji o innych klasach. W UML to zostało uściślone, pojawiła się zależność. Nie należy stosować asocjacji, jeżeli wystarcza zależność.

Jeżeli dokument drukuje się na drukarce, być może nie jest konieczna asocjacja dokument – drukarka, zwłaszcza, jeśli mamy metodę klasy Dokument drukuj(drukarka).

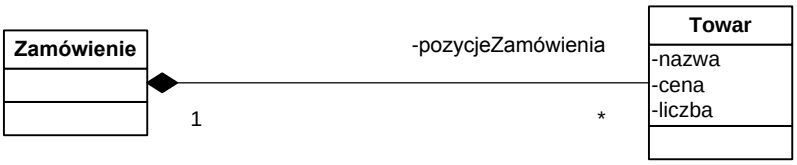
Asocjacja powinna zapisywać związek, który w trwały sposób jest przechowywany w systemie. Jeśli dokument pamięta, że używa określonej drukarki, to jest to asocjacja, jeżeli nie to tylko zależność.

Dla porównania: dokument MS Word pamięta, jakiej używa drukarki, bo został przeformatowany tak, aby zapewnić zgodność ekranu z wydrukiem. Dokument TeX nie.

Brak klas asocjatywnych

Częstym błędem jest brak klas pośredniczących w realizacji asocjacji.

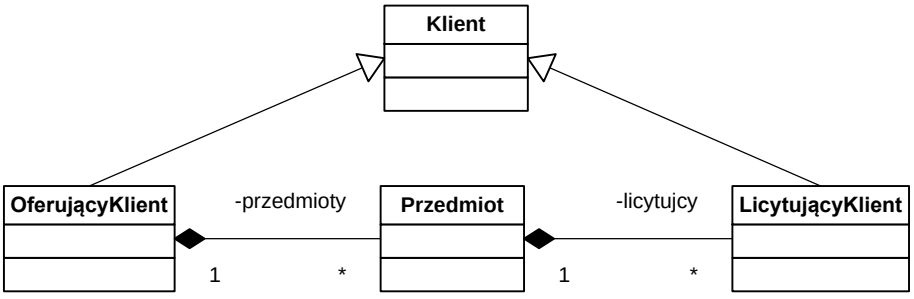
Zamówienie w agregacji lub asocjacji z klasą Towar. Gdzie jest zapisywana ich liczba, cena?



Rys. 5 Brak klasy realizującej asocjację.

Czy ta liczba to element pozycji zamówienia, czy stan magazynu? Należy dodać klasę PozycjaZamówienia będącą w asocjacji z towarem.

Kolejny przykład: Klient wystawia na aukcji Przedmioty (agregacja) Przedmioty maja licytujących klientów (agregacja) . Powstałoby drzewo, które nie jest założonym modelem systemu.



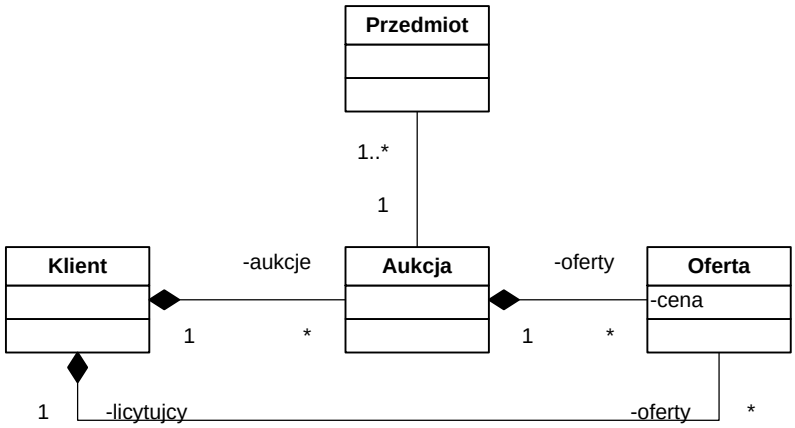
Rys. 6 Brak klasy realizującej asocjację. Błędne odzwierciedlenie roli w hierarchii klas

Błędem jest brak klasy pełniącej rolę pośrednika w asocjacji pomiędzy sprzedawanym przedmiotem a potencjalnymi nabywcami. Na przykład tą klasą może być Oferta. Przedmiot ma oferty kupna wystawiane przez klientów.

