Head First EJB. Edycja polska

Autorzy: Kathy Sierra, Bert Bates
Tłumaczenie: Rafał Jońca, Piotr Rajca
ISBN: 83-7361-548-2
Tytuł oryginału: Head First EJB
Format: 200×234, stron: 720

EJB (Enterprise JavaBeans) to technologia najczęściej wykorzystywana do tworzenia aplikacji opartych na komponentach. Aby ją efektywnie wykorzystywać, musisz zgłębić jej podstawowe założenia, dowiedzieć się, na jakie typy dzielimy komponenty, jak działają mechanizmy transakcji i do czego służą wzorce projektowe. Przeraża Cię to? Niepotrzebnie. Otwórz swój umysł. Poznaj technologię EJB w sposób gwarantujący jej szybkie i skuteczne opanowanie. Zapomnij o listingach liczących tysiące wierszy i długich, nużących opisach teoretycznych. Czytając książkę „Head First EJB. Edycja polska", poznasz technologię EJB w ciekawszy sposób.

Dzięki tej książce wszystkie pojęcia związane z EJB przestaną być dla Ciebie wiedzą tajemną. Autorzy książki, wykorzystując najnowsze elementy teorii uczenia, przedstawią Ci wszystkie zagadnienia niezbędne do rozpoczęcia projektowania i tworzenia aplikacji w technologii EJB. Poznasz architekturę EJB, cykle życia komponentów entity bean, session bean i message-driven bean, CMP, EJB-QL, transakcje, bezpieczeństwo, wzorce i ogólne idee tworzenia aplikacji opartych na komponentach. Jednak, co najważniejsze, nauczysz się stosować tę wiedzę w praktyce.

W książce poruszono między innymi następujące tematy:

- Architektura aplikacji EJB
- Typy komponentów
- Tworzenie i stosowanie komponentów session bean oraz entity bean
- Powiązania pomiędzy komponentami
- Połączenia z bazą danych
- Komunikaty
- Obsługa wyjątków w komponentach
- Tworzenie mechanizmów autoryzacji
- Wdrażanie aplikacji EJB

Przekonaj się, że nawet przy poznawaniu skomplikowanych technologii można się świetnie bawić.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści (podsumowanie)

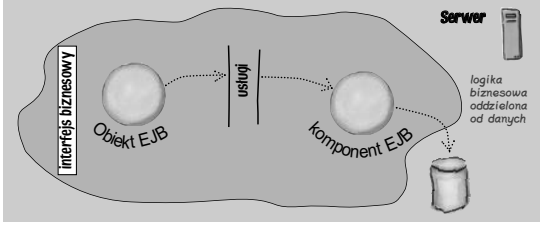
Spis treści (szczegółowy)

Wprowadzenie

Twój mózg jest skoncentrowany na EJB. W tym rozdziale *Ty* starasz się czegoś dowiedzieć, a Twój *mózg* robi Ci przysługę i nie przykłada się do *zapamiętywania* zdobywanej wiedzy. Twój mózg myśli sobie: „Lepiej zostawię miejsce w pamięci na bardziej istotne informacje, na przykład: jakich dzikich zwierząt należy unikać bądź czy jeżdżenie nago na snowboardzie jest dobrym pomysłem". A zatem, w jaki sposób możesz przekonać swój mózg, że Twoje życie zależy od poznania EJB?

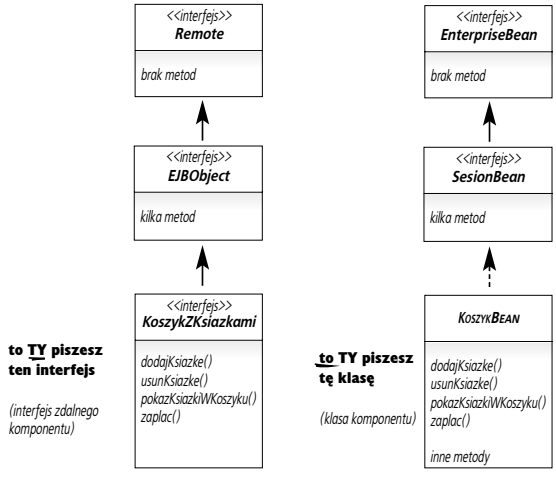
1 Zapraszamy do EJB

Komponenty Enterprise JavaBean są łatwe. Przynajmniej jeśli porównamy je z tym, co trzeba by napisać, aby własnoręcznie stworzyć skalowalny, transakcyjny, bezpieczny, trwały i współbieżny serwer komponentów korporacyjnych. W tym rozdziale stworzymy, wdrożymy i uruchomimy aplikację EJB, a następnie opiszemy jej szczegóły. Nim skończymy, dowiesz się także jakie zalety ma technologia EJB, jakie są jej cechy charakterystyczne i poznasz (pobieżnie) zasady działania kontenerów EJB.



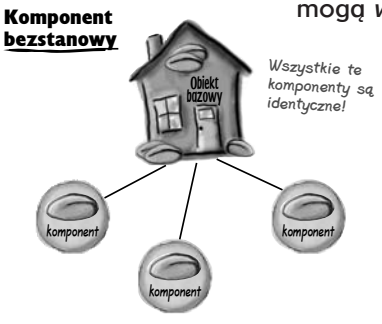
2 Architektura EJB

Technologia EJB jest związana z infrastrukturą. Komponenty są elementami konstrukcyjnymi. Technologia ta umożliwia tworzenie bardzo dużych aplikacji. Aplikacji, które pozwalają na niemal wszystko, począwszy od obsługi firmy Victoria's Secrets, a skończywszy na zarządzaniu wszystkimi dokumentami w centrum badawczym CERN. Niemniej jednak architektura rozwiązania o takiej elastyczności, mocy i skalowalności nie jest prosta. Wszystko zaczyna się od modelu programowania rozproszonego…



3 Ujawnianie się

Swoich komponentów nie możesz zachować tylko dla siebie. Klienty muszą mieć dostęp do istniejących komponentów. (Nie dotyczy to komponentów *message-drive bean*, które *nie dysponują* widokiem klienta.) Nasz `Komponent Doradca` udostępnia metodę `getPorada()` wchodzącą w skład interfejsu komponentu, stanowiącego miejsce, w którym są deklarowane metody biznesowe. Jednak to jeszcze *nie wszystko*, co klienty mogą zobaczyć i do czego mają dostęp. Pamiętaj, że interfejs `Doradca` dziedziczy po interfejsie `EJBObject`, który posiada *własne* metody. Metody, do których klienty mają dostęp. Metody, które klienty mogą *wywołać*. To samo dotyczy interfejsu bazowego.



W przypadku bezstanowych komponentów *session bean* utworzonych przez ten sam obiekt bazowy metoda `isIdentical()` zawsze zwraca wartość *true*, nawet dla różnych komponentów.

4 Być komponentem session bean

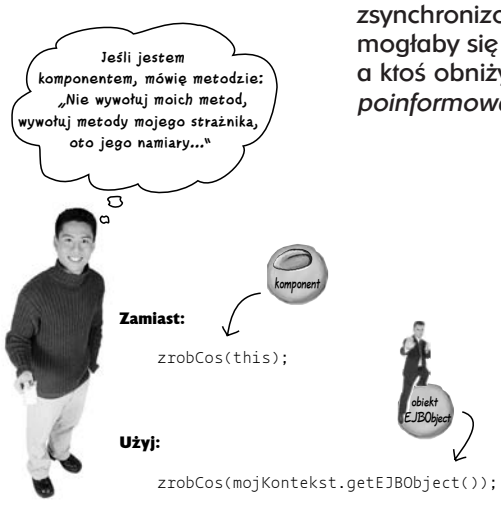
Komponenty *session bean* są tworzone i usuwane. Jeśli masz szczęście, to jesteś komponentem *bez*stanowym. Szczęście, gdyż istnienie komponentów *stanowych* jest całkowicie zależne od kaprysów nieczułych klientów. Są one tworzone na żądanie klienta, a *jedynym* sensem ich istnienia i śmierci jest służenie temu klientowi. Ale co tam, życie komponentu bezstanowego jest ***cudowne***! Baseny, drinki z tymi malutkimi parasolami i żadnej nudy, bo wciąż spotyka się tak wiele różnych klientów.

5 Encje są trwałe

Komponenty *entity bean* są trwałe. Komponenty *entity bean* istnieją. Komponenty *entity bean* po prostu są. Stanowią one obiektową reprezentację informacji pochodzących z ***trwałego magazynu***. (Wyobraź sobie, że to ***baza danych***, gdyż większość komponentów *entity bean* reprezentuje informacje pochodzące z relacyjnych baz danych.) Jeśli dysponujesz komponentem *entity bean* `Klient`, to jeden taki komponent może reprezentować encję Jan Kowalski, ID #342, a inny encję Dorota Miśkiewicz, #90. Trzy komponenty, reprezentujące trzy faktycznie istniejące encje. Komponenty *entity bean* są po prostu *realizacją* czegoś, co już istnieje.

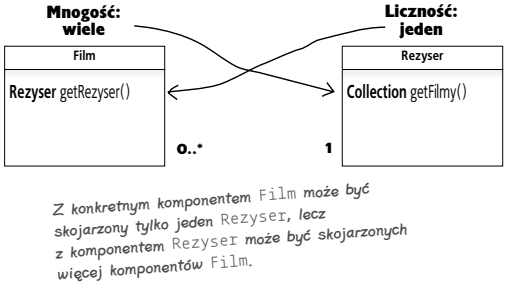
6 Być komponentem entity bean

Komponenty *entity bean* są aktorami. Gdy istnieją, przebywają bądź w puli, bądź *stają się* kimś. Kimś z ukrytego trwałego magazynu (encjami pochodzącymi z bazy danych). Kiedy komponent wciela się w czyjąś postać, musi być cały czas zsynchronizowany z reprezentowaną encją. Wyobraź sobie jaka katastrofa mogłaby się wydarzyć, gdyby komponent udawał, na przykład, Janka Zimocha, a ktoś obniżył jego limit kredytowy w bazie danych… *i zapomniał o tym fakcie poinformować komponent.*



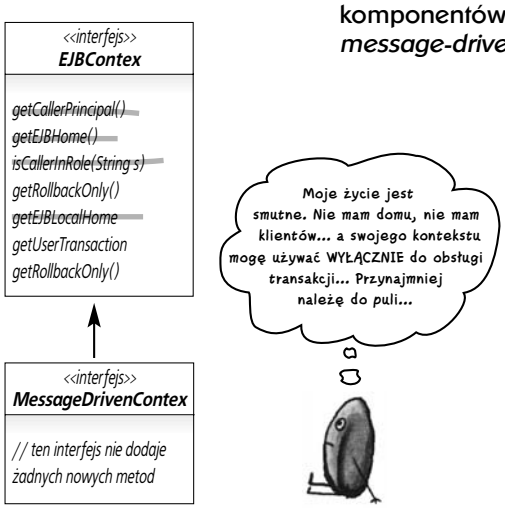
7 Kiedy komponenty są ze sobą powiązane

Komponenty *entity bean* potrzebują relacji. `Zamówienie` potrzebuje `Klienta`. `ElementZamówienia` potrzebuje `Zamówienia`. `Zamówienie` potrzebuje `ElementuZamówienia`. Pomiędzy komponentami *entity bean* mogą występować relacje zarządzane przez kontener i w takim przypadku to kontener dba praktycznie o *wszystko*. Trzeba stworzyć nowy `ElementZamówienia` skojarzony z `Zamówieniem`? Jeśli poprosimy `Klienta` o przedstawienie jego `Zamówień`, uzyskamy nowy `ElementZamówienia`. Jeszcze lepsze jest jednak to, iż stosując język EJB-QL, możemy tworzyć *przenaszalne* zapytania.



8 Odbieranie komunikatu

Odbieranie komunikatów to świetna zabawa. Może nie jest tak miłe jak odbieranie paczki z lampą Olśniewająca Smurfetka™ po wygranej aukcji na portalu eBay, niemniej jednak i tak jest zabawne i *efektywne*. Wyobraź sobie, że po przesłaniu zamówienia nie byłbyś w stanie opuścić mieszkania aż do momentu dostarczenia przesyłki. Tak właśnie się dzieje w przypadku komponentów *session bean* i *entity bean*. Jednak dzięki komponentom *message-driven bean* klient może przesłać komunikat i iść na spacer.



9 Wiek niepodzielności

Transakcje strzegą Twojego bezpieczeństwa. Dzięki transakcjom możesz spróbować wykonać pewną operację, wiedząc, że jeśli coś się nie powiedzie, będziesz mógł udawać, że w ogóle nic się nie stało. Wszystko wróci do stanu, w jakim aplikacja znajdowała się *przed* podjęciem próby wykonania operacji. Transakcje w EJB są czymś wspaniałym — można wdrożyć komponent z transakcjami dostosowanymi do jego potrzeb i to bez wprowadzania *jakichkolwiek* zmian w jego kodzie źródłowym! Oczywiście, jeśli trzeba, to *można* napisać kod obsługujący transakcje.

Zapamiętaj

Komponenty CMT transakcji nie znają a komponenty BMT własnych używają.

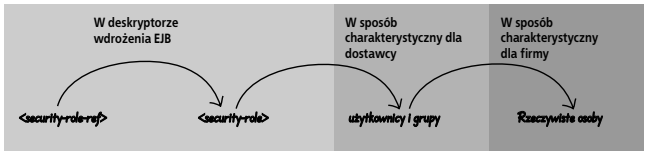
W porządku, sami wiemy, że ten wierszyk nie jest najlepszy. Ale może **samemu** spróbujesz napisać lepszy? Metody wspomagające zapamiętywanie mogą pomóc, ale działają znacznie lepiej, kiedy stosuje się je samemu.

10 Gdy coś się przytrafia komponentom

Oczekuj nieoczekiwanego. Niezależnie od Twych starań, w aplikacji może się zdarzyć coś niedobrego. Coś strasznego, *tragicznego*! Musisz się przed tym zabezpieczyć. Nie możesz dopuścić do tego, by awaria przerwała pracę całej aplikacji i to tylko dlatego, że jeden komponent zgłosił wyjątek. ***Aplikacja musi działać dalej.*** Nie możesz zapobiec tragedii, jednak możesz się na nią *przygotować*. Musisz wiedzieć z jakiej sytuacji da się wyjść, a jaka już na to nie pozwoli, no i *kto* jest odpowiedzialny za zaistniały stan rzeczy.

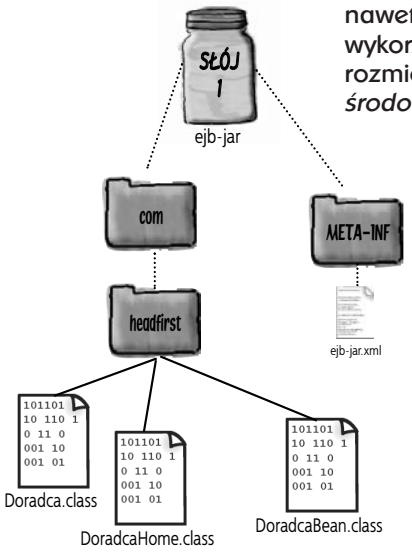
11 Chroń swoje tajemnice

Strzeż swoich tajemnic. Bezpieczeństwo wiąże się z **uwierzytelnianiem** i **autoryzacją**. W pierwszej kolejności musisz potwierdzić swoją tożsamość, a my potem powiemy Ci co możesz zrobić. W technologii EJB łatwo można zapewnić bezpieczeństwo, gdyż programista musi zadbać jedynie o autoryzację. To Ty decydujesz, kto będzie mieć dostęp do określonych metod komponentu. Jest tylko jeden problem… jeśli jesteś `Dostawcą Komponentów` lub `Twórcą Aplikacji`, to prawdopodobnie nie *wiesz* kim będą jej użytkownicy!



12 Radość wdrażania

Ciężko pracowałeś nad stworzeniem tego komponentu. Kodowałeś, kompilowałeś, przetestowałeś. Chyba z miliard razy. *Ostatnią* rzeczą, jaką chciałbyś robić jest modyfikowanie już przetestowanego kodu tylko dlatego, że zmienił się jakiś szczegół w konfiguracji wdrażania aplikacji. A co zrobić, jeśli nawet nie *mamy* kodu źródłowego? Technologia EJB pozwala na wielokrotne wykorzystywanie komponentów dzięki zastosowaniu deskryptorów rozmieszczenia, które można dostosować do własnych potrzeb oraz specjalnego *środowiska* komponentów.



Ostateczny Próbny Egzamin Baru Kawowego. To jest to. 70 pytań. Forma, zagadnienia i poziom trudności jest niemal taki sam jak na *prawdziwym* egzaminie. *My to wiemy*.

Architektura EJB

Technologia EJB jest związana z infrastrukturą. Komponenty są elementami konstrukcyjnymi. Technologia ta daje możliwość tworzenia bardzo dużych aplikacji. Aplikacji, które pozwalają na niemal wszystko, począwszy od obsługi firmy Victoria's Secrets, a skończywszy na zarządzaniu wszystkimi dokumentami w centrum badawczym CERN. Niemniej jednak architektura rozwiązania o takiej elastyczności, mocy i skalowalności nie jest prosta. Wszystko zaczyna się od modelu programowania rozproszonego, w którym klienty, serwery, a nawet poszczególne fragmenty tej samej aplikacji działają „gdzieś" w sieci. Ale w takim razie, jak klient ma *odnaleźć* komponent? Jak może wywołać metody komponentu? Dlaczego istnieją różne typy komponentów? Czy Ridge ponownie ożeni się z Taylor?

Informacje podstawowe

Celem niniejszego rozdziału jest przekazanie Ci podstawowych informacji związanych z technologią EJB, a nie udzielenie wyjaśnień dotyczących konkretnych celów egzaminacyjnych. Niemniej jednak, równie dobrze mógłbyś stwierdzić, że każdy cel i każda odpowiedź udzielana na pytania egzaminacyjne *zależą* od znajomości i zrozumienia *tych* zagadnień podstawowych.

Nie przejmuj się jednak, wystarczająco dużo celów egzaminacyjnych znajdziesz w kolejnym, 3. rozdziale. W rozdziale 6. będziesz już z rozrzewnieniem wspominać ten rozdział i przypominać sobie co czułeś, kiedy nie miałeś na głowie żadnych celów. Będziesz jeszcze tęsknić za tym rozdziałem, więc delektuj się jego lekturą póki jeszcze możesz.

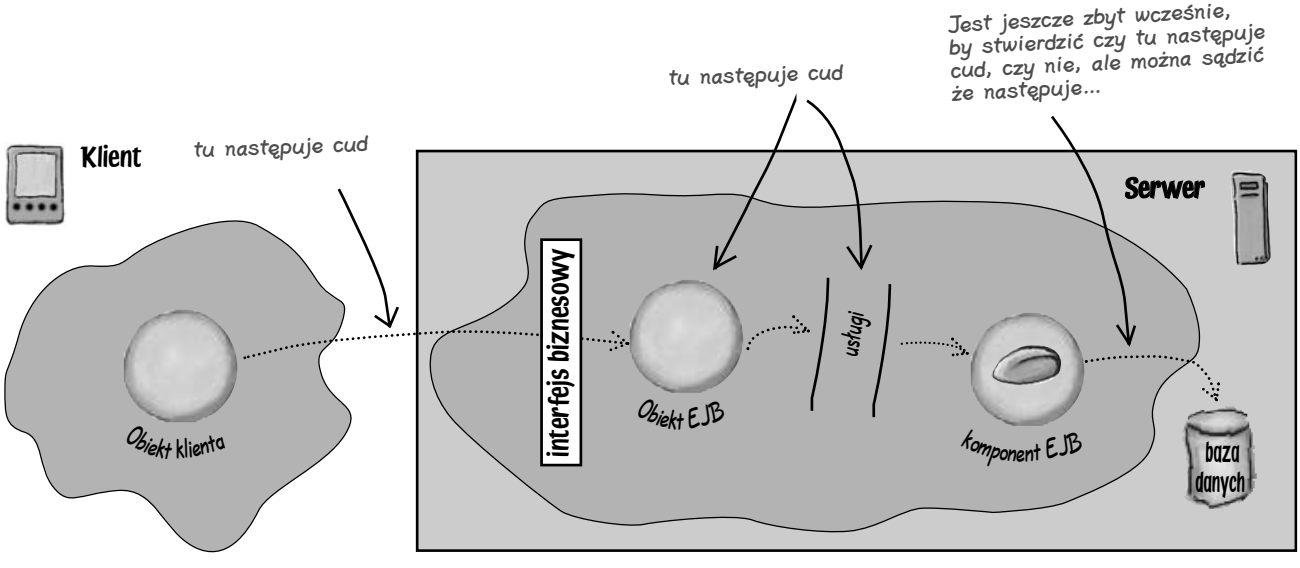
Czy pamiętasz ten rysunek...

Ale to wszystko było zbyt ogólne, aby gdziekolwiek nas doprowadzić. Pomyśl ilu elementów nie ma na tym rysunku. Na przykład, w jaki sposób klient zdobywa referencję do czegoś, co działa na zupełnie innym komputerze? Jak faktycznie przebiega komunikacja klienta z komponentem? Jak to się dzieje, że serwer może ingerować w realizację metody komponentu wywoływanej przez klienta?

Jednym z fundamentów technologii EJB jest inna technologia Javy — *RMI* (ang. *Remote Method Invocation*, wywoływanie zdalnych metod). To fakt, że EJB ukrywa przed programistami komponentów niektóre z bardziej złożonych aspektów RMI, niemniej jednak technologia RMI jest używana i bez jej dokładnego zrozumienia niektóre zagadnienia związane z działaniem EJB nigdy nie ułożą się w logiczną całość.

A zatem, porzucimy ogólne rozważania i zajmiemy się poznawaniem szczegółów i tajników technologii EJB, zaczynając od krótkiej lekcji poświęconej RMI. Jeśli jesteś jednym ze szczęśliwców, którzy mieli już okazję dokładnie poznać technologię RMI, to możesz pominąć tę część rozdziału i przejść bezpośrednio do fragmentu, w którym opisujemy sposoby wykorzystania RMI w EJB. Jednak nawet jeśli dobrze znasz RMI, to powinieneś przynajmniej *pobieżnie przejrzeć* tę część rozdziału, choćby po to, by poznać terminologię i rysunki, których będziemy używać w dalszej części książki.