Pomieszanie roli i klasy

Rys. 6 pokazuje jeszcze jeden błąd: rola (oferent, nabywca) staje się kryterium rozróżnienia pomiędzy podklasami. W konsekwencji klient może być albo wyłącznie `OferującymKlientem`, albo `LicytującymKlientem`. Jest niezbyt funkcjonalne obejście tego problemu – klient może zarejestrować się w systemie dwa razy.

Ostatecznie, raczej proponowany jest model, jak na poniższym rysunku.



Rys. 7 Proponowany prawidłowy model

Architektura MVC

Istotą wzorca jest rozdzielenie elementów architektury odpowiedzialnych za wizualizację danych, ich przetwarzania oraz ich przechowywanie. Typowy przebieg interakcji w tym modelu to:

> Aktor dokonuje interakcji z widokiem, np.: naciska przycisk '*wyświetl szczegóły towaru*'. Obiekt widok identyfikuje, o jaki towar chodziło aktorowi i przesyła informacje do kontrolera, który dalej propaguje wywołanie do warstwy modelu, po otrzymaniu informacji zwrotnej uaktualnia widok (lub widoki).

W modelowaniu obiektowym elementów architektury wzorca MVC należy doszukiwać się klas i obiektów. W rezultacie powinny zostać zidentyfikowane klasy widoku, kontrolera i modelu.

- Model: klasami są np.: tabele bazy danych, albo cała baza danych widziana jak pojedynczy obiekt (singelton) pewnej klasy.
- Kontroler: na przykład klasami są skrypty (w przypadku aplikacji internetowych)
- Widok: zazwyczaj to okna, formatki, ale w przypadku aplikacji internetowych są to strony HTML

Brak widoku

Problem dotyczy zwłaszcza projektów dla aplikacji internetowych. Dla programisty często strona HTML nie jest obiektem, ale po prostu generowanym tekstem. Skoro strona nie jest widokiem, jako widok identyfikowany jest skrypt generujący stronę.

Jest to błędne podejście. Strona HTML powinna być traktowana, jako obiekt pewnej klasy. Może mieć metody (kod w JavaScript, kod Ajax, kod wywołania kontrolera) może mieć atrybuty może składać się z podstron, sekcji, itd. (agregacja).

W projekcie identyfikacja klas stron HTML jest ważna, ponieważ pozwala oszacować nakład pracy na ich implementację:

- Dla każdej strony musi istnieć fragment kodu, który ją wygenerował
- Jeśli osadzono w niej kod, to musi być on zaimplementowany i przetestowany
- Zapewne ma grafikę, którą ktoś zaprojektuje

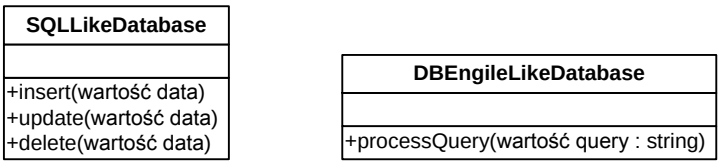
Błędy w modelu

Ta warstwa architektury może być w różny sposób zamodelowana i w jakiś sposób zależy to od przyjętej platformy i metody opisu.

Możliwe jest szerokie spektrum rozwiązań pomiędzy pojedynczym monolitycznym obiektem reprezentującym bazę danych i operacje na bazie danych, a diagramami o bardzo dużej granulacji, w której pojawiają się odpowiedniki obiektów dziedziny, obiekty odpowiadające tabelom bazy danych, obiekty dla transakcji, w których uaktualniana jest równocześnie zawartość kilku tabel. To drugie rozwiązanie jest bardziej naturalne dla systemów wspierających ORM.

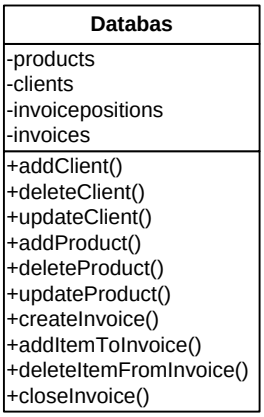
Zdecydowana większość projektów, jak dotąd, raczej wykorzystywała monolityczny singelton opakowujący BD

Porównajmy projekty klas zamieszczonych na Rys. 8. W przypadku klasy `SQLLikeDatabease` zidentyfikowane metody to polecenia SQL działające na bliżej nieokreślonych danych. W przypadku `DBEngineLikeDatabase` jedyną zidentyfikowaną metodą jest `processQuery(String)`, która wszelkie niuanse oczekiwanej funkcjonalności sprowadza do tekstu kwerendy generowanej po stronie kontrolera.



Rys. 8 Baza danych z metodami odpowiadającymi poleceniom interpretera SQL w miejsce dedykowanych metod

Obie wersje są niwłaściwe, ponieważ nie specyfikują, jakich usług oczekujemy od zaimplementowanej bazy danych. Nie są też punktem odniesienia do ewentualnej specyfikacji tych usług, na przykład w formie prewarunków i postwarunków.



Rys. 9 Dedykowana baza danych z dedykowanymi metodami

Przedstawiona na Rys. 9 wersja jest zdecydowanie lepsza, ponieważ definiuje metody obiektu niezbędne do implementacji systemu. Metody te mogą być wyspecyfikowane w sposób opisowy lub jako prewarunki i postwarunki.

boolean deleteProduct (produkt: Produkt) :

PRE: rekord r z kluczem r.id = produkt.id jest obecny w tabeli products

POST: oznacz r.deleted=true i zwróć true

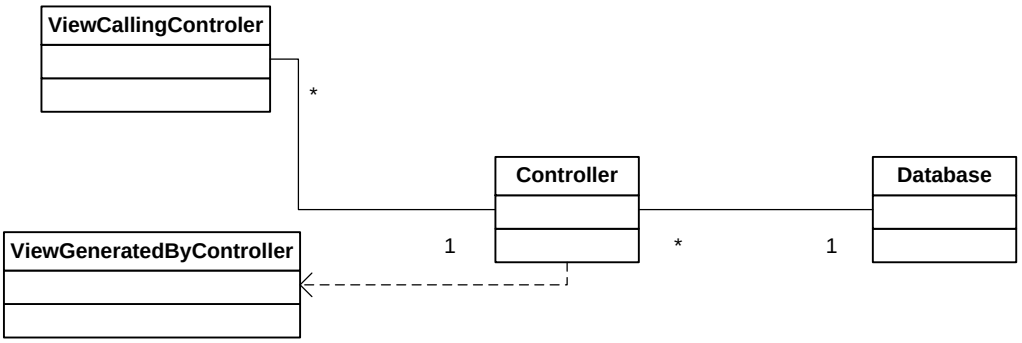
PRE: rekord r z kluczem r.id = produkt.id nie jest obecny w tabeli products

POST: zwróć false

Powiązania w MVC

Aby widok mógł wywołać kontroler- musi znać jego lokalizację. Potrzebna jest asocjacja. Podobnie kontroler może być w asocjacji z bazą danych.

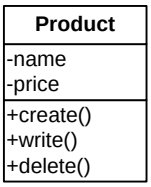
Jeżeli kontroler generuje stronę HTML (obiekt widoku) to jest to raczej zależność, kontroler musi wiedzieć, jak ją wygenerować, znać typ, ale nie musi być bezpośrednio powiązany z obiektem tej klasy.