W porządku, wróćmy na chwilę do punktu wyjścia — czego brakuje na poniższym rysunku? Zacznij od miejsc, w których zdarzają się cuda…



Wygląd architektury EJB przedstawiony ze śmiesznie wysokiego poziomu

Wywoływanie zdalnej metody

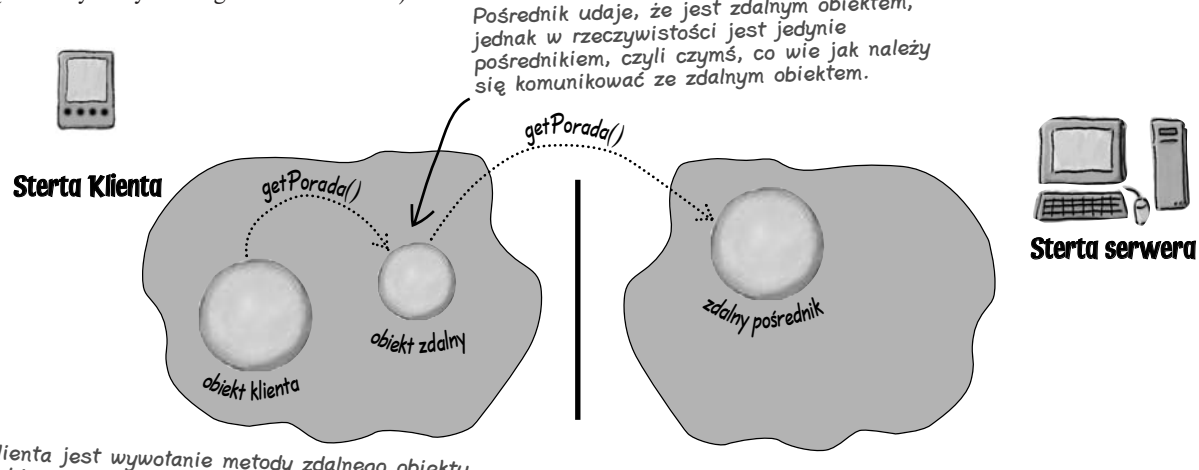
Kiedy piszesz klienta, który będzie korzystać z komponentu, może to być klient *lokalny* bądź *zdalny*. Klient lokalny to taki, który działa na tej samej wirtualnej maszynie Javy (JVM) co komponent. Innymi słowy, zarówno klient, jak i komponent istnieją na tej samej stercie. Zagadnienia te zostaną bardziej szczegółowo opisane w rozdziale 3., pt. *Co widzi klient*, jak na razie powinieneś jednak pamiętać, że *lokalny* oznacza istniejący na *tej samej stercie* (w *tej samej JVM*). Jest wysoce prawdopodobne, że klientów lokalnych będziesz używać wyłącznie wraz z komponentami *entity bean* i tylko w bardzo szczególnych sytuacjach.

Jeśli chcesz, by komponent był używany przez program z „dowolnego miejsca świata", to powinieneś skorzystać z klienta *zdalnego*. Większość aplikacji korporacyjnych posiada klienty zdalne, nawet jeśli niektóre komponenty *używane* w tych aplikacjach komunikują się wzajemnie na zasadach klientów lokalnych. (Nim dotrzesz do końca tej książki dokładnie poznasz wszystkie najdrobniejsze szczegóły tych zagadnień.)

A zatem, w jaki sposób, obiekt umieszczony na jednej stercie (wykonywany przez jedną JVM) może wywołać jakąś metodę, używając w tym celu referencji do obiektu działającego na innej stercie (wykonywanego przez inną JVM)? Z technicznego punktu widzenia wywołanie takie nie jest możliwe! Referencje w Javie to ciągi bitów, które nic nie znaczą poza konkretną wirtualną maszyną Javy, na której zostały stworzone i są używane. Gdybyś był obiektem i posiadał referencję do innego obiektu, *to ten drugi obiekt musiałby się znajdować na tej samej stercie co Ty*.

Technologia RMI rozwiązuje ten problem poprzez udostępnienie klientowi obiektu pośrednika (ang. *stub*), który obsługuje wymianę informacji pomiędzy klientem i zdalnym obiektem. Klient wywołuje metody *pośrednika*, a *pośrednik* obsługuje wszystkie czynności niskiego poziomu związane z komunikacją ze zdalnym obiektem (i bazujące na wykorzystaniu gniazd i strumieni).

Dzięki RMI obiekt klienta zaczyna działać w taki sposób, jak gdyby wywoływał zdalną metodę. Jednak w rzeczywistości wywołuje on jedynie metody obiektu „pośrednika", działającego na tej samej stercie. Pośrednik obsługuje wszystkie operacje sieciowe niskiego poziomu związane z wykorzystaniem gniazd i strumieni.



Celem klienta jest wywołanie metody zdalnego obiektu (czyli zrobienie czegoś, co w rzeczywistości jest niemożliwe). Jednak ze względu na to, że zdalny obiekt znajduje się na innej stercie, klient wywołuje metodę obiektu pośrednika (który z kolei przesyła odpowiednie informacje siecią).

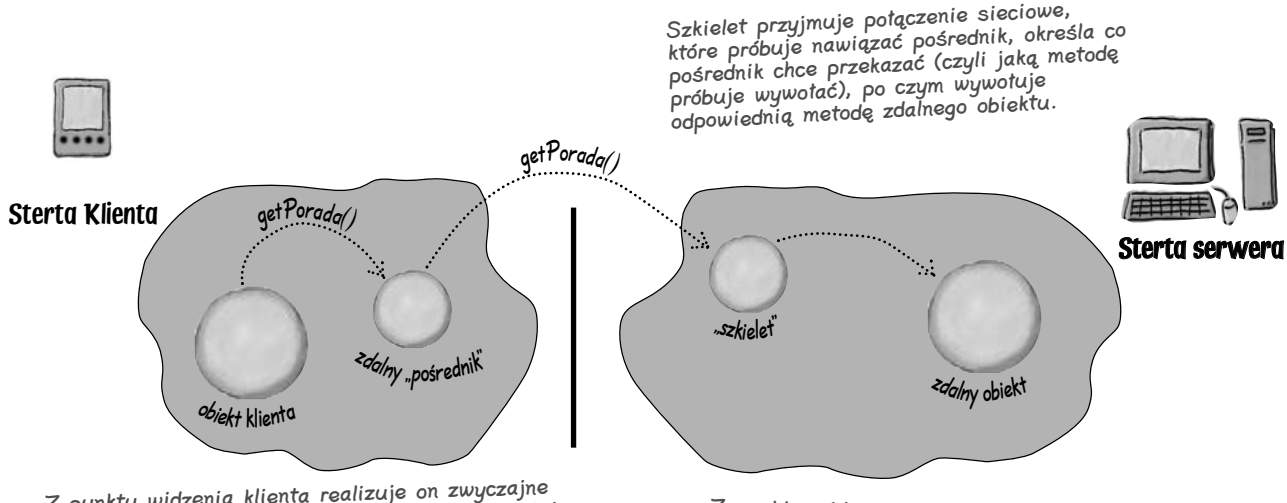
Obiekt zdalny, to obiekt posiadający faktyczną metodę wykonującą operacje, które klient chce wykonać (na przykład: `sprawdzKarteKredytowa()`, `obliczWartoscPI()` i tak dalej.)

Także po stronie serwera jest dodatkowy „pomocnik"...

Zdalny obiekt dysponuje metodą, którą klient chce wywołać. Jednak, kiedy pośrednik nawiązuje połączenie sieciowe z serwerem, *coś* na serwerze musi pobrać informacje ze strumienia wejściowego i przekształcić na wywołanie metody zdalnego obiektu. Oczywiście *można by* umieścić kod służący do obsługi połączeń sieciowych w samym zdalnym obiekcie, jednak takie rozwiązanie zaprzeczałoby podstawowej idei RMI — której zadaniem jest takie ułatwienie wywołania metody zdalnego obiektu przez klienta, by było ono równie proste jak wywołanie metody obiektu istniejącego na tej samej stercie co klient. Celem technologii RMI jest zapewnienie *przezroczystości operacji sieciowych*. Innymi słowy, fakt, że obiekt klienta oraz obiekt zdalny działają na różnych komputerach, powinien być prawie niezauważalny dla programisty. Z Twojego punktu widzenia oznacza to możliwość tworzenia krótszego i łatwiejszego kodu.

A zatem, przy tak zdefiniowanych celach, technologia RMI bierze na siebie odpowiedzialność także za obsługę wywołania zdalnej metody po stronie serwera. To *coś*, co po stronie serwera odbiera połączenie sieciowe, nosi nazwę *szkieletu* (ang. *skeleton*). Szkielet stanowi przeciwieństwo pośrednika. W początkowych wersjach technologii RMI dla każdego pośrednika trzeba było stworzyć odpowiadający mu obiekt szkieletu. Jednak obecnie nie zawsze jest to konieczne. Oczywiście *coś* po stronie serwera musi zapewniać możliwości *funkcjonalne*, jakimi dysponuje szkielet, jednak tworzenie samego *obiektu* szkieletu jest opcjonalne. Nie będziemy szczegółowo wyjaśniać tego zagadnienia, gdyż w kontekście technologii EJB nie ma ono większego znaczenia. Sposób, w jaki kontener implementuje możliwości funkcjonalne szkieletu, zależy wyłącznie od jego twórców.

Nas obchodzi tylko to, że na serwerze jest *coś*, z czym pośrednik jest w stanie się komunikować, oraz że to *coś* potrafi zinterpretować przesyłane przez niego komunikaty i wywoływać odpowiednie metody zdalnego obiektu.



Nie ma niemądrych pytań

P: W jaki sposób możliwe jest uzyskanie „przezroczystości operacji sieciowych"? Co się dzieje, gdy serwer zostanie wyłączony lub sieć ulegnie uszkodzeniu w momencie gdy klient wywoła zdalną metodę? Wydaje się, że w tym przypadku istnieje znacznie więcej powodów, które mogą doprowadzić do wystąpienia problemów, niż gdyby obiekt klienta realizował zwyczajne wywołanie metody innego obiektu umieszczonego na tej samej stercie.

O: Tak, tak. Bez wątpienia rozumiesz, że „przezroczystość operacji sieciowych" nie jest jedynie mitem — jest złym pomysłem. Oczywiście, że w przypadku wywoływania zdalnej metody może wystąpić WIELE problemów, które nie pojawiają się podczas realizacji zwyczajnych wywołań, a klient musi być na te sytuacje przygotowany.

To właśnie z tego powodu w technologii RMI *wszystkie* zdalne metody muszą deklarować wyjątek `java.rmi.RemoteException`, który jest wyjątkiem weryfikowanym. A to oznacza, że klient musi bądź go obsługiwać, bądź zadeklarować. Innymi słowy, klient tak *naprawdę* nie może udawać, że wywołanie metody jest standardowe, a nie „zdalne".

Ale chwileczkę, to nie wszystko — klient musi wykonać specjalne operacje, żeby w ogóle *zdobyć* referencję do zdalnego obiektu. A właściwie czym *jest* ta referencja? Tak naprawdę, jest to zwyczajna referencja do pośrednika zdalnego obiektu.

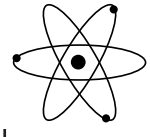
A zatem, należałoby odpowiedzieć, że RMI nie zapewnia prawdziwej przezroczystości operacji sieciowych. Projektanci tej technologii chcą, by klient potwierdził świadomość, że *podczas wywoływania zdalnej metody mogą się zdarzyć katastrofalne problemy*.

Niemniej jednak, przyglądając się wszystkim operacjom, które należy wykonać w celu wywołania zdalnej metody (nawiązaniu połączenia sieciowego, obsłudze strumieni, spakowaniu i przekazaniu argumentów itd.), możemy dojść do wniosku, że klient i tak musi wykonać jedynie dwie proste czynności: użyć specjalnej usługi wyszukiwawczej w celu uzyskania referencji do zdalnego obiektu i umieścić wywołanie zdalnej metody w bloku `try-catch`. To naprawdę banalne operacje w porównaniu z tym wszystkim, co klient musiałby robić, gdyby samodzielnie chciał obsługiwać cały proces.

(A całe zadanie możemy jeszcze bardziej uprościć, stosując wzorzec programowy wykorzystywany w technologii EJB, który poznamy w ostatnim rozdziale.)

P: Czy to ja jestem odpowiedzialny za stworzenie obiektu pośrednika i szkieletu? Skąd pośrednik wie jakie metody posiada mój zdalny obiekt? Na podobnej zasadzie — skąd klient wie jakie metody udostępnia mój zdalny obiekt?

O: Nie, nie musisz tworzyć ani obiektu pośrednika, ani obiektu szkieletu. W przypadku technologii RMI są one generowane za pomocą kompilatora RMI (rmic). Jeśli chodzi o pozostałe dwa pytania... to nim zaczniemy wyjaśniać, pozwolimy Ci jeszcze pomyśleć chwilkę nad odpowiedziami na nie.



Wysil szare komórki

Jaki jest najlepszy sposób, który można zastosować w Javie, by przekazać klientowi informacje o dostępnych metodach? Innymi słowy, w jaki sposób można pokazać swoje publiczne metody całemu światu?

Pomyśl o relacji pomiędzy obiektem pośrednika a faktycznym zdalnym obiektem. Jakie są wspólne cechy, które oba te obiekty muszą posiadać?

(Odpowiedzi na te pytania znajdziesz kilka stron dalej.)

A co z argumentami i wartościami wynikowymi?

Wywołania zdalnych metod przypominają wywołania metod lokalnych, z tym że mogą zgłaszać wyjątki `RemoteException`. A jaki byłby pożytek z wywołania metody, gdyby nie można było przekazać do niej żadnych argumentów ani pobrać wartości wynikowej? Równie dobrze mógłbyś używać RPC*, jak Twoi rodzice.

I tak doszliśmy do jednego z kluczowych zadań, jakie muszą realizować obiekty pośrednika i szkieletu (czy też cokolwiek, co pełni funkcję szkieletu) — pakowania i rozpakowywania argumentów wywołania przesyłanych siecią.

Pamiętaj, że w rzeczywistości klient wywołuje metodę pośrednika, do którego ma lokalny dostęp (co oznacza, że zarówno klient, jak i pośrednik znajdują się na tej samej stercie). Dlatego też, z punktu widzenia klienta przesyłanie argumentów wywołania metody nie ma w sobie nic szczególnego. To *pośrednik* wykonuje całą „czarną" robotę. Musi on spakować argumenty wywołania (do czego jest wykorzystywany proces określany jako szeregowanie, ang. *marshaling*), zapisać je w strumieniu wyjściowym, a następnie, za pośrednictwem połączenia sieciowego, przesłać na serwer.

Z kolei do zadań szkieletu należy odebranie i przetworzenie danych przesłanych przez pośrednika, rozpakowanie argumentów, określenie co należy z nimi zrobić (na przykład, jaką metodę jakiego obiektu należy wywołać) i wywołanie metody (w tym przypadku jest to wywołanie *lokalne*) zdalnego obiektu wraz z przekazaniem do niej odpowiednich argumentów.

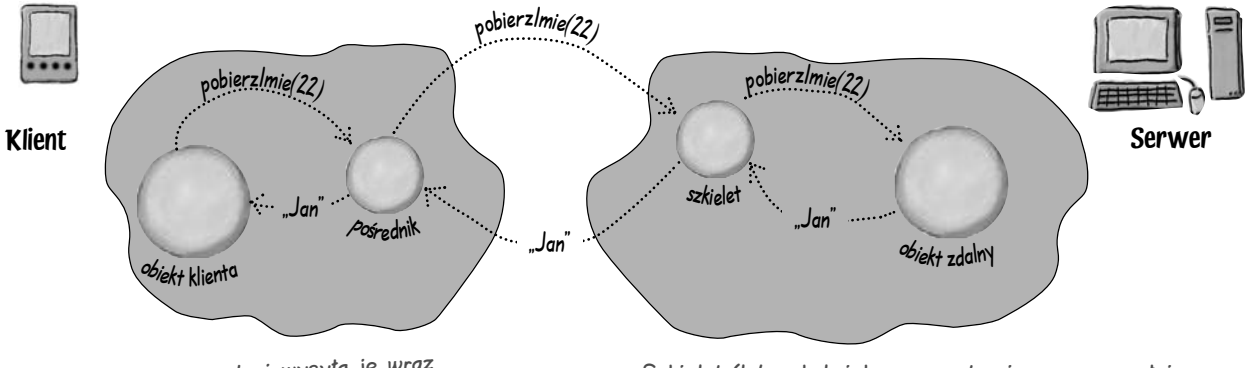
Następnie cały proces jest wykonywany raz jeszcze, tylko w przeciwnym kierunku! Szkielet pakuje wartości wynikowe i przesyła je do pośrednika, który z kolei je rozpakowuje i zwraca klientowi jako zwyczajne wartości. Jednak, aby można było przesłać argumenty i wartości wynikowe, muszą to być wartości typów podstawowych, obiekty implementujące interfejs `Serializable`, tablice lub kolekcje wartości typów podstawowych lub obiektów implementujących interfejs `Serializable` bądź też obiekty implementujące interfejs `Remote`.

Zarówno pośrednik, jak i szkielet uczestniczą w całym procesie wywoływania zdalnej metody. Oba są odpowiedzialne za pakowanie oraz rozpakowywanie wartości przesyłanych siecią.

Cały ten proces nie mógłby działać, gdyby argumenty oraz wartości wynikowe nie nadawały się do przesłania siecią.

Wartości, które można przysyłać to:

- **wartości typów podstawowych,**
- **obiekty implementujące interfejs Serializable,**
- **tablice lub kolekcje wartości typów podstawowych bądź obiektów implementujących interfejs Serializable,**
- **obiekty implementujące interfejs Remote.**



Pośrednik pakuje argumenty i wysyła je wraz z informacjami o wywołaniu metody, a następnie odbiera wyniki, rozpakowuje je i przekazuje klientowi.

Szkielet (lub cokolwiek, co po stronie serwera pełni funkcję szkieletu) rozpakowuje argumenty wywoływanej metody obiektu zdalnego, po czym pakuje i wysyła wyniki wywołania.

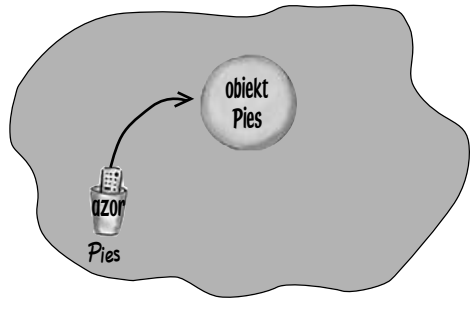
* *RPC* to skrót od terminu *Remote Procedure Call* (wywoływanie zdalnych procedur). Ale to nudy...

Zaraz, chwileczkę... Przecież Java przekazuje obiekty, przekazując kopię *referencji* do danego obiektu, a nie sam obiekt. A zatem, jakim cudem coś takiego może działać w przypadku wywołań zdalnych? Przecież taka referencja nie ma najmniejszego znaczenia na innej stercie...

W przypadku zwyczajnych, lokalnych wywołań metod Java przekazuje referencje przez wartość. Innymi słowy, kopiuje bity zapisane w zmiennej referencyjnej.

Sam obiekt nigdy nie jest przekazywany.

Będziemy sobie wyobrażać, że referencja jest pilotem do obiektu znajdującego się na stercie. Pilotem, czyli czymś, czego możemy użyć do naciskania przycisków i sterowania obiektem (czyli, na przykład, do wywoływania jego metod).

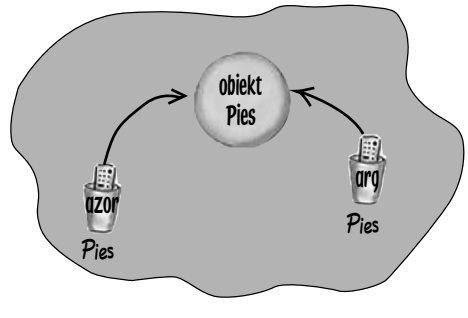



```
void doDziela() {
   Pies azor = new Pies();
   this.szkolZwierzaka(azor);
}
```

Co tak naprawdę tu przekazujemy?

Kopia referencji (pilota), a nie obiekt `Pies`.

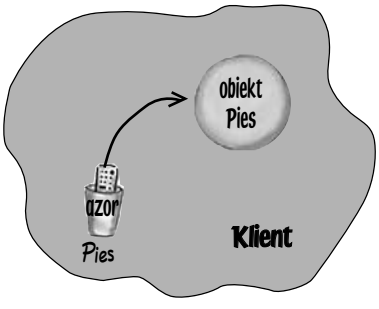
```
void szkolZwierzaka(Pies arg) {   }
```



Teraz parametr „arg" oraz zmienna „azor" są identyczne. Obie odwołują się do tego samego obiektu Pies.

Oczywiście wiemy, że Ty to wszystko wiesz, ale wyjaśniamy to tak na wszelki wypadek, aby mieć pewność, że jesteś „na tej samej stronie co my" (co brzmi nieco dziwnie, bo to przecież oczywiste, że jesteśmy na tej samej stronie), i że rozumiesz ideę.

Co tak naprawdę jest przekazywane, gdy przekazujesz obiekt w wywołaniu zdalnej metody?



Wyobraź sobie poniższy fragment kodu używanego przez KLIENTA:

```
try {
   Pies azor = new Pies();

   zdalnyPosrednik.szkolZwierzaka(azor);
} catch (RemoteException ex) {
     ex.printStackTrace();
}
```

Co tak naprawdę tutaj przekazujemy??

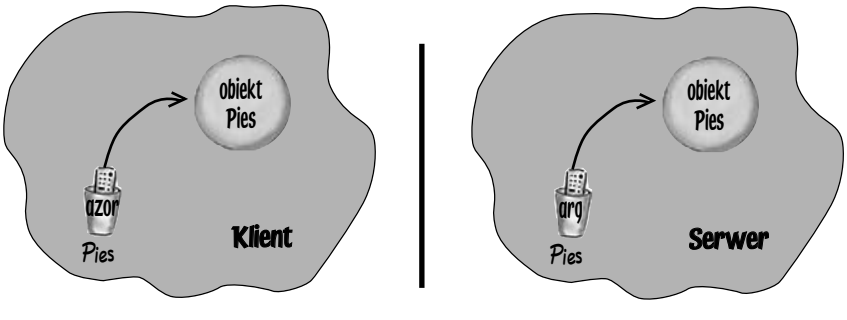
To NIE jest referencja!

To serializowana kopia faktycznego obiektu Pies.

Oraz ten kod ZDALNEJ metody:

```
void szkolZwierzaka(Pies pies) { }
```

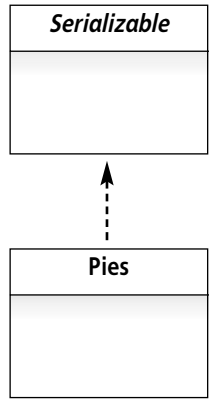
Obiekt `Pies` *na serwerze jest kopią obiektu istniejącego po stronie klienta.*



Jeśli Twoja zdalna metoda wymaga przekazania argumentu, którego wartość jest obiektem, to argument ten będzie przekazywany jako kompletna kopia obiektu!