Rys. 10 Relacje pomiędzy klasami MVC dla aplikacji internetowej

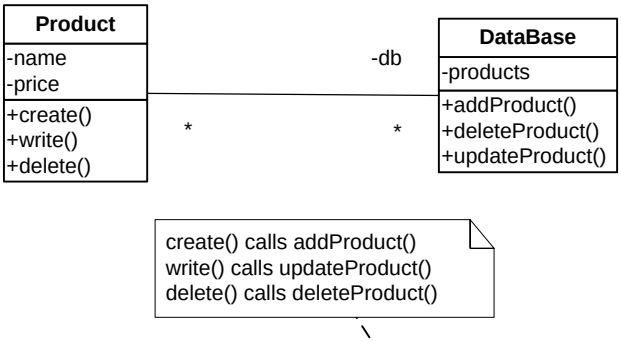
Pomieszanie klas dziedziny i kontrolera

W architekturze MVC pojawiają się nagle klasy o strukturze relacji typowej dla klas dziedziny. Mają odpowiednie atrybuty i metody CRUD (Rys. 11).



Rys. 11 Metody CRUD w klasie

Dopiero analiza dołączonego kodu wskazuje, że są to raczej klasy kontrolera dla projektu wykorzystującego np.; język PHP. Atrybuty pochodzą z przetwarzanego w konstruktorze zapytania, natomiast metody, które należałoby raczej przetłumaczyć na `zapiszSię()`, itd. to manipulacje na BD wyrażone jako zapytania SQL (Rys. 12).



Rys. 12 Wynik analizy kodu dla metod CRUD

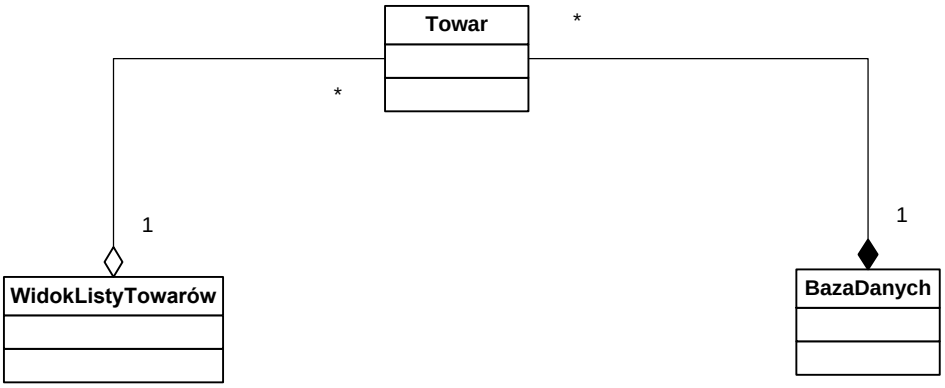
Taka konstrukcja jest dopuszczalna, nawet jest charakterystyczna dla ORM, ale problem w tym, że na diagramie wskazano także rozmaite relacje pomiędzy obiektami, które występują pomiędzy obiektami dziedziny, ale których wcale manualnie utworzony kod nie implementuje.

Brak powiązania widoku z klasami dziedziny

Widok ma prezentować zawartość modelu i umożliwiać manipulacje na modelu. Model z kolei najczęściej jest kontenerem klas zidentyfikowanych w dziedzinie problemu.

Oznacza to, że najczęściej trzeba wprowadzić asocjację pomiędzy klasą widoku, a klasami dziedziny. Bez tej asocjacji nie byłoby możliwe przekazanie informacji do kontrolera, że wymagana jest operacja na konkretnym obiekcie.

W przypadku widoków w postaci stron HTML za asocjację odpowiada np.: całkowitoliczbowy identyfikator obiektu w bazie danych osadzony na stronie.

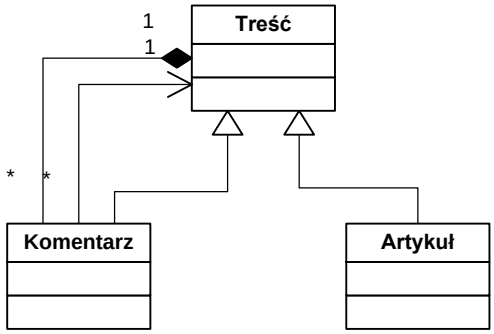


Rys. 13 Powiązanie widoku z klasami dziedziny

Rys. 14 Powiązanie złożonego widoku z klasami dziedziny

Hierarchiczny widok dla drzewa

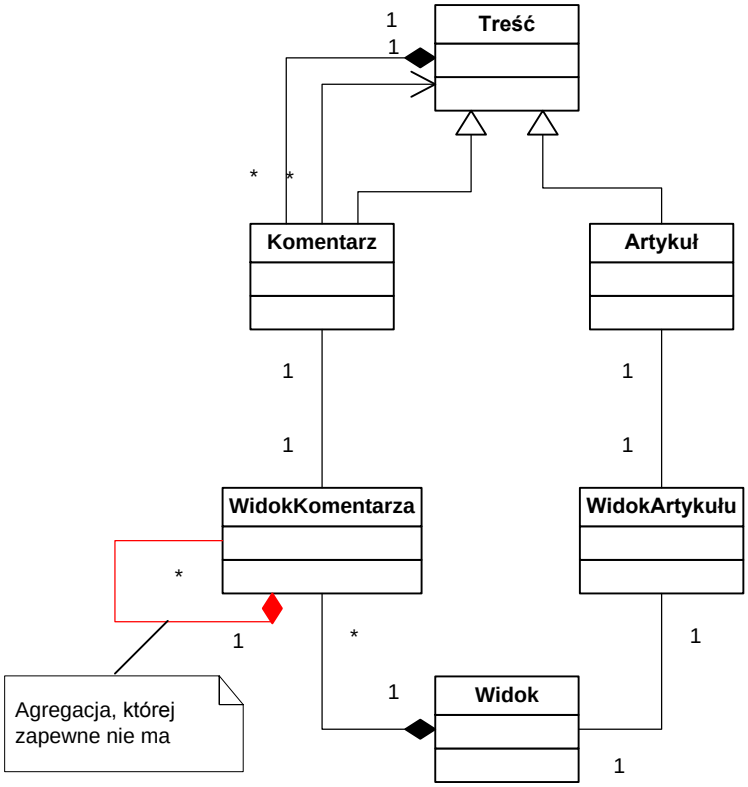
Wyobraźmy sobie strukturę klas dziedziny generującą drzewo obiektów



Rys. 15 Struktura klas dziedziny generujące drzewo obiektów

Dowolna treść (artykuł lub komentarz) może mieć komentarze.

Relacje pomiędzy klasami dziedziny (drzewo) nie zawsze przenoszą się na widok, który je wyświetla. Widok zawiera elementy, które może są tak sformatowane, aby przypominały drzewo, ale są zdecydowanie strukturą liniową (tabelą listą). Aby stwierdzić, że tak jest warto zajrzeć do kodu…



Rys. 16 Nieistniejące drzewo. Widok komentarza najczęściej nie zawiera widoków komentarza.

Błędem jest przeniesienie drzewiastej struktury klas dziedziny na widok.

Diagramy sekwencji

Typową część projektu stanowią diagramy sekwencji pokazujące interakcje pomiędzy obiektami należącymi do klas widoku, kontrolera i modelu.

Przyjmując perspektywę aplikacji internetowych: jeżeli wykluczymy technologie pozwalające na komunikację bezpośrednią serwera z klientem, (np.: aplety, Reverse Ajax) i wykorzystamy wyłącznie bazowe rozwiązania, wówczas interakcja z systemem odbywa się według powtarzalnego scenariusza