W przypadku wywołań zdalnych metod, Java przekazuje obiekty kopiując je, a nie w formie referencji.

Serializowana kopia obiektu jest przekazywana do obiektu zdalnego.

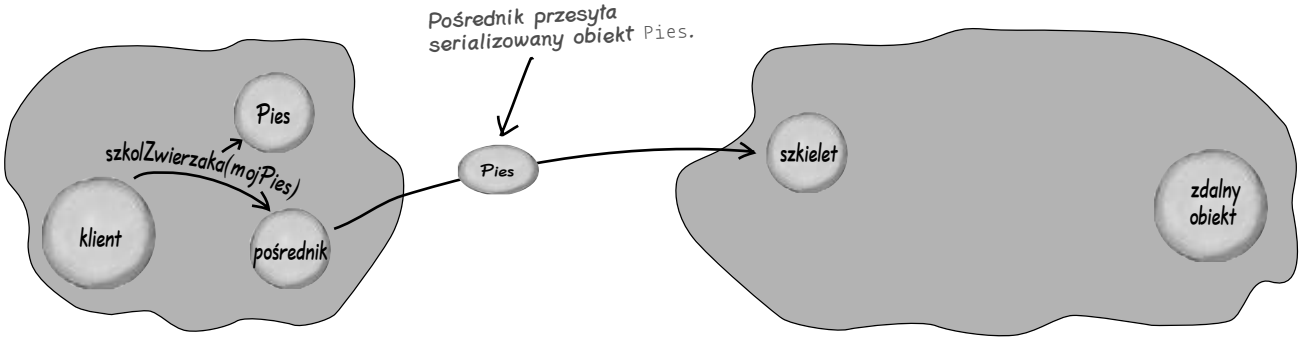


Klasa `Pies` implementuje interfejs `Serializable`

Przekazywanie argumentu będącego obiektem z klienta na serwer

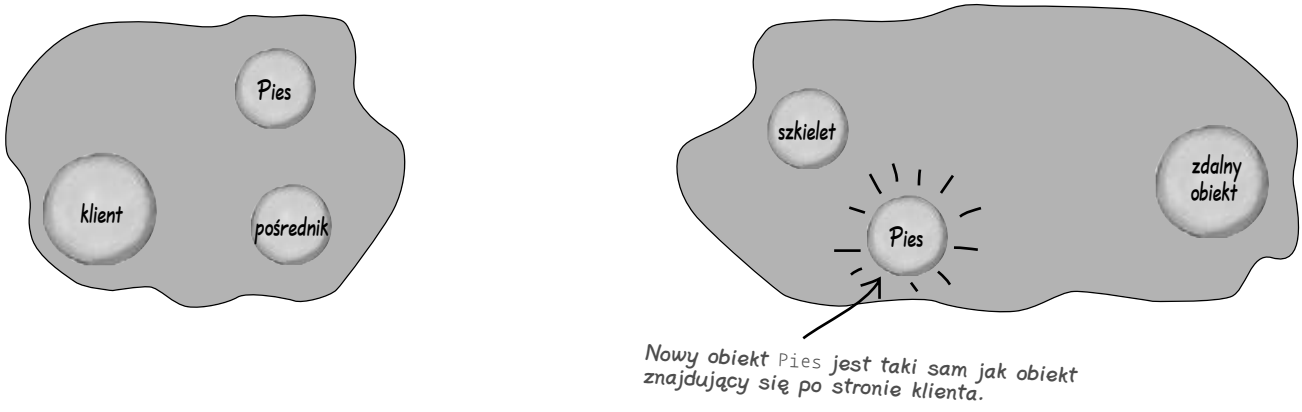
❶ **Klient wywołuje metodę** `szkolZwierzaka(mojPies)` **obiektu pośrednika, przekazując w nim referencję do obiektu** `Pies`.

❷ **Pośrednik tworzy serializowaną kopię obiektu i przesyła ją siecią do obiektu szkieletu.**

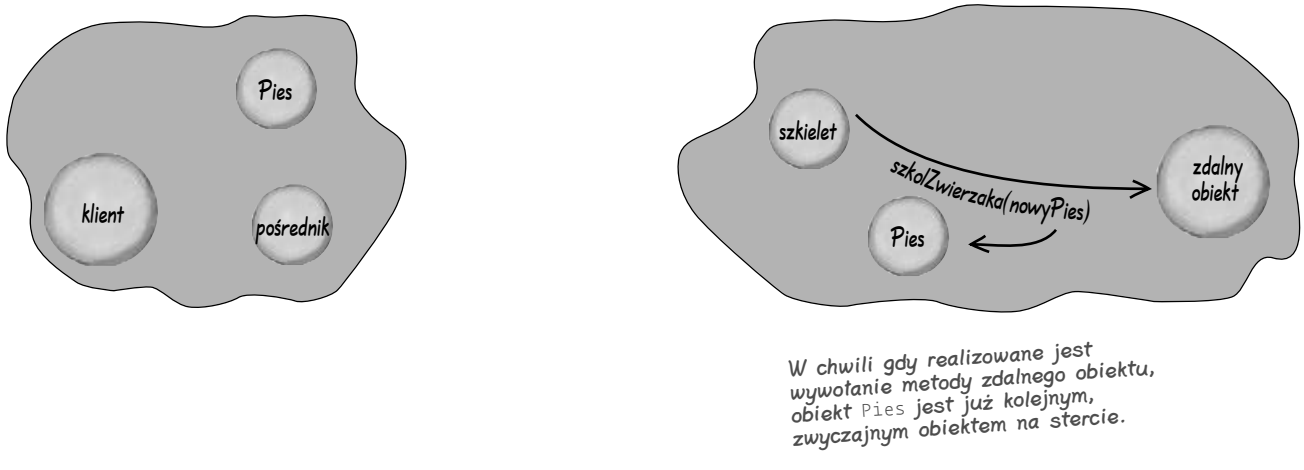


Rozpakowywanie (deserializacja) obiektu na serwerze

3 **Szkielet przeprowadza deserializację przekazanych argumentów, tworząc tym samym nowy obiekt** `Pies` **na stercie zdalnego obiektu.**



4 **Szkielet wywołuje metodę zdalnego obiektu, przekazując w wywołaniu zwyczajną referencję do nowego obiektu** `Pies`**.**



Nie ma niemądrych pytań

P. Mam obiekt `HashMap` pełny obiektów `Klient` (implementujących interfejs `Serializable`), z których każdy ma unikatowy klucz typu `String`. Czy powinienem martwić się tym, czy sam obiekt `HashMap` można serializować?

O. W porządku, to jedno z serii podchwytliwych pytań. Wszystkie implementacje interfejsu `Collection` dostępne w J2SE API implementują interfejs `Serializable`. A zatem, nie musisz przejmować się obiektem `HashMap` — jeśli obiekty, które w nim umieszczasz implementują interfejs `Serializable`, to wszystko będzie w porządku.

Ale... jest *coś*, co może doprowadzić do wystąpienia problemów. Prawdopodobnie nigdy się nie znajdziesz w takiej sytuacji, niemniej jednak warto o niej wspomnieć. Zapewne już wiesz, że klasy implementujące interfejs `Map`, takie jak `HashMap` oraz `Hashtable`, dysponują metodą `values()` zwracająca kolekcję wartości, bez skojarzonych z nimi kluczy. Innymi słowy, możesz wywołać tę metodę swojego obiektu `HashMap` i uzyskać w rezultacie kolekcję obiektów `Klient`.

Jednak kolekcję *jakiego* typu? I na tym właśnie polega problem. Nie wiadomo jakiego typu. Wiadomo tylko to, że jest to obiekt implementujący interfejs `Collection`. Jednak informacja ta nie wystarcza nam, by stwierdzić czy obiekt zwrócony przez wywołanie metody `values()` może być serializowany! Innymi słowy, może się okazać, że pomimo umieszczenia w kolekcji obiektów implementujących interfejs `Serializable`, nie można przeprowadzić serializacji samej kolekcji!

A oto wniosek: nie polegaj na tym, że kolekcje zwrócone przez metodę `values()` obiektów implementujących interfejs `Map`, będzie można serializować. Umieszczaj swoje obiekty w czymś, co na pewno pozwala na serializację, czyli, na przykład, w obiektach `ArrayList`, a dopiero potem próbuj przesyłać je jako argument wywołania zdalnej metody.

P. Jeśli czegoś, co przekazujesz w wywołaniu zdalnej metody nie można serializować, to czy próba wykonania serializacji spowoduje wystąpienie błędu podczas kompilacji, czy też w trakcie działania programu?

O. W czasie działania programu! A przynajmniej zazwyczaj tak bywa. Pamiętaj, że *zadeklarowany* typ argumentu lub wartości wynikowej niekoniecznie odpowiada typowi obiektu, który będzie przekazywany lub zwracany w czasie wykonywania programu.

Problemy mogłyby zostać wykryte podczas *kompilacji* tylko i wyłącznie w przypadku, gdyby *zadeklarowanym* typem argumentów lub wartości wynikowych był interfejs **`Serializable`**:

```
public void wezTo(Serializable s);
```

W takim przypadku kompilator mógłby wykorzystać standardowe mechanizmy kontroli typów, by sprawdzić, czy typ przekazywanego argumentu faktycznie implementuje `Serializable`.

Jednak w większości przypadków `Serializable` nie jest zadeklarowanym typem argumentów ani wartości wynikowych (są nimi raczej takie typy jak `Pies`, `ArrayList`, `String` itp.); dlatego też Java nie będzie wiedzieć czy przekazywany obiekt można serializować, czy nie, aż do chwili gdy spróbuje to zrobić (a w takiej sytuacji, gdy obiekt nie implementuje interfejsu `Serializable`, zostanie zgłoszony wyjątek).

W przypadku serializacji tablic i kolekcji, jeśli którykolwiek z umieszczonych w nich obiektów nie będzie implementować interfejsu `Serializable`, to cała operacja serializacji nie powiedzie się.

P. Czy moje klasy muszą jawnie implementować interfejs `Serializable`, czy wystarczy, że będzie on dziedziczony po klasie bazowej?

O. Przypomnij sobie jedną z podstawowych zasad obowiązujących w Javie: jeśli Twój *rodzic* (czyli klasa bazowa) jest czymś, to także Ty jesteś tym czymś. Jeśli klasa Pies dziedziczy po klasie `Zwierze`, a klasa `Zwierze` implementuje interfejs `Serializable`, to także obiekty `Pies` nadają się do serializacji i to niezależnie do tego czy klasa `Pies` jawnie implementuje interfejs `Serializable`, czy nie.

Niemniej jednak, jawne deklarowanie, że klasa implementuje interfejs `Serializable`, nawet jeśli jest on implementowany w klasie bazowej, jest uznawane za dobrą praktykę programistyczną. Dzięki temu inne osoby używające stworzonych przez Ciebie klas nie będą musiały wnikliwie analizować całej ich hierarchii, by sprawdzić czy któraś z klas bazowych implementuje interfejs `Serializable`.

Pamiętaj, że argumenty wywołania oraz wartości wynikowe zwracane przez zdalne metody muszą być:

- **wartościami typów podstawowych,**
- **obiektami implementującymi interfejs** `Serializable`**,**
- **tablicami lub kolekcjami wartości typów podstawowych lub obiektów implementujących interfejs** `Serializable`**,**
- **obiektami implementującymi interfejs** `Remote`**.**

Za chwilę wszystko się wyjaśni i ułoży w logiczną całość. Jednak zanim odwrócisz kartkę, zastanów się przez chwilę nad implikacjami wynikającymi z faktu przesyłania w wywołaniu zdalnej metody zdalnego obiektu.

Przekazywanie obiektu Remote w wywołaniu zdalnej metody

Przesyłanie zdalnego obiektu w wywołaniu zdalnej metody nie ma sensu. W końcu taki obiekt istnieje po to, aby być zdalnym. Aby mogły z niego korzystać klienty działające… gdzie indziej. Na innej stercie.

Co się jednak stanie, jeśli zechcemy przekazać do zdalnego klienta referencję do innego zdalnego obiektu? Co się stanie, jeśli zamiast normalnej kopii obiektu `Klient`, przekażemy do niego pośrednika do zdalnego obiektu `Klient`?

Przemyśl to sobie *zanim* rozpoczniesz lekturę następnej strony.

Wysil szare komórki

Jakie są skutki przekazania pośrednika do zdalnego obiektu `Klient` zamiast zwyczajnego obiektu `Klient`?

Jakie korzyści wynikają z przekazania pośrednika zamiast zwyczajnego obiektu `Klient`?

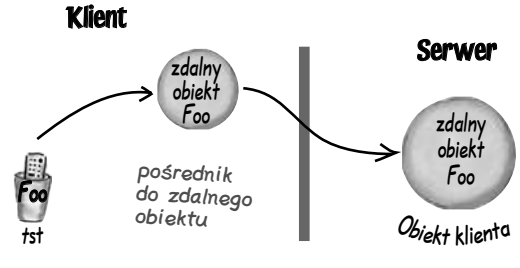
Jakie są wady takiego rozwiązania?

(Notatka: wybór pomiędzy przekazywaniem serializowanych obiektów a przekazywaniem pośredników do obiektów zdalnych jest kluczową decyzją projektową. Przeanalizujemy ją, gdy zajmiemy się zagadnieniami związanymi z wydajnością i wzorcami.)

W przypadku przekazywania obiektu typu `Remote` do lub z wywołania zdalnej metody, Java przekazuje w rzeczywistości pośrednika do tego obiektu.

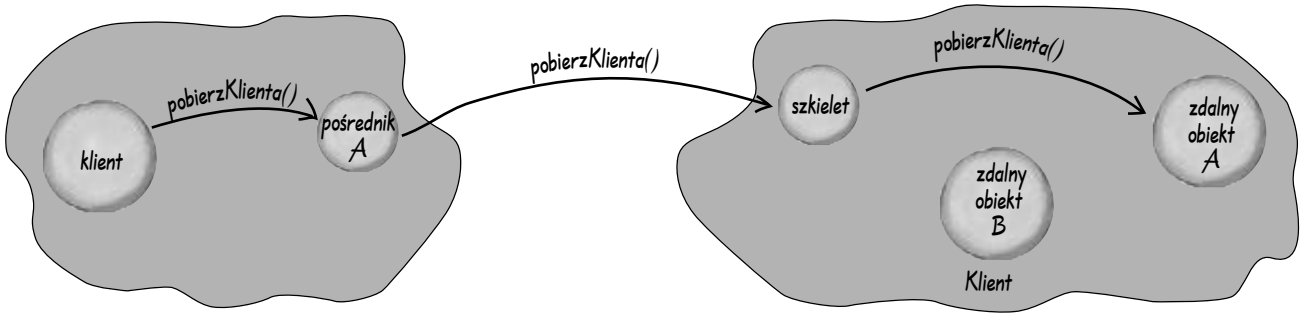
Innymi słowy, podczas działania programu zdalny obiekt pozostaje dokładnie tam, gdzie jest, a zamiast niego przekazywany jest jedynie jego pośrednik.

Zdalna referencja to pośrednik do zdalnego obiektu. Jeśli klient dysponuje zdalną referencją, oznacza to, że dysponuje zwyczajną, lokalną referencją do pośrednika, który z kolei jest w stanie komunikować się ze zdalnym obiektem.



Kiedy wartość wynikowa jest zdalnym obiektem...

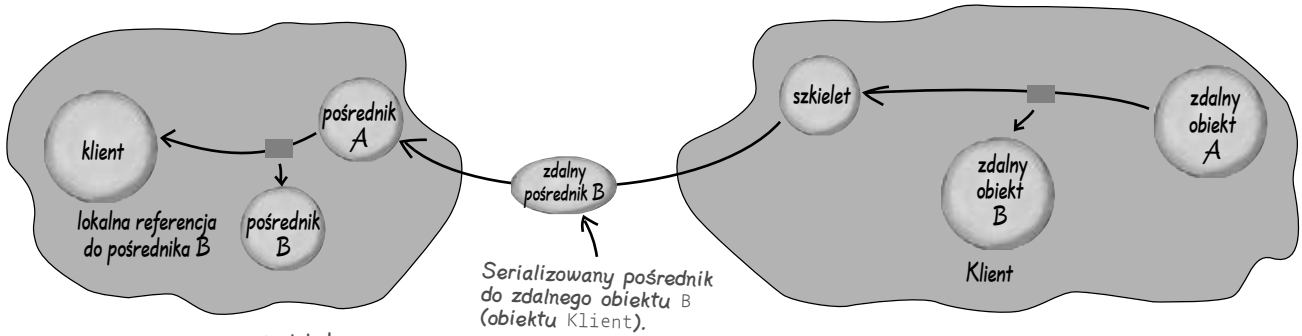
1. **Klient wywołuje metodę `pobierzKlienta()` zdalnego obiektu A (korzystając przy tym z pośrednika A).**



2. **Zdalny obiekt A zwraca referencję do obiektu `Klient` (zdalny obiekt B). Szkielet zastępuje (i serializuje) pośrednika do tego obiektu i przesyła go z powrotem do klienta.**

   Pośrednik klienta (B) zostaje odtworzony (deserializowany) po stronie klienta, po czym klient otrzymuje lokalną referencję do nowego obiektu pośrednika.

Obiekt zdalny (A) zwraca lokalną referencję do obiektu `Klient` (zdalnego obiektu B), jednak z powrotem jest tak naprawdę przesyłany pośrednik.



Serializowany pośrednik do zdalnego obiektu B (obiektu `Klient`).

Teraz klient otrzymuje lokalną referencję do pośrednika do obiektu `Klient` (B).

Nie ma niemądrych pytań

P. Co się stanie, jeśli obiekt klienta oraz obiekt zdalny będą działać na różnych JVM, lecz na tym samym komputerze? Innymi słowy, jeśli będą działać w dwóch różnych programach napisanych w Javie i wykonywanych na tym samym serwerze?

O. To nie ma żadnego znaczenia. Liczy się tylko to, czy dwa obiekty istnieją na tej samej czy na różnych stertach, a na szczęście dwie wirtualne maszyny Javy nie używają tej samej sterty, niezależnie do tego, jak są sobie bliskie (czyli, bez względu na to czy działają na tym samym serwerze).

W praktyce jednak możesz (a w przypadku technologii EJB często *musisz*) używać RMI, nawet jeśli obiekty znajdują się na tej samej stercie.

P. Niby dlaczego miałbyś chcieć używać RMI, jeśli nie jest to konieczne? Czy samo wywoływanie zdalnych metod nie jest wystarczająco kosztowne?

O. Zagadnieniem tym zajmiemy się szczegółowo nieco później, jednak głównym powodem jest to, iż rezygnacja z wykorzystania RMI do wywoływania metod powoduje ograniczenie aplikacji poprzez utratę możliwości rozmieszczania obiektów w różnych miejscach sieci (lub nawet na tym samym serwerze). Innymi słowy, jeśli nie wykorzystasz RMI, to oba obiekty będą musiały znajdować się na tej samej stercie.

Z punktu widzenia modelu programowania rozproszonego jest to decyzja nieodwołalna, gdyż późniejsza zmiana rozwiązania pociągnęłaby za sobą konieczność przepisywania całego kodu. Z drugiej strony, jeśli *zastosujesz* RMI, to w dowolnej chwili będziesz mógł zdecydować się, aby go podzielić na części i rozmieścić w różnych miejscach swojego systemu, przy czym będzie to wymagało niewielkich, a niejednokrotnie nie będzie wymagało *żadnych*, zmian w kodzie.

A zatem, kosztem uzyskania elastyczności traci się wydajność działania, jednak w przypadku znacznej większości rozproszonych aplikacji korporacyjnych, nie jest to wcale największy problem. Zazwyczaj największy wpływ na wydajność działania aplikacji ma przepustowość sieci oraz możliwości współbieżnej realizacji zadań. Ogólnie rzecz biorąc, to prawdopodobnie inne czynniki będą w większym stopniu oddziaływać na wydajność aplikacji niż to, czy używasz wywołań zdalnych czy lokalnych. Jednak życie nie zawsze jest takie proste, dlatego też powrócimy do tego zagadnienia w dalszej części książki.

KLUCZOWE ZAGADNIENIA

- Technologia EJB używa RMI, aby zdalne klienty mogły korzystać z komponentów.
- W tym kontekście klient zdalny to obiekt wykonywany przez inną wirtualną maszynę Javy, co jednocześnie oznacza, że jest on umieszczony na innej stercie.
- Zdalny obiekt pozostaje na swojej stercie, natomiast klienty wywołują metody *pośrednika* tego zdalnego obiektu.
- Obiekt pośrednika obsługuje wszystkie sieciowe operacje niskiego poziomu związane z komunikacją ze zdalnym obiektem.
- Kiedy klient chce wywołać metodę zdalnego obiektu, wywołuje tę samą metodę pośrednika. Pośrednik istnieje na tej samej stercie co klient.
- Z punktu widzenia klienta, wywołanie zdalnej metody jest niemal identyczne z wywołaniem metody lokalnej; jedyna różnica polega na tym, że wywołanie metody zdalnej może zgłosić wyjątek **`RemoteException`** (wyjątek weryfikowany).
- Pośrednik przygotowuje informacje o argumentach wywołania metody i przesyła je do obiektu *szkieletu* działającego na serwerze. Sam obiekt szkieletu jest wprawdzie opcjonalny, jednak jego zadania muszą zostać wykonane. Nie musimy przejmować się tym kto — lub co — wykona te zadania.
- Argumenty wywołania zdalnej metody oraz zwracane przez nie wartości muszą być wartościami typów podstawowych, obiektami implementującymi interfejs `Serializable`, tablicami lub kolekcjami wartości typów podstawowych lub obiektów implementujących interfejs `Serializable` bądź też obiektami implementującymi interfejs `Remote`. Jeśli argumenty lub wartości wynikowe będą jakiegokolwiek innego typu, to podczas działania programu zostanie zgłoszony wyjątek.
- Jeśli wartością argumentu lub wynikiem zwracanym przez metodę będzie obiekt, to zostanie on przesłany jako serializowana *kopia*, a następnie odtworzony na lokalnej stercie zdalnego obiektu.
- Jeśli wartością argumentu lub wynikiem metody będzie obiekt typu `Remote`, to zamiast samego obiektu zostanie przesłany jego pośrednik.


O rany! Niemal zapomniałam o najważniejszej sprawie związanej z wywołaniami zdalnych metod! Jeśli to serializowane obiekty są przekazywane jako argumenty lub wartości wynikowe, to koniecznie musisz upewnić się, że plik klasowy dla klasy przekazywanego obiektu jest dostępny na drugim komputerze. Jeśli plik klasowy nie będzie dostępny, to nigdy nie uda się odtworzyć przekazanego obiektu.
To dotyczy także klas pośredników. Jeśli klient nie dysponuje plikami klasowymi obiektu pośrednika, to wszystko na nic!