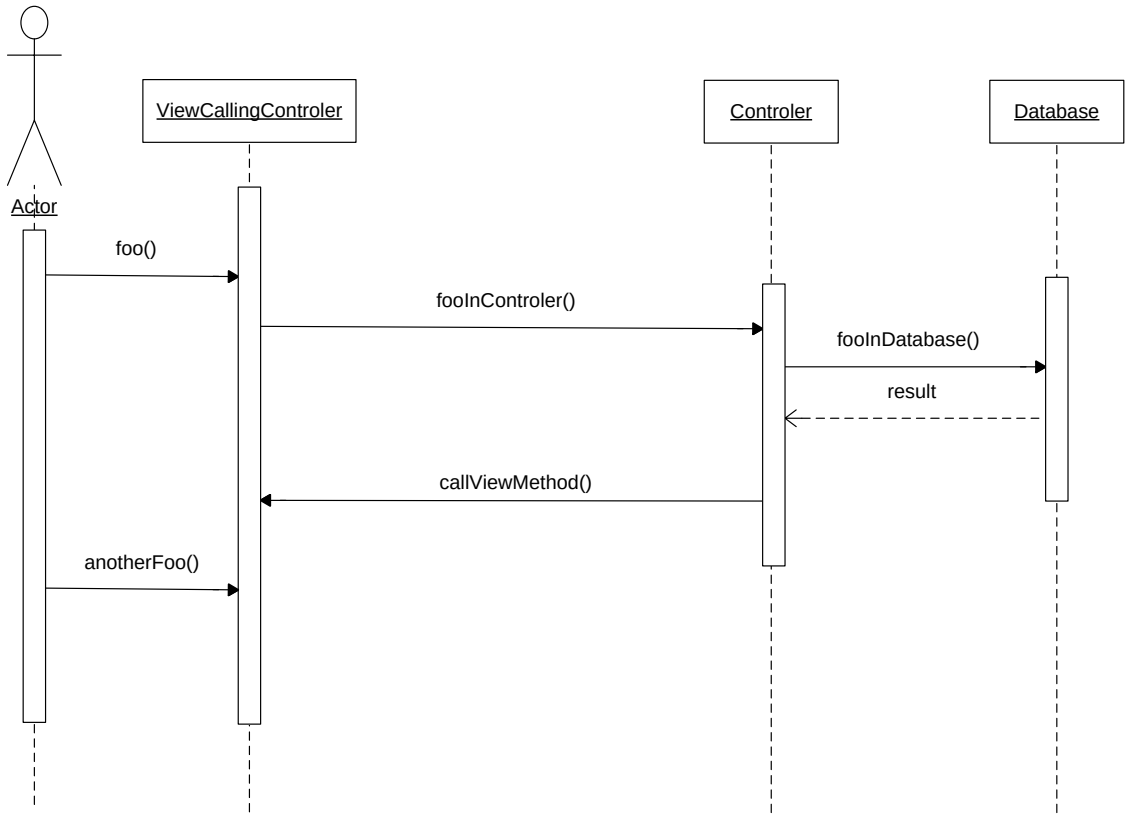
- aktor oddziałuje na widok (stronę HTML)
- obiekt widoku woła kontroler,
- ten realizuje dostęp do obiektu bazy danych i
- generuje **nowy** obiekt widoku.

Należy zachować zgodność z technologią. Kontroler nie ma możliwości wywoływania metody widoku, który go uaktywnił, ale zawsze tworzy nowy obiekt. Do starego można wrócić w przeglądarce naciskając przycisk Wstecz!

Przede wszystkim, aby wywołać metodę widoku, konieczna byłaby jakaś forma asocjacji pomiędzy kontrolerem a widokiem. Asocjacja ta może przybierać formę połączenia

sieciowego (gniazda) ale do tego potrzebny jest jakiś obiekt aktywny osadzony w widoku (aplet, kod Ajax)