Jakie wspólne cechy muszą mieć zdalny obiekt i pośrednik?

Skąd klient ma wiedzieć, jakie metody może wywoływać?

Skąd pośrednik wie, jakimi metodami dysponuje zdalny obiekt?

Pamiętaj — skoro pośrednik udaje, że jest zdalnym obiektem, musi mieć te same metody co on.

Oczywiście znasz odpowiedź na te pytania.

Oczywiście — *interfejs*.

Właśnie w taki sposób klient powinien dowiadywać się o wszystkich metodach dostępnych w środowisku rozproszonym.

Taki interfejs nazywamy ***biznesowym***, gdyż zawiera on metody biznesowe, z których chce korzystać klient. Jeśli chodzi o szczegóły techniczne, to interfejs biznesowy zdalnego obiektu powinien (i to jest prawdziwe zaskoczenie) być interfejsem zdalnym.

Aby interfejs mógł być uznany za „zdalny", powinien spełniać trzy poniższe warunki:

- **dziedziczyć po interfejsie `java.rmi.Remote`,**
- **każda jego metoda musi deklarować wyjątek `java.rmi.RemoteException`,**
- **argumenty i wartości wynikowe muszą gwarantować możliwość przesłania (czyli muszą być obiektami `Serializable`, wartościami typów podstawowych itd.).**

Wszystko zaczyna się od zdalnego ***interfejsu****. Zarówno zdalny obiekt, jak i pośrednik implementują ten sam interfejs... zawierający metody, które klient chce wywoływać.*

Zdalny interfejs musi dziedziczyć po interfejsie `java.rmi.Remote`, a każda jego metoda musi deklarować wyjątek `RemoteException`.

Klasa `RemoteException` *oraz interfejs* `Remote` *należą do pakietu java.rmi.*

Zdalny interfejs MUSI dziedziczyć po interfejsie `java.rmi.Remote` *(który nie deklaruje żadnych metod).*

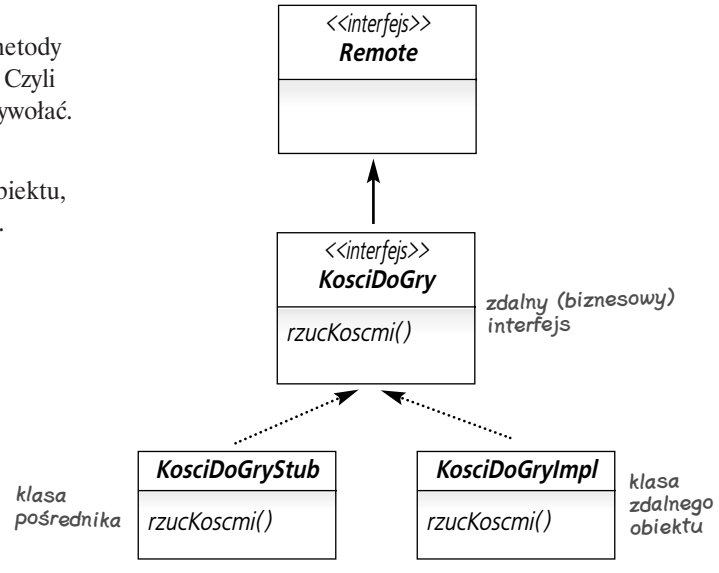
Wszystkie metody zdalnego interfejsu muszą deklarować wyjątek `RemoteException`.

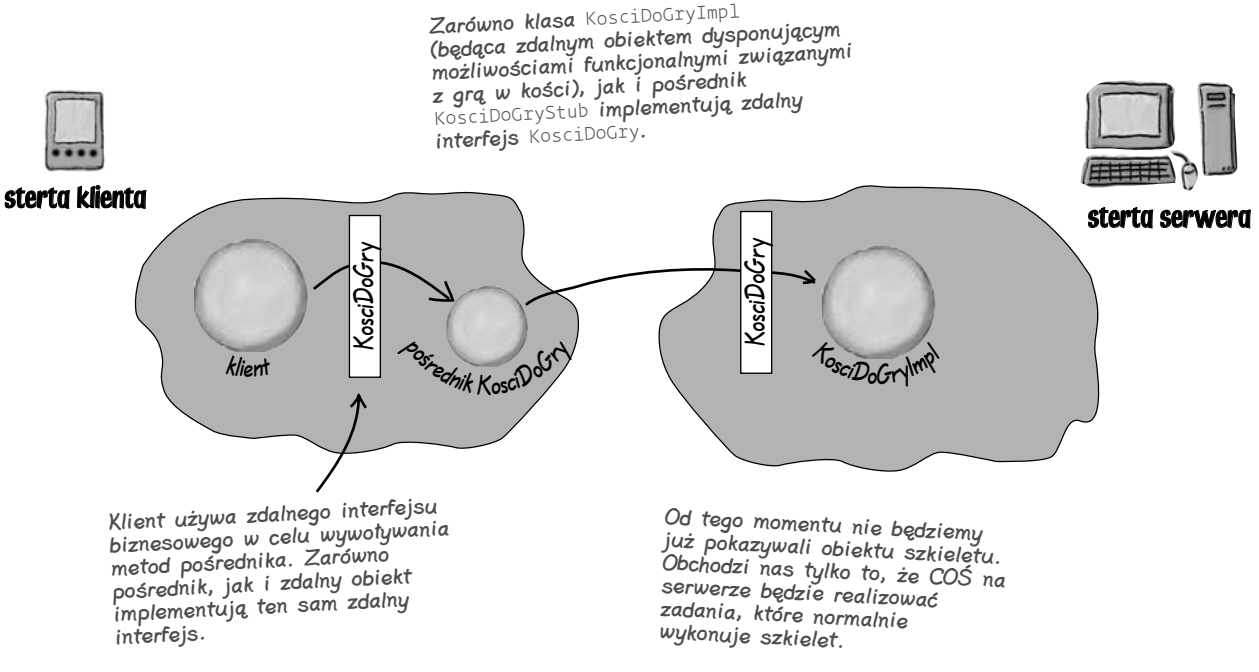
```
import java.rmi.*;

public interface KosciDoGry extends Remote {

   public int rzucKoscmi() throws RemoteException;
}
```

Klient wywołuje metodę biznesową za pośrednictwem zdalnego interfejsu biznesowego

Pamiętaj, że z punktu widzenia klienta, wywołuje on metody Prawdziwego Obiektu. Faktycznego zdalnego obiektu. Czyli obiektu udostępniającego metody, które klient chce wywołać.

Jedynym czynnikiem, który przypomina klientowi, iż w rzeczywistości nie wywołuje on metod zdalnego obiektu, jest konieczność obsługi wyjątków `RemoteException`.

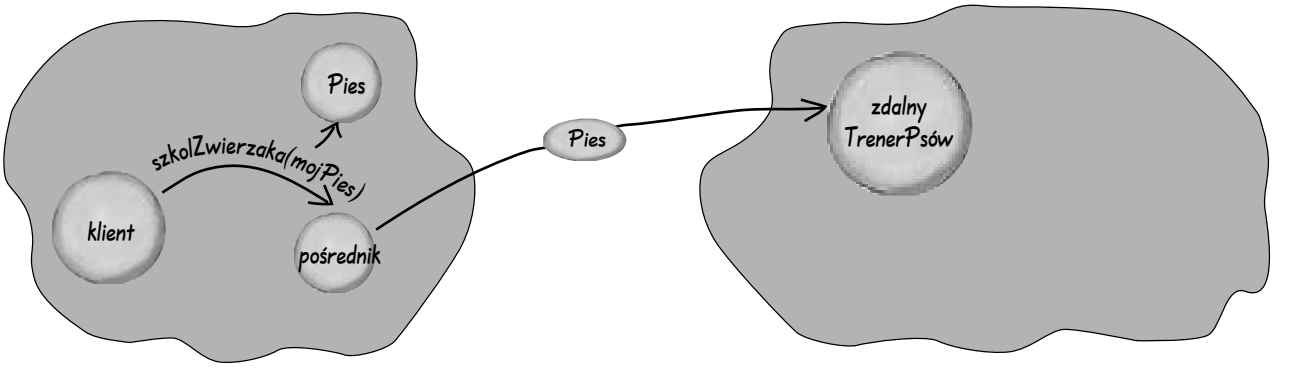


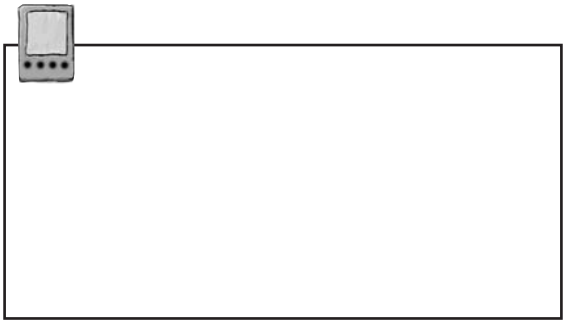


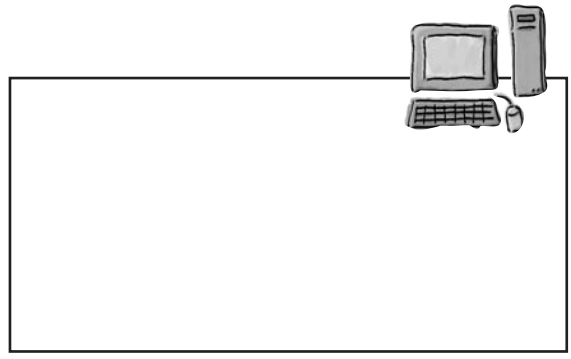
Przygotuj ołówek

To ćwiczenie jest związane z błędem, który programiści używający technologii EJB popełniają najczęściej. Dlatego nie pomijaj go!

Bazując na przedstawionym poniżej schemacie, umieść klasy i interfejsy w odpowiednich prostokątach, reprezentujących odpowiednio klienta i serwer (klasy mogą być używane zarówno po stronie klienta, jak i serwera). Schemat jest uproszczony, zatem nie zostały na nim przedstawione wszystkie elementy aplikacji.







TrenerPsowImpl

interfejs
TrenerPsow

TrenerPsowStub

klasa klienta

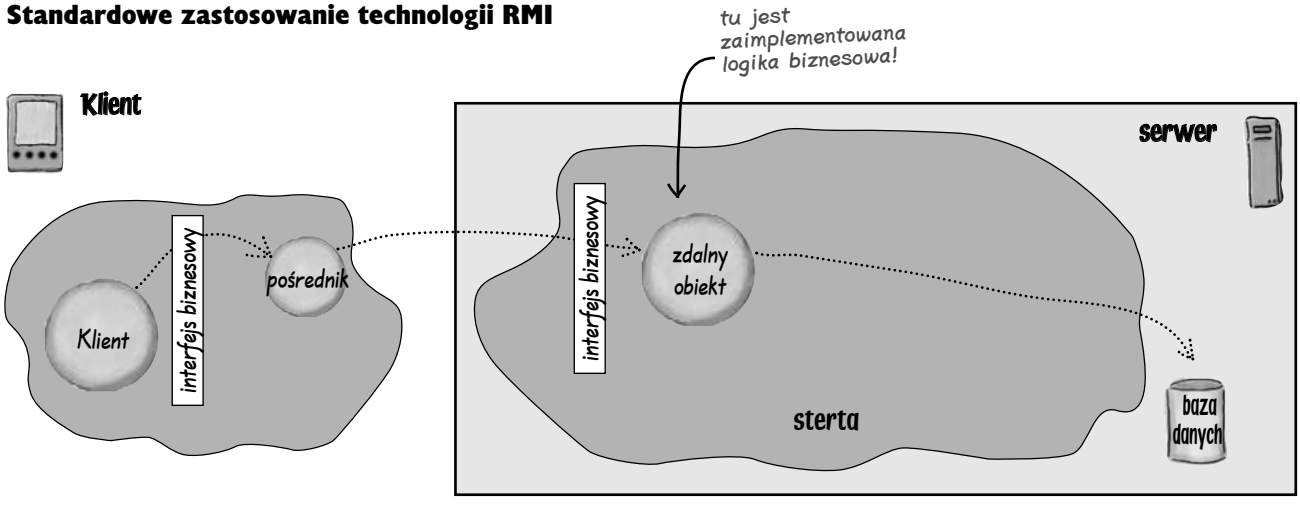
Pies

Sposób zastosowania RMI w technologii EJB

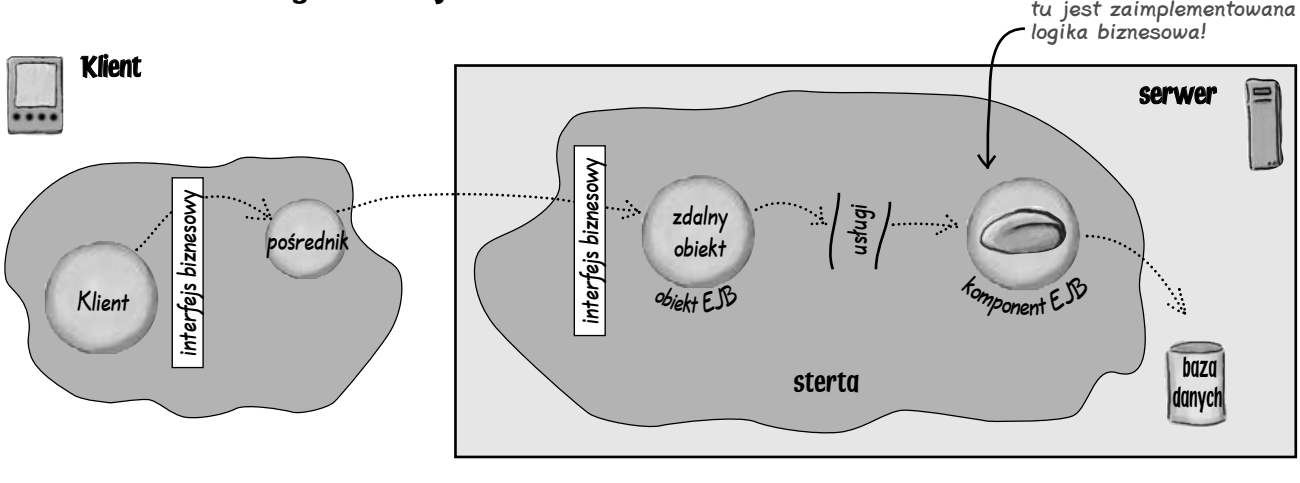
W technologii EJB klient niemal zawsze uzyskuje dostęp do aplikacji korporacyjnej za pomocą referencji (pośrednika) do zdalnego obiektu. Owszem, istnieje możliwość stosowania klientów lokalnych (czyli klientów istniejących na tej samej stercie co komponent, które nie używają RMI do wywoływania metod biznesowych), a w niektórych sytuacjach jest to nawet konieczne. Jednak dotyczy to nieznacznej liczby bardzo szczególnych sytuacji.

A zatem, w przeważającej większości przypadków RMI stanowi podstawę komunikacji pomiędzy klientem i komponentem. Jednak, jak miałeś okazję przekonać się w pierwszym rozdziale, architektura EJB jest nieco bardziej skomplikowana niż prosty schemat komunikacji „klient-pośrednik-obiekt zdalny". W technologii EJB komponent, czyli to coś, co posiada metody biznesowe, które chcemy wywołać, nie jest obiektem zdalnym!

Standardowe zastosowanie technologii RMI



Zastosowanie technologii RMI w EJB



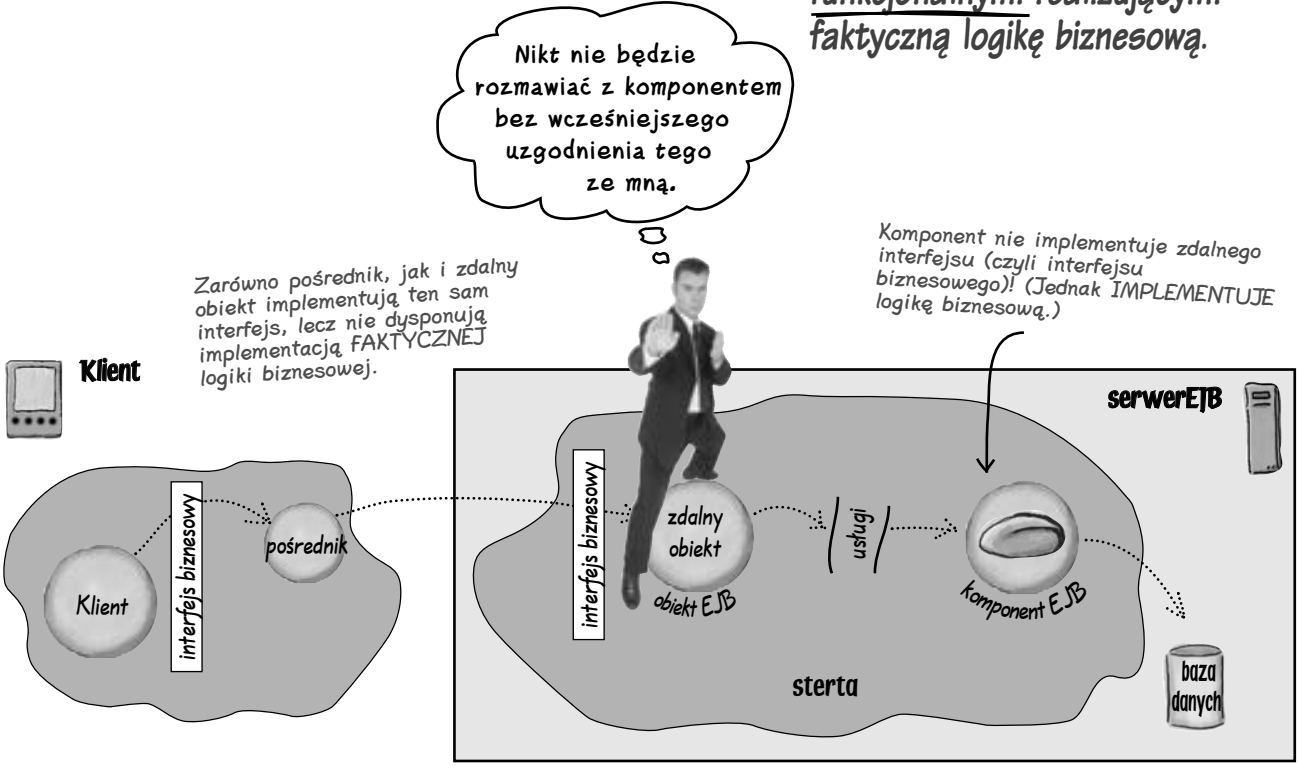
Zdalny obiekt — EJBObject — nie jest komponentem, to strażnik komponentu

Pamiętaj, że w technologii EJB zdalny obiekt (`EJBObject`) pełni funkcję „strażnika" komponentu. Sam komponent trzyma się z tyłu i jest zabezpieczony przed wszystkimi bezpośrednimi odwołaniami ze strony klientów, natomiast to obiekt typu `EJBObject` implementuje zdalny interfejs i przyjmuje wywołania metod. Kiedy wywołanie dotrze do obiektu `EJBObject`, ingeruje w nie serwer, dodając wszelkie usługi, takie jak: bezpieczeństwo (czy klient ma prawo wywoływać metodę?), transakcje (czy to wywołanie jest elementem istniejącej transakcji, czy też należy rozpocząć nową), czy też trwałość (czy przed wykonaniem metody komponent powinien pobrać jakieś informacje z bazy danych?).

Obiekt `EJBObject` implementuje zdalny interfejs biznesowy, a zatem wywołania zgłaszane przez klienta trafiają właśnie do tego obiektu. Jednak to sam komponent implementuje właściwą logikę biznesową, choć z technicznego punktu widzenia nie jest komponentem „zdalnym", gdyż nie implementuje interfejsu `Remote`.

Zarówno zdalny obiekt, jak i pośrednik implementują ten sam interfejs — interfejs biznesowy (nazywany także interfejsem komponentu) — lecz nie dysponują faktyczną logiką biznesową.

Z kolei klasa komponentu NIE implementuje interfejsu biznesowego (oczywiście jedynie z formalnego punktu widzenia), lecz dysponuje możliwościami funkcjonalnymi realizującymi faktyczną logikę biznesową.



Interfejs komponentu

W technologii EJB interfejs biznesowy jest nazywany interfejsem *komponentu*. To właśnie on informuje klienta o dostępnych metodach biznesowych. Podstawowa różnica pomiędzy interfejsem RMI oraz zdalnym interfejsem komponentu polega na tym, że w EJB dziedziczymy po interfejsie `javax.ejb.EJBObject`, a nie po interfejsie `java.rmi.Remote`.

Kluczowe zagadnienia:

1. Każdy interfejs, którego drzewo dziedziczenia zawiera interfejs `java.rmi.Remote`, może być interfejsem zdalnym.

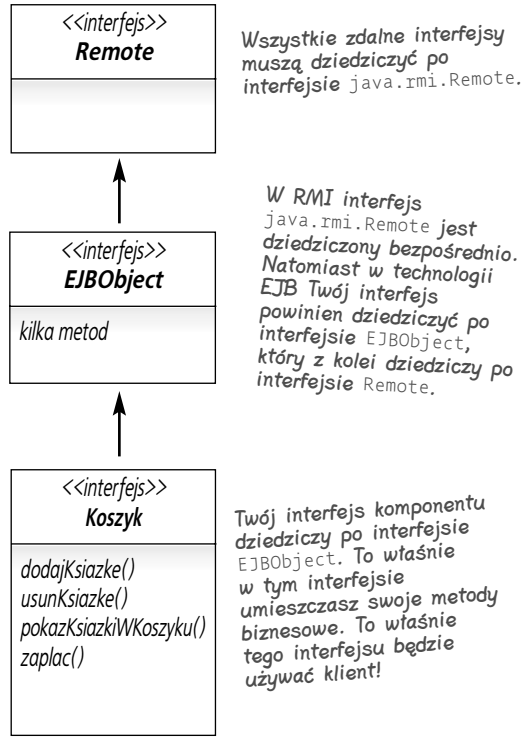
2. Interfejs `EJBObject` dziedziczy po interfejsie `Remote`, a zatem `EJBObject` jest interfejsem zdalnym.

3. Twój zdalny interfejs komponentu musi dziedziczyć po interfejsie `EJBObject`.

   (*Możesz* stworzyć lokalny interfejs komponentu i w takim przypadku zasady postępowania są nieco inne; zajmiemy się tym przypadkiem w rozdziale 3. pt.: *Co widzi klient*.)

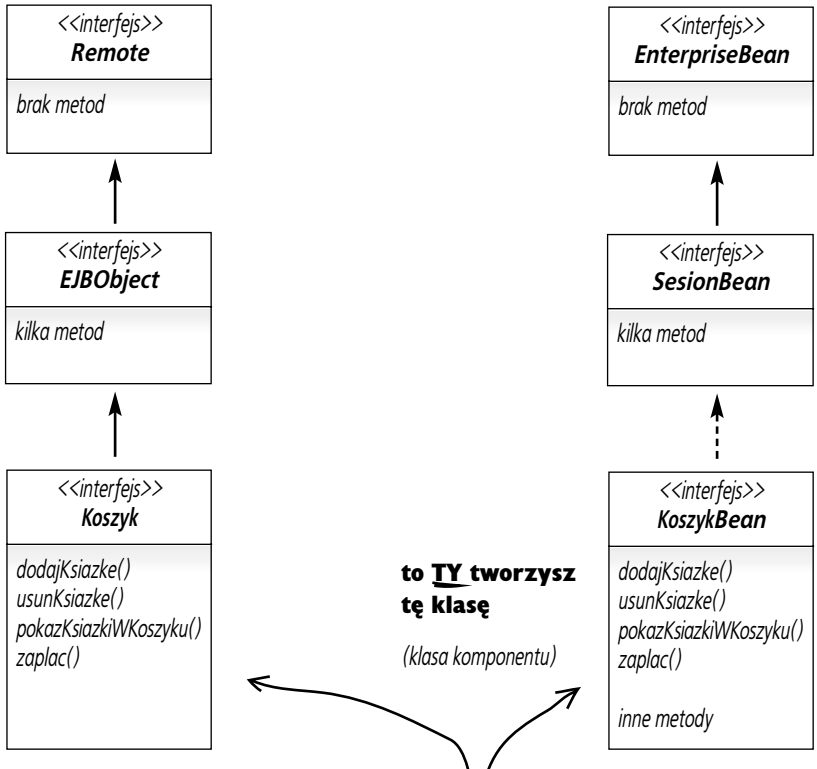
4. Udostępniane metody biznesowe prezentujesz klientowi za pomocą interfejsu komponentu.

5. Interfejs `EJBObject` udostępnia także inne metody, z których może korzystać klient. (Zaprezentujemy je w dalszej części książki.)



Niezależnie od tego jaka klasa implementuje interfejs `Koszyk`, musi ona implementować **wszystkie** metody dwóch interfejsów — `Koszyk` oraz `EJBObject`. Interfejs `EJBObject` zawiera metody, których mogą potrzebować wszystkie klienty EJB.

Jak klasa komponentu pasuje do tej układanki



to TY tworzysz ten interfejs

(zdalny interfejs komponentu)

to TY tworzysz tę klasę

(klasa komponentu)

Te same metody, które jednak nie są ze sobą bezpośrednio powiązane z punktu widzenia języka. Klasa `KoszykBean` nie implementuje interfejsu `Koszyk`!

(a jednak wygląda tak, jak gdyby implementowała...)

Twórca komponentu musi upewnić się, że klasa komponentu ma te same metody, co interfejs komponentu. Metody w klasie oraz w interfejsie muszą być identyczne, jak gdyby klasa IMPLEMENTOWAŁA interfejs komponentu.

NAPRAWDĘ musisz znać te interfejsy!

Podchodząc do egzaminu, musisz dokładnie wiedzieć gdzie jest deklarowana każda z metod. Musisz wiedzieć, które z nich należą do interfejsu `EJBObject`*, a które, na przykład, do interfejsu* `SessionBean`*. Musisz także znać dokładną sygnaturę każdej z tych metod. Wszystkie te interfejsy zostaną szczegółowo opisane w kilku kolejnych rozdziałach, a gdy zaczniemy je analizować, lepiej się skoncentruj i wytęż uwagę!*

Nie ma niemądrych pytań

P. Chciałbym mieć pewność, że wszystko dobrze zrozumiałem... interfejsy mogą dziedziczyć po innych interfejsach?

O. Tak, interfejsy mają swoje własne drzewo dziedziczenia. W rzeczywistości interfejsy pozwalają na coś, czego nie można zrobić z klasami — na to, by jeden interfejs dziedziczył po kilku innych interfejsach jednocześnie!

```
interface Koszyk extends EJBObject,
LogikaBiznesowaKoszyka
```

P. Ale co to oznacza, że jeden interfejs dziedziczy po innym? No bo właściwie co interfejs może dziedziczyć?

O. W przypadku interfejsów dziedziczenie oznacza, iż interfejs dziedziczący będzie dysponować wszystkim, czym dysponuje jego interfejs (lub interfejsy) bazowy. Ktokolwiek będzie implementować taki interfejs, musi zaimplementować nie tylko jego metody, lecz także wszystkie metody każdego interfejsu bazowego... aż do samego wierzchołka drzewa dziedziczenia.

A zatem, w powyższym przykładzie ktokolwiek będzie implementować interfejs `Koszyk`, musi także zaimplementować wszystkie metody interfejsu `EJBObject`.

P. Dlaczego w komponencie nie jest implementowany interfejs Remote (interfejs biznesowy)? Czy nie po to właśnie są stosowane interfejsy — aby kompilator mógł wymusić uczciwość programisty, oraz by móc wykorzystać mechanizmy kontroli typów?

Implementując interfejs, musimy zaimplementować *wszystkie* metody, które interfejs ten odziedziczył po swoich interfejsach bazowych.

Oznacza to, że ktokolwiek zajmie się implementacją interfejsu `Koszyk`, będzie musiał zaimplementować metody zarówno interfejsu `Koszyk` *jak i* `EJBObject`.

O. Już wcześniej zadałeś to pytanie. Ale nie przejmuj się, wszyscy o czymś zapominamy, dlatego przypomnimy Ci odpowiedź. Komponent nie implementuje interfejsu Remote, gdyż nigdy nie powinien być obiektem zdalnym (w terminologii technologii RMI). Innymi słowy, nie chcesz, by ktokolwiek i kiedykolwiek dysponował pośrednikiem do samego komponentu! Starając się w jakiś sposób przemycić i udostępnić klientom referencję do komponentu (czyli pośrednika komponentu), niszczysz podstawowy cel technologii EJB! Jeśli pozwolisz, by klient komunikował się bezpośrednio z komponentem, to serwer nie będzie w stanie udostępnić Ci swoich usług, a jeśli nie potrzebujesz usług serwera, to... sam wiesz dokąd zmierza to rozumowanie.

Z technicznego punktu widzenia implementowanie w komponencie interfejsu `Remote` nie jest błędem. Jednak stosowanie takiego rozwiązania jest fatalnym pomysłem, gdyż sprawia, że możesz popełnić błędy, które nie zostaną wykryte na etapie kompilacji (i uwidocznią się dopiero podczas działania aplikacji). Jednak wcale nie musisz stosować takiego rozwiązania, gdyż niemal wszystkie narzędzia programistyczne (z praktycznie wszystkimi zintegrowanymi środowiskami programistycznymi wspomagającymi tworzenie komponentów *EJB* włącznie) doskonale znają wzajemne zależności występujące pomiędzy komponentem, interfejsem `EJBObject` oraz `Remote` i są w stanie zagwarantować, że zarówno interfejs komponentu, jak i komponent będą mieć te same metody.

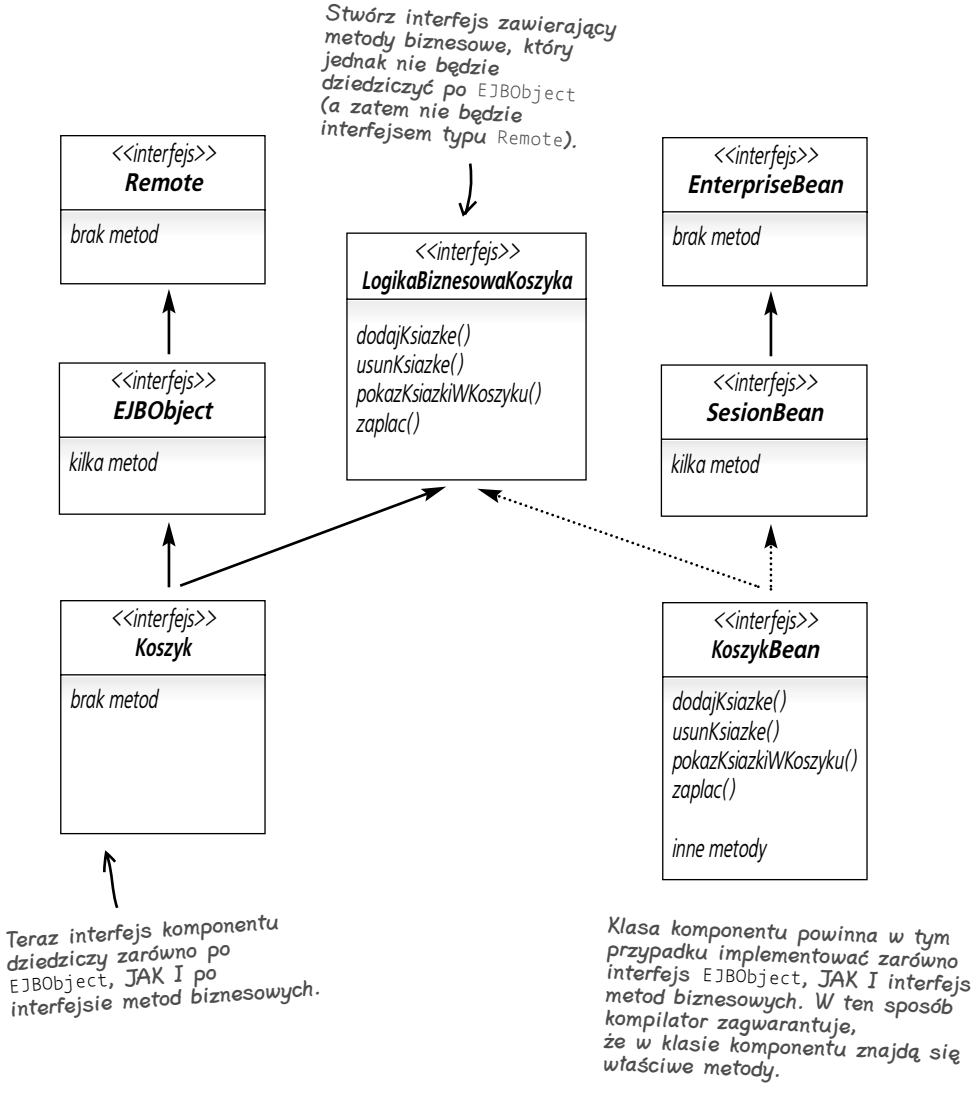
P. No dobrze, ale jeśli to całe rozwiązanie jest dla mnie zbyt stresujące? Taki pomysł kłóci się z moją programistyczną wrażliwością, pozostaje w sprzeczności z podstawowymi zasadami, którymi kieruję się, pisząc programy w Javie. Jestem pewny, że istnieje jakieś inne rozwiązanie!

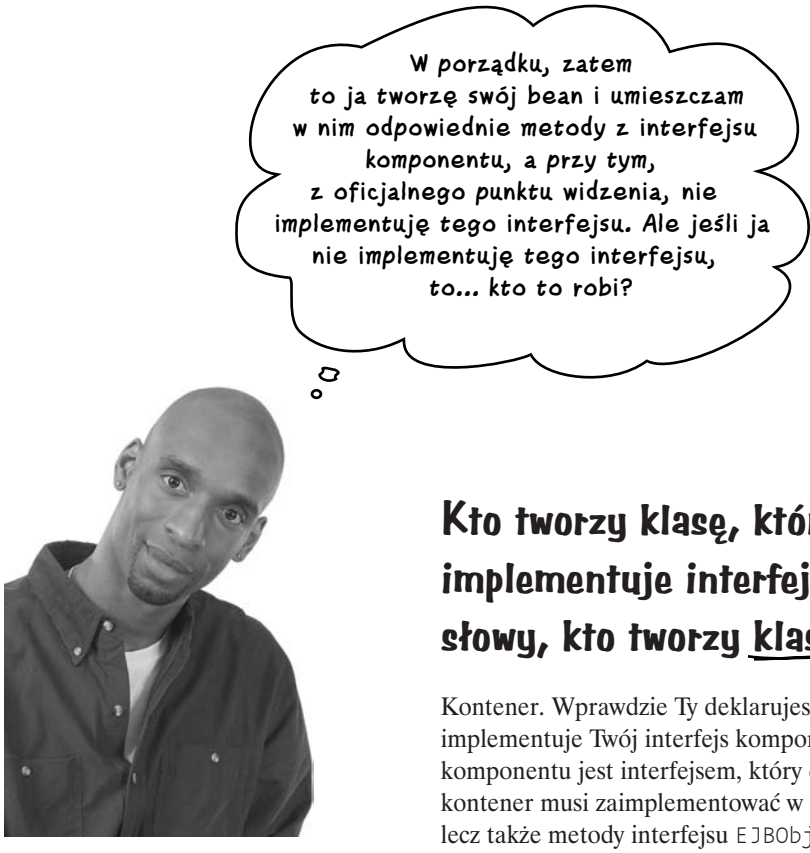
O. Mógłbyś nam zaufać i uwierzyć stwierdzeniu, że „niemal na pewno będziesz używać narzędzi, więc to naprawdę nie ma znaczenia. Poważnie"; ale nie... No dobrze, zatem faktycznie jest coś, co mógłbyś zrobić i co pewnie pozwoli, byś poczuł się lepiej. Opisaliśmy to rozwiązanie na następnej stronie, lecz prawdopodobnie sam możesz zgadnąć na czym ono polega.

Niemniej jednak, obstajemy przy swojej opinii, że większość programistów nie będzie musiała uciekać się do tego rozwiązania.

Własnymi ścieżkami

Rozwiązanie dla tych, których sama myśl o tym, że komponent może nie implementować interfejsu metod biznesowych napawa obrzydzeniem.





Kto tworzy klasę, która w rzeczywistości implementuje interfejs komponentu? Innymi słowy, kto tworzy klasę EJBObject?

Kontener. Wprawdzie Ty deklarujesz metody, lecz to właśnie kontener implementuje Twój interfejs komponentu. Pamiętaj, że interfejs komponentu jest interfejsem, który dziedziczy po `EJBObject`, a zatem, kontener musi zaimplementować w nim nie tylko Twoje metody biznesowe, lecz także metody interfejsu `EJBObject` (których jeszcze nie poznaliśmy).

P. Ale skąd kontener wie co należy umieścić w tych metodach? Przecież to ja je deklaruję...

O. Pamiętaj, że kontener nie implementuje faktycznej logiki biznesowej. Prawdziwe możliwości funkcjonalne metod biznesowych znajdują się w klasie komponentu — którą Ty implementujesz. Klasa implementująca interfejsu komponentu posłuży do utworzenia obiektu `EBJObject`. „Strażnika" komponentu. Obiektu zdalnego. A nie zapominaj, że obiekt `EJBObject` tylko udaje, że jest komponentem. Może on odpowiadać na wywołania zdalnych metod przesyłane przez klienty (za pomocą pośredników), jednak jego zadanie polega wyłącznie na przechwytywaniu wywołań kierowanych do komponentu. Wszystko co dzieje się potem zależy wyłącznie od kontenera (lub serwera).

Tak naprawdę nie wiemy w jaki sposób jest implementowany obiekt typu `EJBObject`, to w całości zależy od twórców kontenera lub serwera. Jednak właściwie w ogóle nas to nie obchodzi. Wystarczy byś wiedział, że specyfikacja wymaga, by kontener EJB był w stanie wygenerować kod klasy `EJBObject` (oraz odpowiadającego jej pośrednika).

Kto tworzy poszczególne klasy?

Wiesz, że w przypadku komponentów widocznych dla zdalnych klientów (czyli komponentów, do których zdalne klienty mają dostęp), to Ty musisz napisać interfejs komponentu oraz klasę komponentu. Jednak *ktoś* musi stworzyć klasę implementującą Twój interfejs komponentu (aby można było stworzyć obiekt typu `EJBObject`), jak również *ktoś* musi stworzyć pośrednika, który będzie współpracować z obiektem `EJBObject`. Tym „kimś" jest kontener. Aby poniższe informacje były kompletne, wymieniliśmy w nich także interfejs oraz klasy obiektu bazowego, choć do tej pory jeszcze ich nie opisywaliśmy.

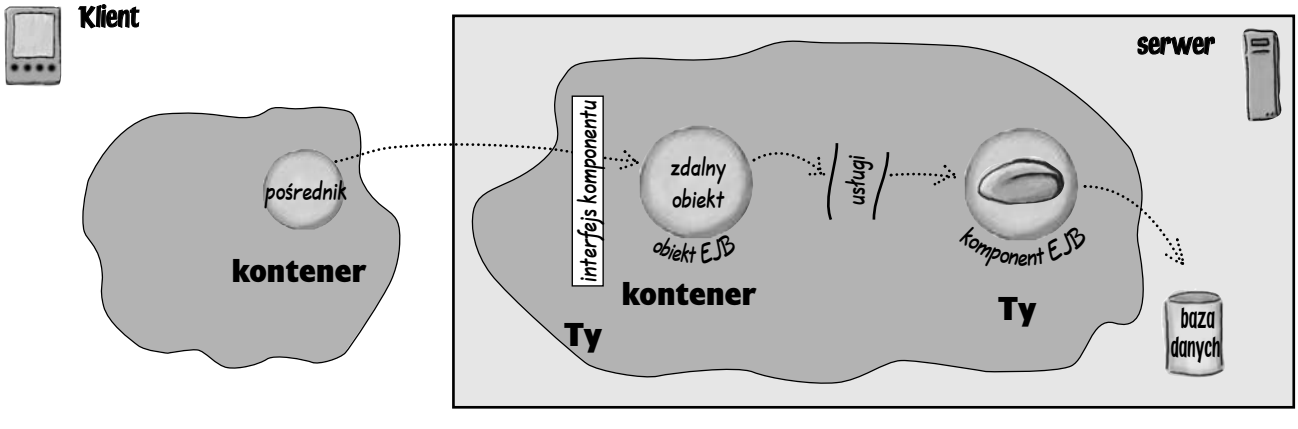
Ty

1. **Interfejs komponentu** (dziedziczy po `javax.ejb.EJBObject`).
2. **Klasa komponentu** (implementuje `javax.ejb.SessionBean` lub `javax.ejb.EntityBean`).
3. **Interfejs obiektu bazowego** (dziedziczy po `javax.ejb.EJBHome`; podyskutujemy na jego temat na następnej stronie).

Nie pisaliśmy jeszcze na temat interfejsu oraz klas obiektu bazowego, dlatego też nie przedstawiliśmy ich na poniższym rysunku.

Kontener

1. Klasa `EJBObject` (implementuje interfejs komponentu).
2. Klasa pośrednika `EJBObject` (implementuje interfejs komponentu i wie jak należy komunikować się z obiektem `EJBObject`).
3. Klasa obiektu bazowego (implementuje interfejs obiektu bazowego).
4. Klasa pośrednika obiektu bazowego (implementuje interfejs obiektu bazowego i wie jak należy się z nim komunikować).



Zasłyszane w POCZEKALNI DLA KOMPONENTÓW

EJBObject: Hej Beanku... czy nie denerwuje Cię to, że zawsze mieszam się do wszystkiego co robisz? Czy nie chciałbyś czasami pogadać z kimś bezpośrednio?

Komponent: Nie. Jestem zbyt ważny. Zbyt cenny. A poza tym nie mam najmniejszego zamiaru zabezpieczać swoich własnych wywołań. W końcu to wy od tego jesteście.

EJBObject: My?

Komponent: No, wy — wszyscy ludzie kontenera. Czyli obiekt bazowy, pośrednicy... wy wszyscy. Moim zadaniem jest obsługa złożonej logiki biznesowej. Krytycznych funkcji, od których może zależeć sukces lub porażka operacji wykonywanych w środowisku korporacyjnym.

EJBObject: (O rany... to brzmi jak gadka gości od marketingu.) No dobra, rozumiem, że dysponujesz ważnymi metodami, ale wciąż nie łapię dlaczego ja muszę uczestniczyć w wywoływaniu każdej z tych metod.

Komponent: Słuchaj, moja praca jest zbyt ważna, aby byle klient, który w ogóle nie ma potrzeby kontaktowania się ze mną, mógł ją przerywać swoimi wywołaniami. Czy naprawdę uważasz, że mam zamiar sprawdzać wszystkich klientów i upewniać się, że mają prawo kontaktować się ze mną? Jakbym nie miał ważniejszych rzeczy do roboty...

EJBObject: W porządku, czyli tak naprawdę chodzi o bezpieczeństwo, ale w takim razie nie rozumiem dlaczego nie możesz zawierać kodu sprawdzającego prawa, jakimi dysponuje każdy klient. To pozwoliłoby na duże oszczędności (zwłaszcza dla MNIE).

Komponent: Przede wszystkim, bezpieczeństwo jest tylko JEDNYM z powodów, dla których obsługujesz wywołania kierowane do mnie. Za chwilę mogę Ci je podać. A wracając do umieszczania kodu związanego z zabezpieczeniami we mnie — mogę to zrobić, jeśli programista sobie tego zażyczy. Ale nie jest to przyjęty sposób zapewniania bezpieczeństwa.

EJBObject: A co w tym złego, że sam będziesz zabezpieczał swoje metody?

Komponent: Poważnie nie wiesz? [wznosi oczy do nieba] Przede wszystkim, gdy mechanizmy zabezpieczeń zostaną umieszczone w moich metodach, to zostaną ograniczone możliwości wielokrotnego stosowania mnie w różnych aplikacjach. Jednym z celów EJB jest zapewnienie możliwości konfigurowana i dostosowywania komponentu do potrzeb aplikacji w trakcie eksploatacji, a nie poprzez wprowadzanie zmian w kodzie źródłowym. Gdyby mechanizmy zabezpieczeń były umieszczone we mnie, to nie można by ich zmienić bez wprowadzania modyfikacji w moim kodzie źródłowym. A kto by tego chciał?

EJBObject: Hm... To ma sens. Można umieścić informacje o sposobie zabezpieczania w XML-owym deskryptorze rozmieszczenia, a kiedy odbiorę wywołanie, serwer będzie w stanie sprawdzić czy konkretny klient ma niezbędne uprawnienia. Ale co w sytuacji, gdy nie zwracasz uwagi na kwestie zabezpieczeń? W sytuacji, gdy ktoś, kto Cię zainstalował nie zwraca uwagi na to kto wywołuje Twoje metody?