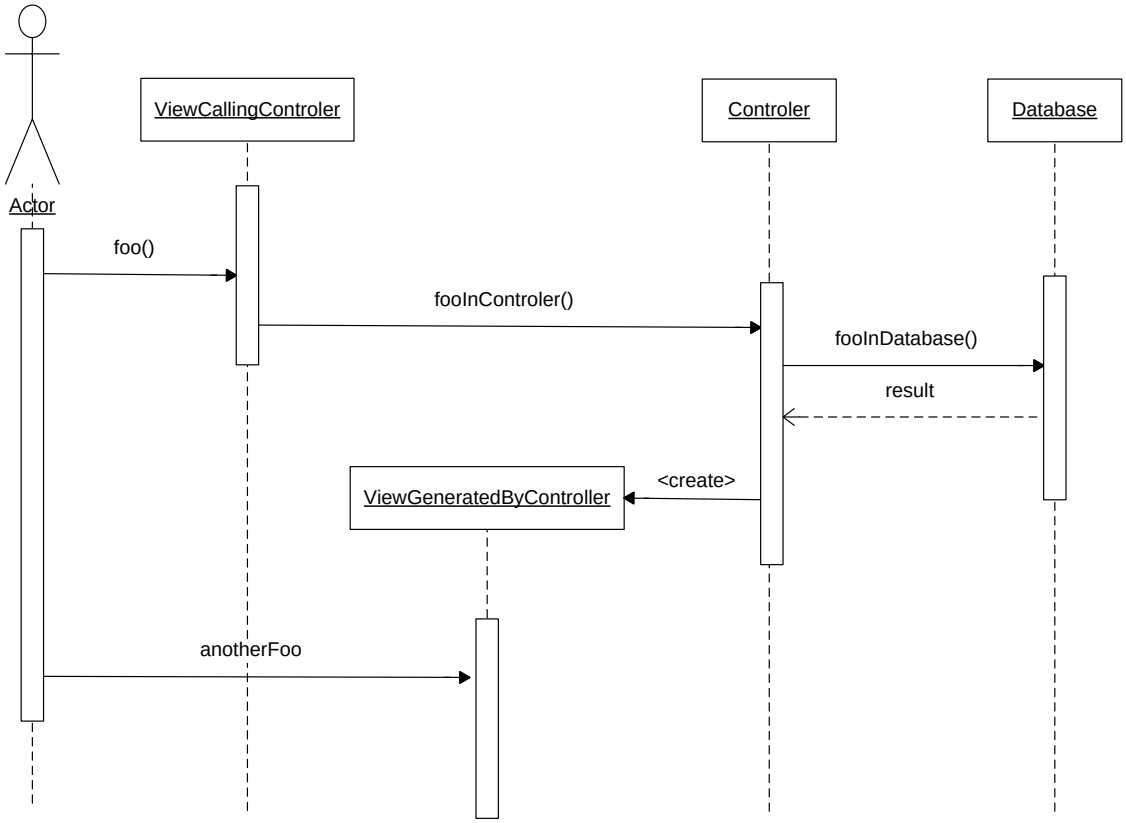
Rys. 17 Pokazuje rozwiązanie, które jest błędne. Standardowa technologia nie pozwala na wywołanie metody widoku.



Rys. 17 Kontroler wołający metodę widoku? Możliwe w technologii apletów, JavaFX, Reverse Ajax, ale nie standardowym modelu cienkiego klienta

Pewną odmianą tego podejścia jest wywołanie przez kontroler metody `refresh()` widoku; oczywiście takiej metody nie ma, ale jest to jakaś próba przekazania informacji, że kontroler odświeża stronę. To rozwiązanie jest znowu błędne – raczej niezbędna jest metoda `refresh()`, ale kontrolera, która będzie w stanie wygenerować kolejny widok, na przykład dla zmienionych danych.

Poprawne rozwiązanie pokazano na Rys. 18.



Rys. 18 Kontroler generujący nowy widok. Standardowe zachowanie dla aplikacji internetowych

Literatura

[BRJ] Grady Booch, James Rumbaugh, Ivar Jacobson, UML przewodnik użytkownika

[WMW] Stanisław Wrycza, Bartosz Marcinkowski, Krzysztof Wyrzykowski, Język UML 2.0 w modelowaniu systemów informatycznych

[CY] Peter Coad, Edward Yourdon, Analiza obiektowa, Projektowanie obiektowe, 1991

[COCK] Alistair Cockburn, Jak pisać efektywne przypadki użycia

[SW] Geri Schneider, Jason P. Winters, Stosowanie przypadków użycia