Komponent: Nie jesteś zbyt bystry... nieprawdaż? Cóż... ale przynajmniej możesz podnosić ciężkie obiekty... POMYŚL o wszystkich innych sprawach, które mają znaczenie; takich jak transakcje bądź trwałość.

EJBObject: Fakt... zapomniałem o transakcjach. W porządku, transakcje też mają sens. W końcu, aby metoda mogła być wykonana w ramach transakcji, przed jej wywołaniem serwer musi sprawdzić czy istnieje kontekst transakcji. Niezależnie czy to będzie kontekst Twój czy wywołującego klienta...

Komponent: Otóż to...

EJBObject: No ale co z tą trwałością?

Komponent: Cóż, zastanów się przez chwilę nad komponentami entity bean. Jeśli jestem takim komponentem, oznacza to, że reprezentuję jakieś dane pochodzące z trwałego magazynu i...

EJBObject: Chwila — czy mówiąc „trwały magazyn" nie masz na myśli po prostu BAZY DANYCH? Dlaczego po prostu nie powiesz, że te informacje pochodzą z bazy danych?

Komponent: O rany, co za gość. Wyrażenie „trwały magazyn" nie jest synonimem terminu „baza danych". Ale jeśli poczujesz się lepiej, wyobrażając to sobie w taki sposób, proszę bardzo. Ale jak już mówiłem, jeśli będę reprezentować jakieś dane, na przykład, dane klienta Jaś Wędrowniczek, to co się stanie kiedy klient wywoła moją metodę `getAdres()`? Serwer nie może tak po prostu przekazać do mnie wywołania tej metody!

EJBObject: Ponieważ...

Komponent: Ponieważ w pierwszej kolejności muszę pobrać informacje o Jasiu Wędrowniczku z bazy danych!

EJBObject: Ponieważ...

Komponent: Ponieważ mógłbym zwrócić nieprawidłowy, nieważny adres! Jeśli wciąż znajduję się w obrębie wcześniejszej transakcji, to serwer musi dać mi znać, abym odczytał aktualne informacje o Jasiu Wędrowniczku z bazy danych ZANIM każe mi wykonać metodę `getAdres()`. W przeciwnym razie nikt nie wie co bym zwrócił. No dobra, to było ostatnie wywołanie, więc dokończymy rozmowę innym razem.

Nie ma niemądrych pytań

P. Jak i kiedy kontener tworzy obiekt `EJBObject`, obiekt bazowy oraz pośredników?

O. Podczas wdrażania komponentu kontener analizuje deskryptor rozmieszczenia i pobiera komponent ze wskazanego miejsca. Pamiętaj, że deskryptor zawiera informacje o w pełni kwalifikowanej nazwie zdalnego interfejsu komponentu (`EJBObject`) oraz interfejsu zdalnego obiektu bazowego. Gdy kontener zdobędzie informacje o tych interfejsach, wygeneruje kod dwóch klas, które je implementują. A ponieważ oba te interfejsy dziedziczą po interfejsie `Remote`, kontener wygeneruje także klasy pośredników dla tych nowych klas.

P. Czy to zawsze są zwyczajne obiekty pośredników RMI? Podczas wdrażania komponentu na serwerze pojawił się komunikat o wykonywaniu programu rmic...

O. Kontener może generować klasy pośredników w całkowicie dowolny sposób, jedynym ograniczeniem jest wymóg, aby były one zgodne z RMI-IIOP. W celu zaimplementowania możliwości funkcjonalnych pośredników serwer może wykorzystać rozwiązanie nazywane „dynamicznymi pośrednikami" (ang. *dynamic proxies*), ale nas to w ogóle nie obchodzi. Mówiąc „pośrednik", mamy na myśli cokolwiek, co ma możliwości funkcjonalne pośrednika. A to czy jest to pośrednik RMI, czy też coś innego, to jedynie szczegół implementacyjny. Zagadnieniem tym zajmiemy się bardziej szczegółowo w następnym rozdziale. Najważniejsze jest jednak to, iż w rzeczywistości nie wiemy jak wygląda kod klasy pośrednika. A zatem, nie wiemy jak jest zaimplementowany obiekt typu `EJBObject` oraz obiekt bazowy. Co więcej, nawet nie powinniśmy tego wiedzieć.

Być *może*, używany kontener EJB pozwala na oglądnięcie generowanego przez siebie kodu źródłowego (a może nawet na jego *modyfikację*), niemniej jednak nie powinieneś na to liczyć. A na Twoim miejscu nie staralibyśmy się ani oglądać tego kodu, ani go modyfikować, nawet gdybyśmy mieli taką możliwość.

P. A zatem, implementacja tych klas zależy wyłącznie od twórców serwera lub kontenera EJB?

O. Tak! Twórcy mogą wybierać różne sposoby implementacji i starać się uzyskać poprawę wydajności poprzez zastosowanie odpowiedniej postaci obiektów pośredników, obiektów bazowych oraz obiektów `EJBObject`. Ale powtarzamy, te zagadnienia nie powinny Cię nawet interesować, a tym bardziej nie powinieneś w nich czegokolwiek zmieniać.

Specyfikacja wymusza (także na Tobie) i gwarantuje jedynie to, iż obiekty typu `Remote` muszą być zgodne z regułami technologii RMI-IIOP, czyli wersji RMI korzystającej z protokołu transportowego IIOP (należącego do architektury CORBA).

P. Skoro już wspomnieliście o tym, to czym różni się RMI-IIOP od zwyczajnego RMI?

O. W pierwotnej wersji RMI używany był protokół transportowy JRMP. Z kolei protokół IIOP pozwala, by zdalne obiekty współpracowały, korzystając z technologii CORBA (w niniejszej książce nie będziemy szczegółowo zajmować się tą technologią, może z wyjątkiem krótkich wzmianek umieszczonych w kilku rozdziałach; CORBA bezsprzecznie wykracza poza zakres egzaminu oraz poza ramy tematyczne książki).

Protokół IIOP sprawia, że z Twoich obiektów będą mogły korzystać klienty napisane także w innych językach niż Java. Określa on sposób przekazywania wraz z wywołaniem metody informacji dotyczących transakcji i bezpieczeństwa, z których może skorzystać używany kontener.

W większości przypadków nawet nie będziesz zauważać różnicy pomiędzy początkową wersją RMI a RMI-IIOP. Choć... jest kilka różnic pomiędzy obydwiema wersjami RMI, a jedna z nich bez wątpienia pojawi się na egzaminie. Zagadnieniem tym jest „zawężanie" i zajmiemy się nim bardziej szczegółowo w następnym rozdziale. Na razie wystarczy, jeśli będziesz wiedział, że jest to coś, co klient musi zrobić z pośrednikiem EJB, a czego nie musi robić ze zwyczajnym pośrednikiem; a konieczność wykonania tej operacji wynika z faktu, iż specyfikacja EJB nakazuje, by przyjąć założenie, że pośrednik korzysta z protokołu IIOP, a zatem może być pośrednikiem innego *rodzaju*.

P. A kiedy porozmawiamy o obiekcie bazowym?

O. Na następnej stronie.

P. Dlaczego tak długo zwlekaliście z rozmową na ten temat? Czy obiekt bazowy nie jest ważny?

O. Niezależnie od tego jak ważny jest obiekt bazowy, zazwyczaj stanowi on jedynie sposób na *zdobycie* referencji do jakiegoś obiektu, który implementuje interfejs komponentu. Innymi słowy, obiektu bazowego możesz użyć, aby uzyskać pośrednika do obiektu `EJBObject` (używanego przez zdalnych klientów, bo tylko o takich do tej pory pisaliśmy).

Znaczenie obiektu bazowego jest nieco większe w przypadku stosowania komponentów *entity bean* (o czym się przekonamy), jednak i tak sprowadza się ono głównie do umożliwienia zdobycia pośrednika do obiektu `EJBObject`. Na późniejszych etapach przeważająca większość komunikacji pomiędzy klientem i komponentem jest realizowana poprzez obiekt `EJBObject`, a nie poprzez obiekt bazowy. W praktyce, w przeważającej większości przypadków klienty używają obiektów bazowych jedynie w celu zdobycia referencji do obiektu `EJBObject`, potem referencja do obiektu bazowego staje się niepotrzebna i można się jej pozbyć.

Obiekt bazowy komponentów

Każdy komponent *session bean* i *entity bean* ma swój obiekt bazowy.

Komponenty *message-driven bean* nie posiadają obiektów bazowych, gdyż nie są widoczne dla klientów (innymi słowy, klienty nie mogą pobrać referencji do komponentów tego typu).

Obiekty bazowe mają jedno podstawowe zadanie: zwracanie referencji do interfejsu komponentu. W przypadku stosowania komponentów *session bean* będzie to jedyne zastosowanie obiektów bazowych. Z kolei w aplikacjach wykorzystujących komponenty *entity bean* znaczenie obiektów bazowych będzie znacznie większe.

Każdy wdrożony komponent posiada swój obiekt bazowy, który jest odpowiedzialny za wszystkie egzemplarze komponentu tego typu. Na przykład, jeśli wdrożyłeś komponent *session bean* typu `Koszyk`, to kontener utworzy jeden obiekt bazowy dla komponentów typu `Koszyk`. Obiekt ten będzie zarządzać wszystkimi egzemplarzami komponentów `Koszyk`. Innymi słowy, jeśli 2000 klientów będzie chciało uzyskać referencję do komponentu `Koszyk` (co, jak zapewne pamiętasz, oznacza referencję do interfejsu komponentu beanu `Koszyk`), to wszystkie 2000 referencji zostanie udostępnionych przez jeden jedyny obiekt bazowy `Koszyk`.

Jeśli w ramach aplikacji wdrożysz trzy komponenty, na przykład: `Koszyk`, `Towar` oraz `Klient`, to na serwerze pojawią się trzy obiekty bazowe, z których każdy będzie reprezentować konkretny komponent. Nie ma znaczenia ile obiektów `EJBObject` oraz ile pośredników utworzą i zwrócą obiekty bazowe — zawsze będą tylko trzy obiekty bazowe.

A zatem, czy to oznacza, że każdemu obiektowi bazowemu odpowiada tylko jeden egzemplarz klasy implementującej interfejs obiektu bazowego konkretnego typu? Niekoniecznie; jednak my mamy właśnie tak myśleć. Właśnie dlatego cały czas używamy i będziemy używać terminu *obiekt bazowy*; podobnie przyjmiemy, że dla każdego wdrożonego typu komponentu będzie istnieć tylko jeden obiekt bazowy. Specyfikacja zapewnia, że możemy traktować obiekty bazowe właśnie w taki sposób, niezależnie od faktycznej implementacji zastosowanej przez twórców kontenera EJB czy serwera.

Każdy wdrożony komponent *session bean* oraz *entity bean* posiada obiekt bazowy. Na przykład, komponent `DoradcaBean` posiada własny, unikatowy dla siebie obiekt bazowy. Niezależnie od tego ile klientów pobierze referencję do komponentu `DoradcaBean`, i tak będzie tylko jeden obiekt bazowy tego typu.

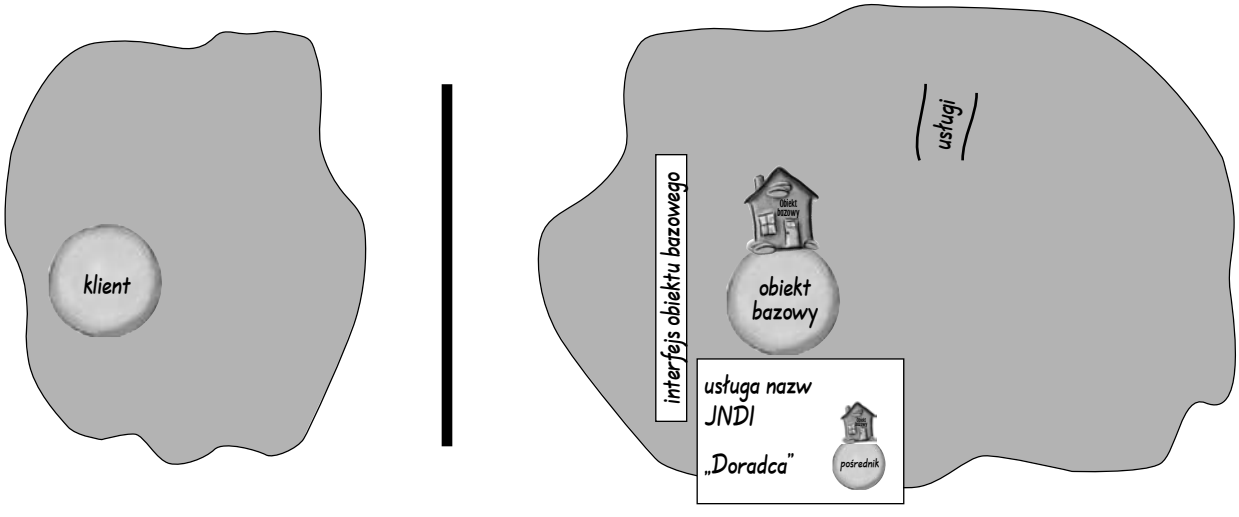
Zadaniem obiektu bazowego jest udostępnianie referencji do interfejsu komponentu.

(Z technicznego punktu widzenia całe to zagadnienie jest nieco bardziej złożone, gdyż komponent może posiadać zarówno lokalny, jak i zdalny obiekt bazowy, choć jest to bardzo mało prawdopodobne. Jednak nawet w takim przypadku istniałby tylko jeden lokalny oraz jeden zdalny obiekt bazowy, niezależnie od tego ile komponentów tego typu istniałoby w aplikacji.)

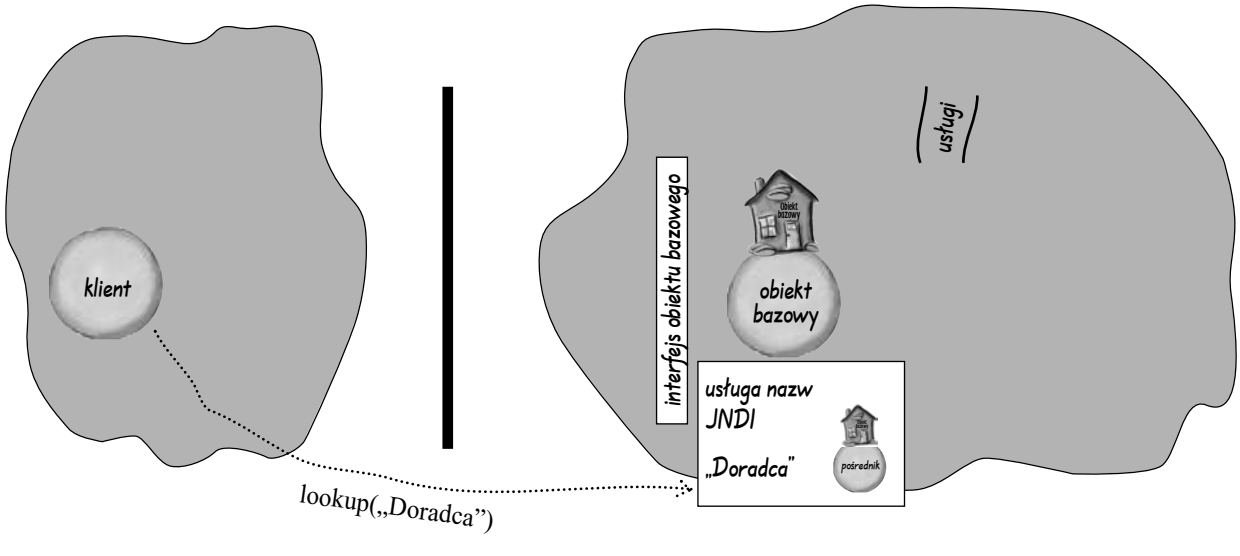
Pobieranie i korzystanie z obiektu bazowego dla komponentu `DoradcaBean`

(Przedstawiony poniżej scenariusz zakłada, że `DoradcaBean` jest stanowym komponentem *session bean*.)

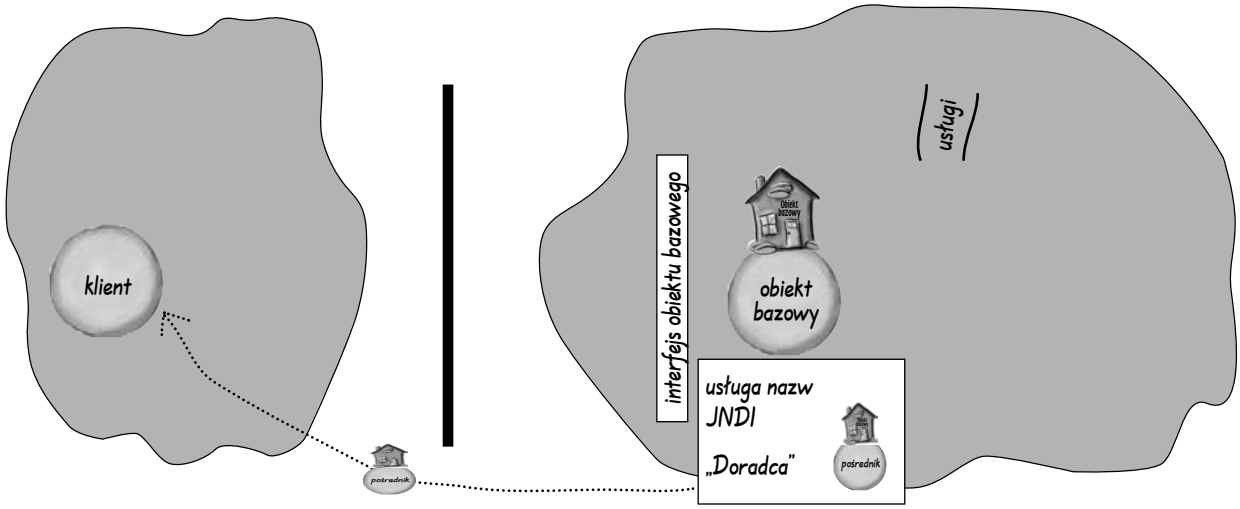
❶ **Komponent DoradcaBean zostaje wdrożony, a serwer tworzy jeden egzemplarz obiektu bazowego DoradcaBean i rejestruje go w usłudze JNDI.**



❷ **Klient przeszukuje rejestr JNDI, poszukując w nim zarejestrowanej nazwy „Doradca".**

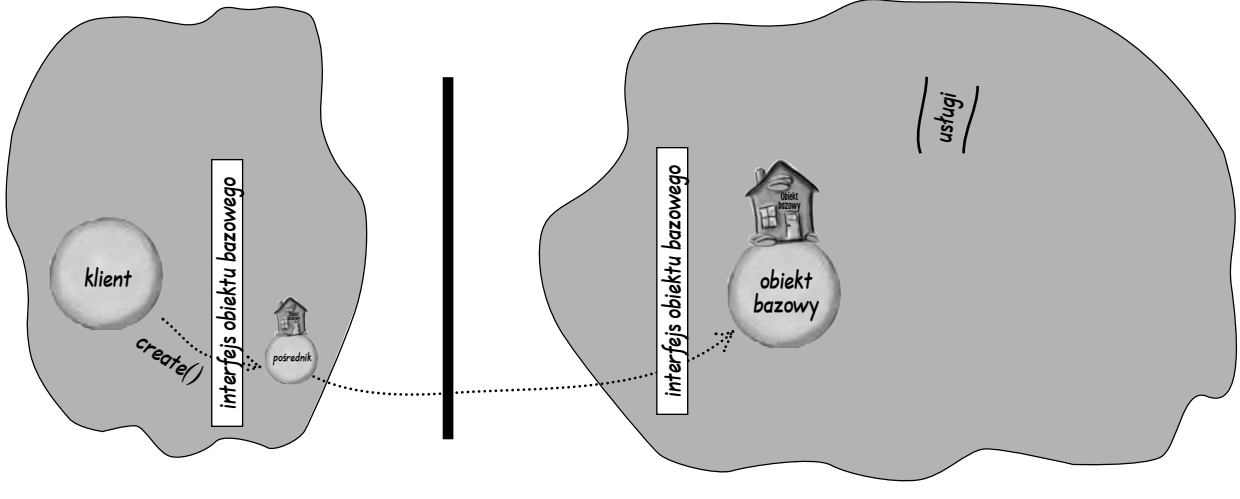


❸ **JNDI przesyła z powrotem pośrednika od zdalnego obiektu bazowego.**



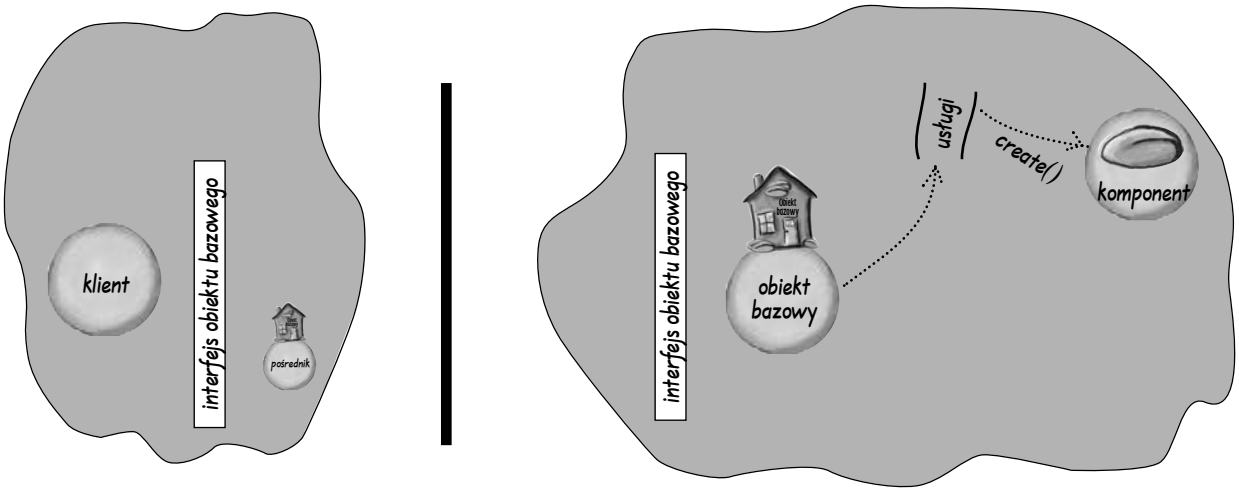
❹ **Klient prosi obiekt bazowy o zwrócenie referencji do interfejsu komponentu, wywołując w tym celu metodę** `create()`**.**

(Innymi słowy, klient chce „utworzyć" komponent i pobrać pośrednika do obiektu `EJBObject` **chroniącego ten komponent.)**

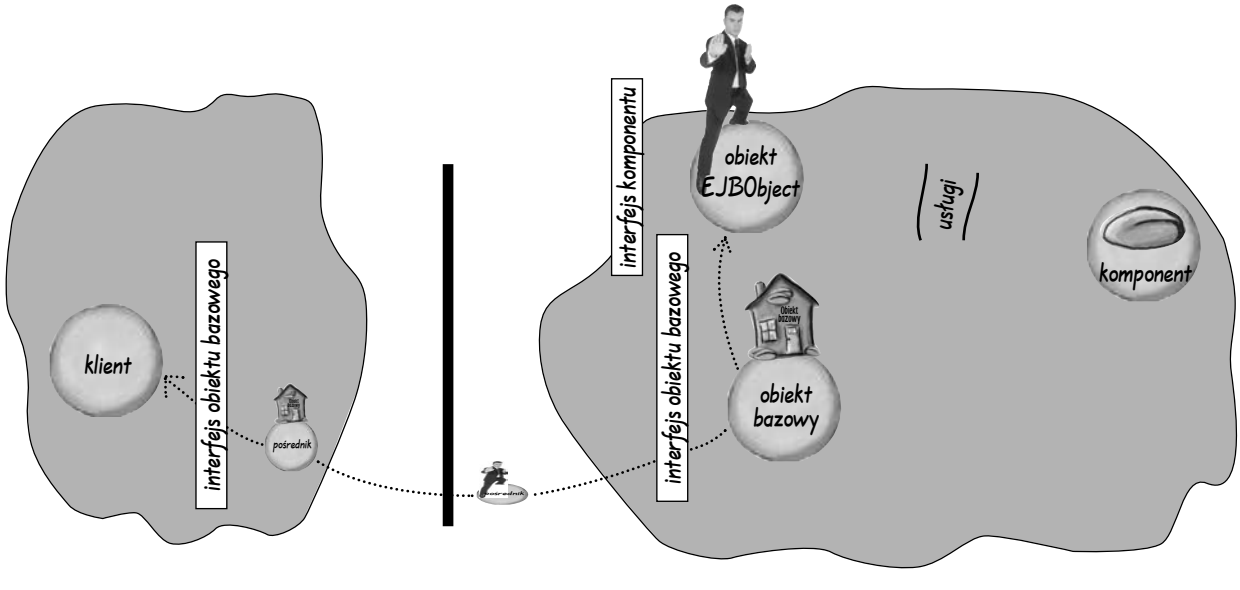


Obiekt bazowy tworzy obiekt EJBObject i przesyła klientowi pośrednika

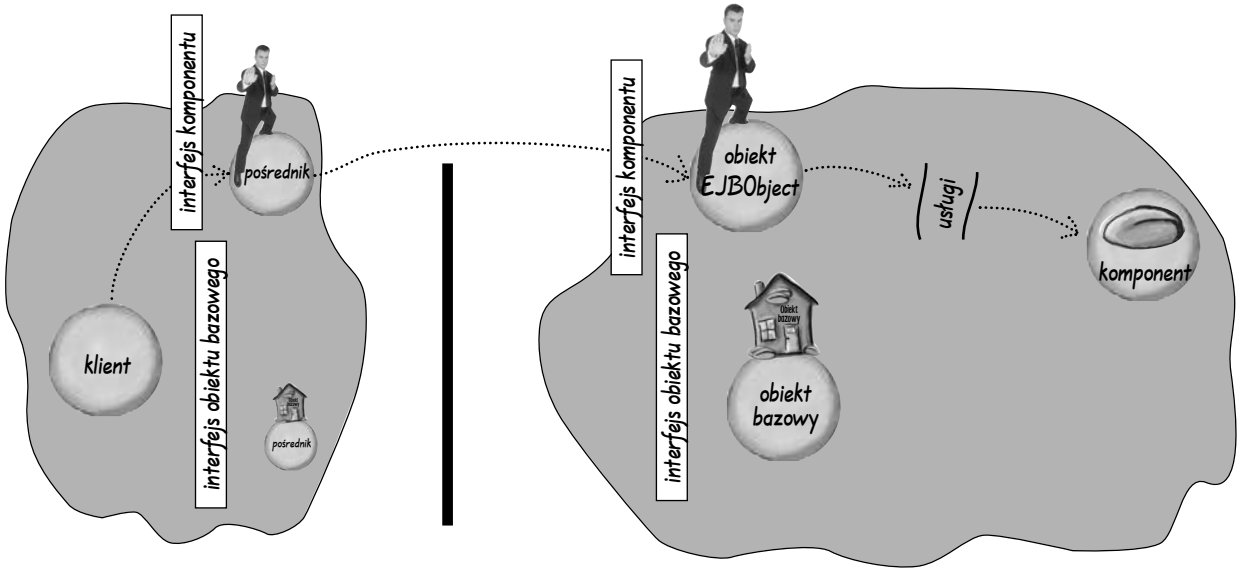
5 **W tym momencie do akcji wkraczają „usługi” i jest tworzony komponent.**



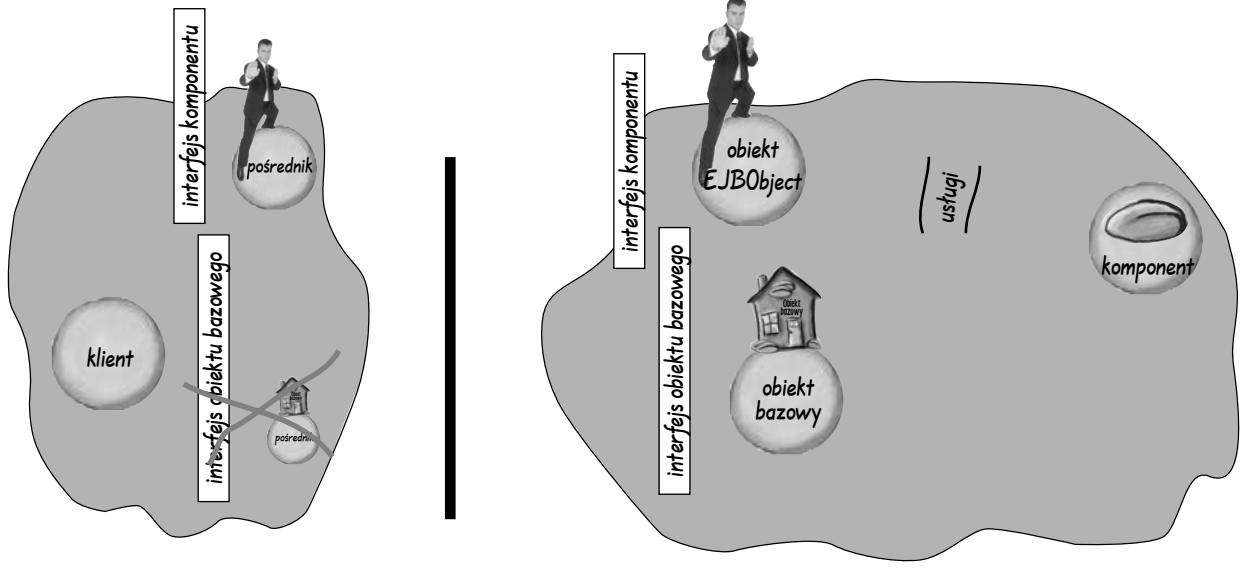
6 **Tworzony jest obiekt** `EJBObject` **(strażnik naszego nowego komponentu), a do klienta zostaje przekazany odpowiedni pośrednik.**



7 **Teraz (nareszcie!) klient może zrobić to, czego tak NAPRAWDĘ pragnął od samego początku — wywołać metodę biznesową komponentu! (Oczywiście wywołanie to musi zostać przekazane przez interfejs komponentu.)**



8 **Jeśli klient nie chce używać większej liczby komponentów tego typu (w naszym przypadku — komponentów `DoradcaBean`), to może się już pozbyć pośrednika obiektu bazowego. Pomimo to klient wciąż będzie mógł wywoływać metody interfejsu komponentu.**



Przygotuj ołówek

Cykl istnienia komponentów EJB:

Klient dysponuje jedynie pośrednikiem obiektu bazowego, a chce wywołać metodę biznesową komponentu.

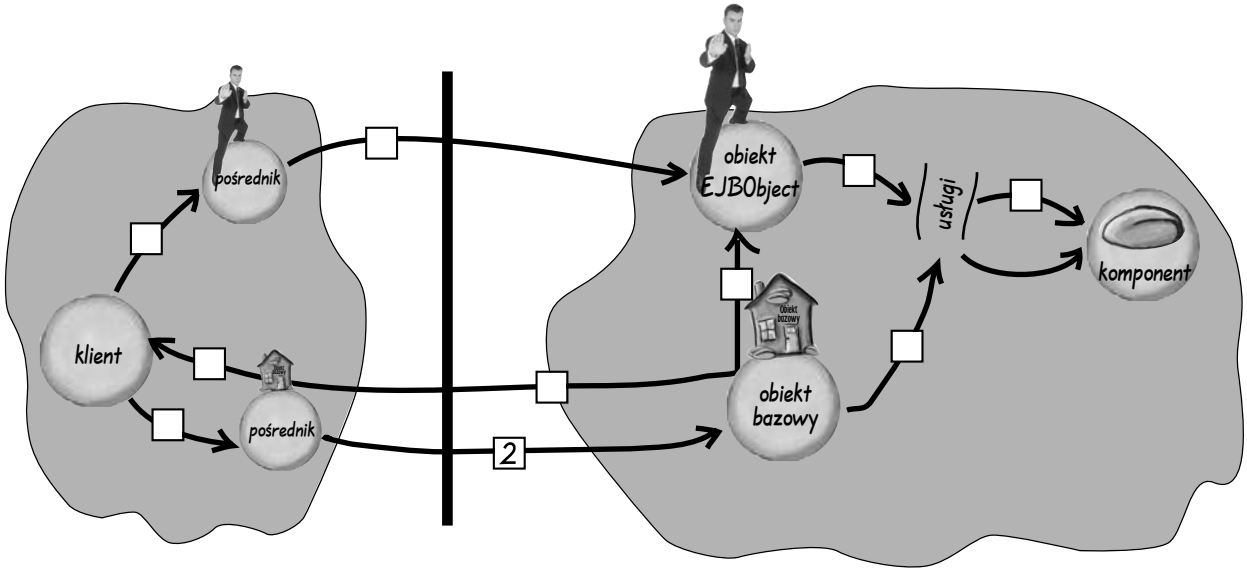
W poniższym scenariuszu przyjmij, że klient już wcześniej wykonał przeszukiwanie rejestru JNDI i zdobył pośrednika zdalnego obiektu bazowego.

Wszystkie operacje przedstawione na poniższym rysunku są wykonywane już PO tym, jak klient zdobył pośrednika obiektu bazowego i gdy stara się pobrać referencję do obiektu `EJBObject`. Cała sekwencja zdarzeń kończy się wywołaniem metody biznesowej komponentu.

Twoim zadaniem jest ponumerowanie strzałek (liczby wpisz w puste kwadraciki) w kolejności odpowiadającej zachodzącym zdarzeniom. Strzałki nie muszą reprezentować tylko i wyłącznie wywołań metod (choć, oczywiście, mogą). Wyobraź sobie, że wskazują one ZDARZENIA. Wymyśl historyjkę opisującą zdarzenie reprezentowane przez poszczególne strzałki. Może być więcej niż jedna poprawna odpowiedź, wszystko zależy od tego jak opowiesz swoją historyjkę. Na rysunku brakuje niektórych strzałek, możesz je dorysować lub po prostu założyć, że zachodzą pewne zdarzenia, które nie są reprezentowane przez strzałki.

Odpręż się i nie śpiesz…

Jeśli nie będziesz wiedział jak odpowiedzieć, przeanalizuj diagramy przedstawione we wcześniejszej części rozdziału.



1.

2. *Pośrednik przekazuje obiektowi bazowemu, że klient chciałby „stworzyć" komponent.*

3.

4.

5.

6.

7.

8.

9.

10.

11.

Zapamiętaj

Kwiatki są na łące, a chmurki na niebie,
interfejsy zdalne są pisane przez Ciebie.

To mały kontener, a to duży projekt,
A serwer sam stworzy obiekty `EJBObject`.

Znowu przyszedł weekend i pogoda marna,
a klasa komponentu nigdy nie jest zdalna.

No wiemy, wiemy... To nie jest nasze najlepsze dzieło. Ale spróbuj *sam*... Pamiętaj, że takie wierszyki zapamiętasz lepiej, jeśli je sam wymyślisz.

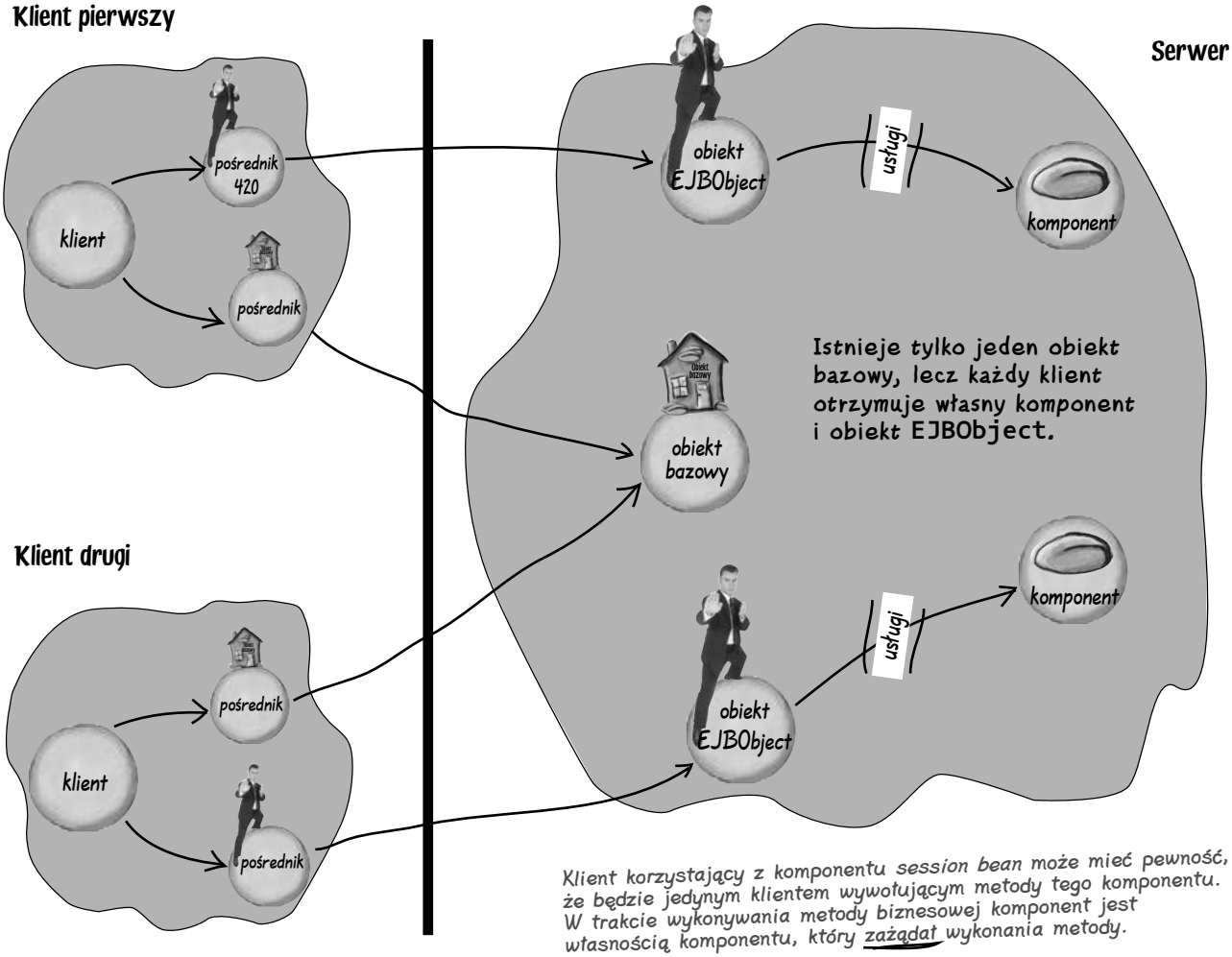
KLUCZOWE ZAGADNIENIA

- Komponenty, które są dostępne dla zdalnych klientów wykorzystują dwa zdalne interfejsy dziedziczące po odpowiednio interfejsach `EJBHome` oraz `EJBObject`.
- Interfejs „zdalny" musi dziedziczyć (jawnie bądź niejawnie) po interfejsie `java.rmi.Remote`, a wszystkie jego metody muszą deklarować wyjątek `java.rmi.RemoteException`.
- W technologii EJB interfejs dziedziczący po interfejsie `EJBObject` jest nazywany zdalnym interfejsem komponentu. To właśnie w nim są deklarowane metody biznesowe.
- Klient nigdy nie wywołuje metod samego komponentu, gdyż komponent NIE jest obiektem zdalnym.
- Kontener implementuje zdalny interfejs komponentu, tworząc klasę, która go implementuje. Klasa ta jest następnie używana w celu stworzenia obiektu typu `EJBObject` dla danego komponentu. (Przypominamy, że obiekt ten jest strażnikiem komponentu.)
- Kontener tworzy także pośrednika do obiektu `EJBObject`.
- Aby stworzyć zdalny interfejs komponentu, należy napisać interfejs dziedziczący po interfejsie `javax.ejb.EJBObject` (a on z kolei dziedziczy po interfejsie `java.rmi.Remote`).
- Ty tworzysz także klasę komponentu, w której są zaimplementowane faktyczne metody biznesowe (pomijając fakt, że z technicznego punktu widzenia w klasie tej wcale nie jest implementowany interfejs zdalnego komponentu).
- Obiekt bazowy jest fabryką komponentów. Jego podstawowym zadaniem jest przekazywanie klientom referencji do komponentów. Pamiętaj jednak, że klient tak naprawdę nigdy nie dysponuje referencją do *komponentu*, w najlepszym przypadku może to być referencja do obiektu typu `EJBObject` wygenerowanego przez serwer dla danego komponentu.
- Interfejs obiektu bazowego tworzony jest poprzez napisanie interfejsu dziedziczącego po interfejsie `javax.ejb.EJBHome` (który z kolei dziedziczy po `java.rmi.Remote`).
- Kontener odpowiada za utworzenie klasy, która zaimplementuje ten interfejs, jak również za wygenerowanie odpowiedniego pośrednika.
- Dla każdego wdrożonego komponentu istnieje tylko jeden obiekt bazowy. Na przykład, niezależnie od liczby istniejących komponentów `Koszyk`, na serwerze będzie istnieć tylko jeden obiekt bazowy zarządzający komponentami tego typu.

Przegląd architektury — komponenty session bean

Klienty wspólnie używają obiektu bazowego, ale nigdy nie współdzielą komponentów.

Każdy klient otrzymuje swoją własną referencję do obiektu typu `EJBObject` oraz swój własny komponent. Klient nigdy nie współdzieli komponentu z jakimkolwiek innym klientem, choć w tym przypadku znaczenie słowa „współdzielić" zależy do tego czy komponent jest stanowy, czy bezstanowy. (Przekonamy się o tym w następnym rozdziale.) Jednak dla wszystkich komponentów danego typu (na przykład, typu `DoradcaBean`) istnieje tylko jeden obiekt bazowy; a zatem oba klienty będą dysponować pośrednikami do tego samego obiektu bazowego. Oba klienty proszą obiekt bazowy o zwrócenie referencji do komponentu `DoradcaBean`. (Oczywiście, klient nigdy nie otrzymuje referencji do *egzemplarza komponentu*, a jedynie referencję do odpowiedniego obiektu `EJBObject`. A ponieważ obiekt `EJBObject` jest obiektem zdalnym — dziedziczy po interfejsie `Remote` — zatem klient otrzymuje pośrednika do tego obiektu.)

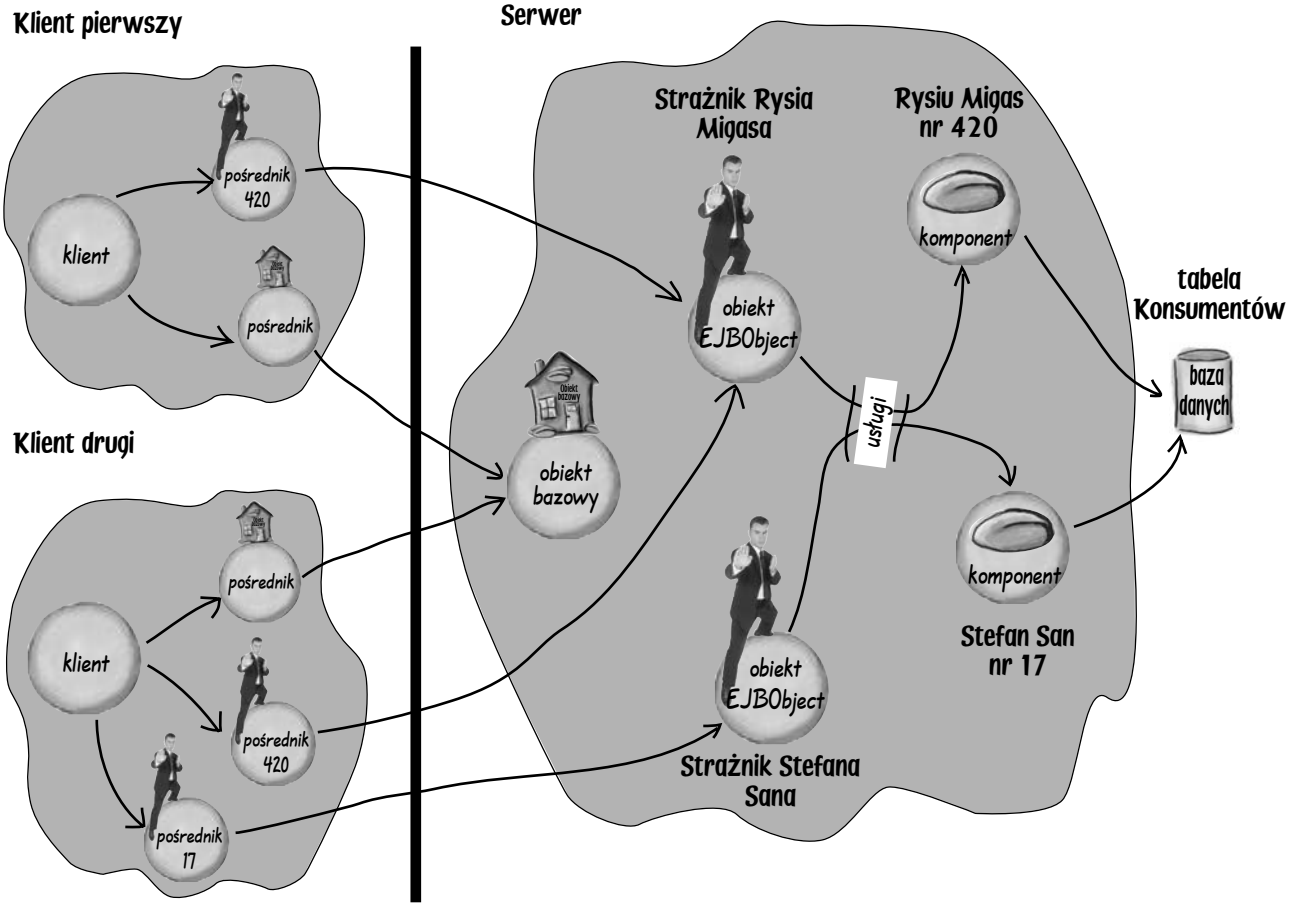


Klient korzystający z komponentu *session bean* może mieć pewność, że będzie jedynym klientem wywołującym metody tego komponentu. W trakcie wykonywania metody biznesowej komponent jest własnością komponentu, który zażądał wykonania metody.

Przegląd architektury — komponenty entity bean

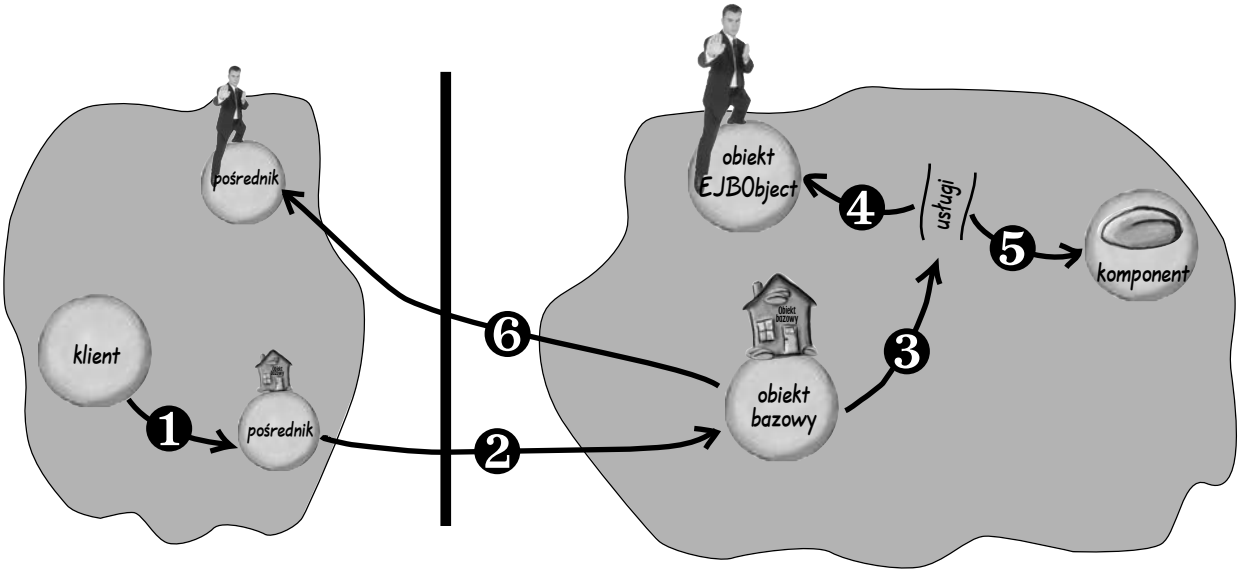
Klienty wspólnie używają obiektu bazowego i mogą także współdzielić same komponenty.

Każdy klient posiada własną referencję do jedynego istniejącego obiektu bazowego zarządzającego komponentami danego typu (na przykład, komponentami `KonsumentBean`). Jednak, w przypadku gdy dwa klienty próbują uzyskać dostęp do tego samego komponentu `Konsument` (Rysiu Migas, nr 420), to oba dostaną referencje do tego samego obiektu `EJBObject`. Obiektu `EJBObject` nr 420. Innymi słowy, obiekt `EJBObject` jest strażnikiem strzegącym konkretnego komponentu `Konsument` (na przykład Rysia Migasa). Jeśli wszystkie klienty będą próbować uzyskać dostęp do komponentu Rysia Migasa nr 420, to każdy z nich będzie dysponować własnym pośrednikiem (to jest oczywiste), lecz jednocześnie wszystkie te pośredniki będą się komunikować z jednym obiektem `EJBObject`. Poza tym będzie istnieć tylko jeden komponent zawierający dane Rysia Migasa nr 420. W przypadku gdy klient spróbuje uzyskać dostęp do dwóch różnych komponentów, otrzyma on dwa obiekty pośredników, komunikujące się z dwoma obiektami `EJBObject`, reprezentującymi dwóch różnych `Konsumentów`. A to oznacza także dwa różne komponenty.



Przegląd architektury — tworzenie stanowego komponentu session bean

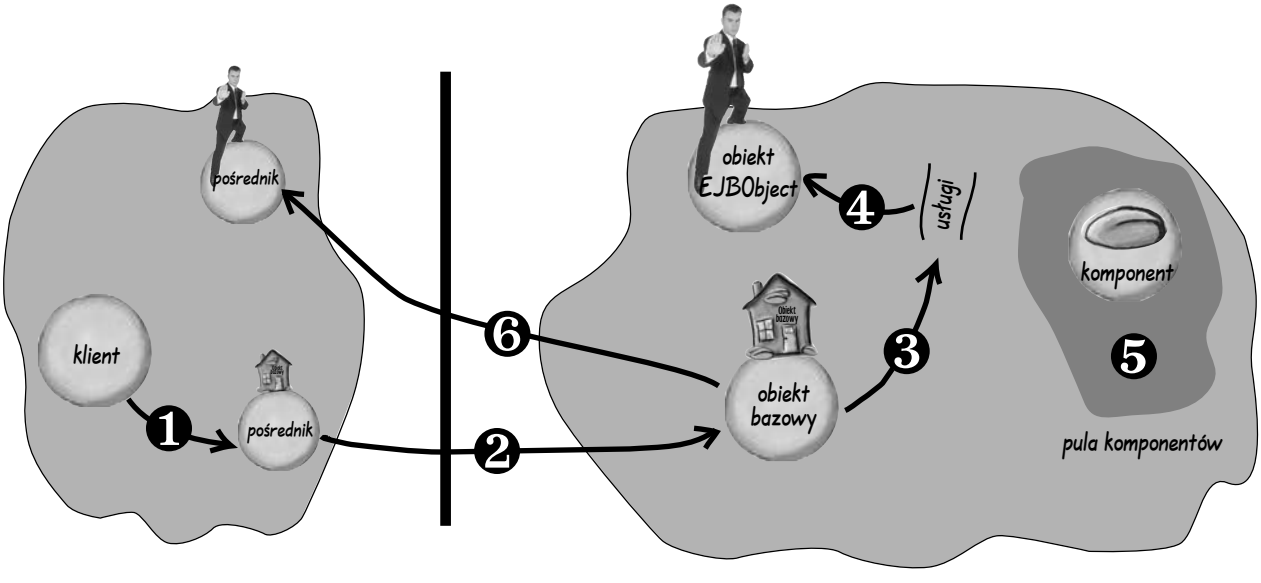
Po uzyskaniu pośrednika obiektu bazowego klient wywołuje jego metodę create(). W odpowiedzi obiekt bazowy tworzy komponent oraz skojarzony z nim obiekt EJBObject i przekazuje klientowi pośrednika obiektu EJBObject.



1. *Klient wywołuje metodę* `create()` *pośrednika obiektu bazowego (*`create()` *jest metodą zdefiniowaną w interfejsie obiektu bazowego).*
2. *Pośrednik przesyła wywołanie metody* `create()` *do zdalnego obiektu bazowego.*
3. *Obiekt bazowy wkracza do akcji i korzysta z dostępnych dla niego usług.*
4. *Zostaje utworzony i zainicjalizowany obiekt* `EBJObject` *dla komponentu danego typu.*
5. *Inicjalizowany jest sam komponent.*
6. *Do klienta przesyłany jest pośrednik obiektu* `EJBObject`*, za pomocą którego klient może wywoływać metody biznesowe zdefiniowane w interfejsie komponentu.*

Przegląd architektury — tworzenie bezstanowego komponentu session bean

Po uzyskaniu pośrednika obiektu bazowego klient wywołuje jego metodę `create()`. W rezultacie obiekt bazowy zwraca klientowi pośrednika już istniejącego obiektu `EJBObject`, lecz jednocześnie nie kojarzy tego obiektu z komponentem! Komponent pozostaje dostępny w puli, aż do chwili gdy klient użyje pośrednika obiektu `EJBObject` do wywołania metody biznesowej.



1. *Klient wywołuje metodę* `create()` *pośrednika obiektu bazowego (metoda* `create()` *jest zdefiniowana w interfejsie bazowym).*

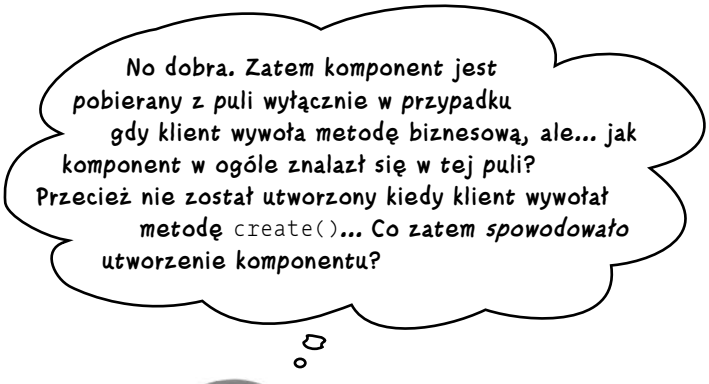
2. *Pośrednik przysyła wywołanie metody* `create()` *do zdalnego obiektu bazowego.*

3. *Obiekt bazowy wkracza do akcji i korzysta z dostępnych dla niego usług.*

4. *Tworzony jest obiekt* `EJBObject`*, który będzie obsługiwać żądania zgłaszane przez danego klienta.*

5. *Komponent cały czas pozostaje w puli! Jest z niej pobierany wyłącznie w celu obsługi wywołania metody biznesowej, jeśli oczywiście klient wywoła tę metodę, używając do tego pośrednika obiektu* `EJBObject`*.*

6. *Do klienta przesyłany jest pośrednik obiektu* `EJBObject`*, za pomocą którego klient może wywoływać metody biznesowe zdefiniowane w interfejsie komponentu.*



Kto i kiedy tworzy bezstanowe komponenty session bean

W pierwszej kolejności musimy zdefiniować, co w tym przypadku oznacza słowo „tworzy". W przypadku komponentów *session bean* oznacza to fizyczne stworzenie i zainicjalizowanie egzemplarza komponentu. Jednak musisz widzieć, że w odniesieniu do komponentów *entity bean* to samo słowo oznacza zupełnie co innego, dlatego też **poniższe rozważania dotyczą tylko i wyłącznie komponentów *session bean***. W dalszej części książki wyjaśnimy co oznacza tworzenie komponentów *entity bean*.

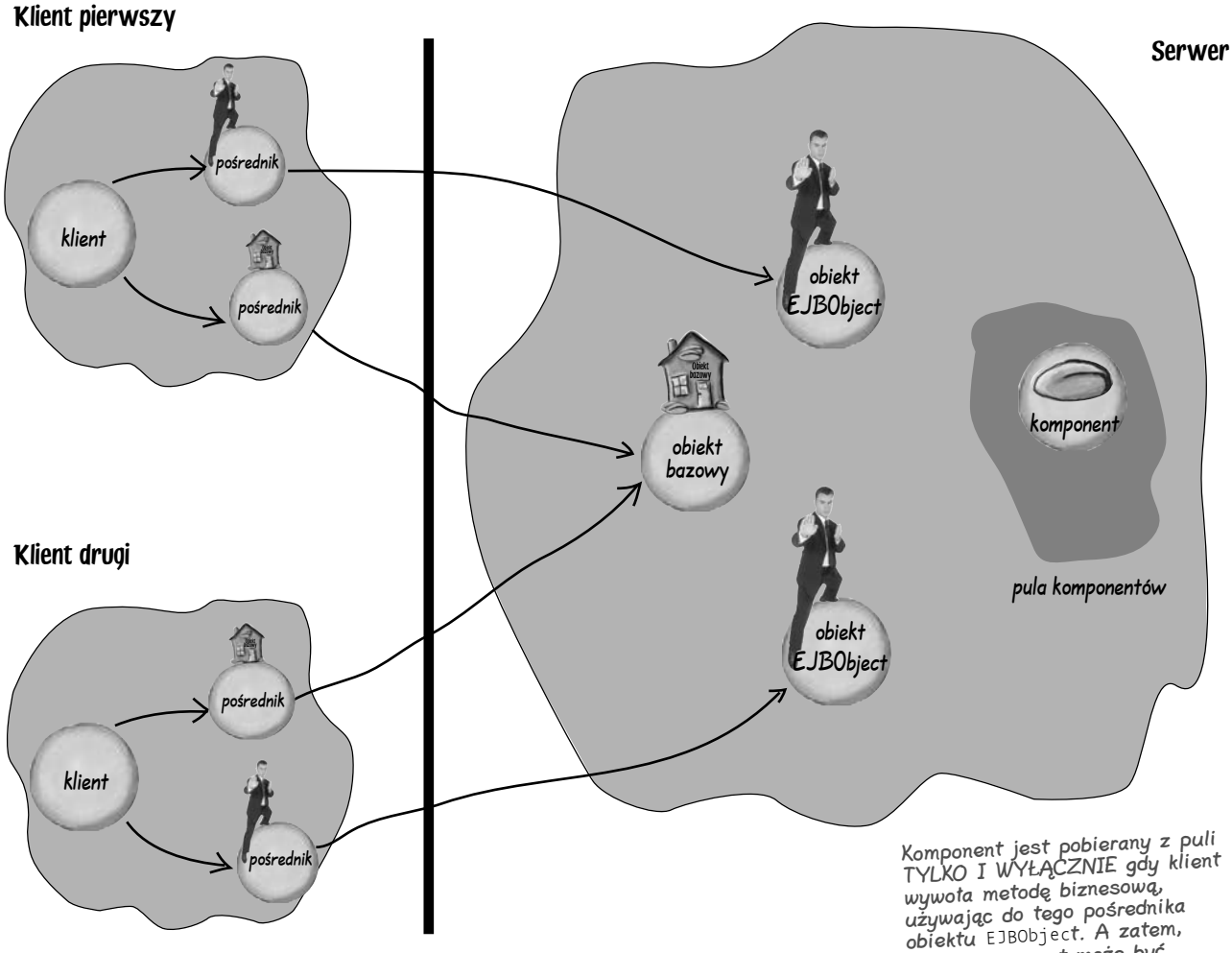
W przypadku *stanowych* komponentów *session bean* proces ich tworzenia jest rozpoczynany przez klienta. To klient wywołuje metodę `create()` pośrednika obiektu bazowego i właśnie to wywołanie rozpoczyna wszystkie kolejne operacje — utworzenie i zainicjalizowanie obiektu `EJBObject` dla przyszłego komponentu, a następnie utworzenie samego komponentu i skojarzenie go z jego strażnikiem — obiektem `EJBObject`.

Jednak w razie korzystania z *bezstanowych* komponentów *session bean* obie operacje — wywołanie przez *klienta* metody `create()` oraz samo stworzenie *komponentu* — nie są ze sobą powiązane. Innymi słowy, fakt wywołania przez klienta metody `create()` pośrednika obiektu bazowego wcale nie oznacza, że od razu zostanie utworzony sam komponent.

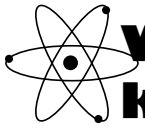
Bezstanowe komponenty *session bean* nie są tworzone aż do momentu, gdy kontener uzna, że taki komponent jest mu potrzebny; a to oczywiście zależy wyłącznie od kontenera. Na przykład, kontener może utworzyć kilka komponentów i umieścić je w puli jeszcze zanim jakikolwiek klient zdąży poprosić o jeden z nich (wywołując metodę `create()` pośrednika obiektu bazowego). Równie dobrze kontener możne tworzyć komponenty „na żądanie" — czyli nie zawracać sobie głowy ich tworzeniem aż do momentu gdy klient faktycznie wywoła metodę biznesową.

Bezstanowe komponenty session bean są bardziej skalowalne

Klienty nie współdzielą obiektów `EJBObject`, jednak ten sam komponent może obsługiwać wywołania metod biznesowych zgłaszane przez *wiele* obiektów `EJBObject`. Jeden komponent może zatem obsługiwać wiele klientów, o ile w danej chwili tylko jeden klient wywołuje metodę biznesową.



Komponent jest pobierany z puli TYLKO I WYŁĄCZNIE gdy klient wywoła metodę biznesową, używając do tego pośrednika obiektu `EJBObject`. A zatem, jeden komponent może być pobrany z puli, by obsłużyć wywołanie metody zgłoszone przez jednego klienta, następnie może wrócić do puli i ponownie zostać z niej pobrany, by obsłużyć wywołanie zgłoszone przez zupełnie innego klienta... i tak dalej...



Wysil szare komórki

Dlaczego w przypadku stosowania bezstanowych komponentów *session bean* architektura bazująca na wykorzystaniu pul komponentów zapewnia lepszą skalowalność, natomiast nie zapewnia jej w razie stosowania komponentów stanowych? Dlaczego stanowe komponenty *session bean* nie mogą być przechowywane w pulach?

Nie ma niemądrych pytań

P. **Jest coś, co mnie NAPRAWDĘ denerwuje... dlaczego mamy interfejs komponentu i interfejs obiektu bazowego, skoro zdalne obiekty są nazywane obiektami bazowymi i obiektami `EJBObject`? Dlaczego nie ma po prostu interfejsu `EJBObject` i interfejsu `HomeObject`? Albo, jeszcze prościej, interfejsów `Home` i `EJB`? Skąd te niespójności?**

O. Cóż... to jest coś, co także nas wyprowadza z równowagi. Ale przyzwyczaisz się do tego. I tak powinieneś być zadowolony, że nie uczyłeś się wersji technologii EJB wcześniejszych niż 2.0, w których interfejsy te były nazywane po prostu `Home` i `Remote`. To dopiero był problem, gdyż w rzeczywistości, w rozumieniu technologii RMI, oba były interfejsami typu `Remote` (czyli dziedziczyły po interfejsie `java.rmi.Remote`). Po drugie, w EJB 2.0 interfejsy te wcale nie muszą być... interfejsami typu `Remote`. Właśnie dlatego trzeba było zrezygnować ze stosowania nazwy `Remote` i zmienić ją na „interfejs komponentu". Wszystko będzie dobrze, jeśli uda Ci się zapamiętać, że interfejs komponentu to ten, w którym są definiowane metody biznesowe, a interfejs obiektu bazowego to ten, w którym są definiowane metody... hm... *bazowe*. Po prostu zapamiętaj proste równanie: komponent = biznes = `EJBObject` (lub `EJBLocalObject`, ale tym zajmiemy się w następnym rozdziale).

Przygotuj ołówek

P F Stanowe komponenty *session bean* mogą być współużytkowane przez wiele klientów.

P F Komponenty Entity Bean mogą być współużytkowane przez wiele klientów, o ile każdy z tych klientów operuje na tej samej encji.

P F Bezstanowe komponenty *session bean* są tworzone w chwili gdy klient wywoła metodę `create()` obiektu bazowego.

P F Stanowe komponenty *session bean* są tworzone gdy klient wywoła metodę `create()` pośrednika obiektu bazowego.

P F Jeśli każdy z klientów posiada referencję do obiektu `EJBObject`, to dla każdego z tych klientów musi istnieć unikatowy bezstanowy komponent *session bean*.

P F Jeśli w danej chwili każdy z klientów jest w trakcie wywołania metody biznesowej, to dla każdego z tych klientów musi istnieć unikatowy bezstanowy komponent *session bean*.

P F Każdy komponent *entity bean* musi posiadać swój własny obiekt `EJBObject`.

Odpowiedzi: F P F P F P P

Nie ma niemądrych pytań

P. Jak działają te pule? Czy jest jedna pula dla wszystkich komponentów, czy jedna dla wszystkich komponentów konkretnego typu?

O. W praktyce nie wiemy jakie rozwiązanie zastosowano w kontenerze, jednak można sobie wyobrazić, że istnieje jedna pula dla każdego typu komponentu. A zatem, gdybyś wdrożył dwa bezstanowe komponenty *session bean*, na przykład: `PoradaBean` oraz `PrognozaPogodyBean`, to dla każdego z nich zostałaby utworzona odrębna pula.

P. Czy każdy bezstanowy komponent session bean ma swój własny obiekt `EJBObject`?

O. Poniekąd. Bezstanowe komponenty *session bean* nie potrzebują strażników, aż do momentu gdy zaczną realizować wywołanie metody biznesowej. A zatem, klient otrzymuje referencję do obiektu `EJBObject`, jednak obiekt ten nie jest kojarzony z komponentem aż do chwili gdy klient wywoła metodę biznesową. W tym momencie komponent jest pobierany z puli, by obsłużyć wywołanie. Jak widać, obiekt `EJBObject`, którym dysponuje klient może współpracować z komponentami konkretnego *typu* (na przykład: `DoradcaBean` lub `PrognozaPogodyBean`), a nie z konkretnym *egzemplarzem* komponentu.

P. Dlaczego stanowe komponenty session bean nie są przechowywane w pulach?

O. Czy zastanawiałeś się nad tym zagadnieniem w ćwiczeniu „Wysil szare komórki" na poprzedniej stronie? Jeśli się nie zastanawiałeś, to nie czytaj dalszej części tej odpowiedzi aż do chwili gdy wymyślisz jakieś własne propozycje. Jeśli czytasz dalszą część tego akapitu, oznacza to, że jednak przemyślałeś zagadnienie, znasz odpowiedź na postawione pytanie, a my jedynie ją potwierdzimy...

Pamiętaj, że stanowe komponenty *session bean* przechowują stan interakcji z klientem. Oznacza to, że komponent musi przechowywać stan klienta (innymi słowy, musi pamiętać pewne informacje dotyczące klienta) pomiędzy kolejnymi wywołaniami metod realizowanymi przez tego samego klienta.

Wyobraź sobie ponownie koszyk na zakupy — komponent stanowy obsługujący taki koszyk musi pamiętać co jest w koszyku za każdym razem gdy klient wywoła metodę `dodajTowarDoKoszyka()`. Z drugiej strony, komponent bezstanowy nie musi pamiętać żadnych informacji związanych z klientem; dlatego też, z punktu widzenia klienta, poszczególne komponenty bezstanowe (konkretnego typu) absolutnie niczym się między sobą nie różnią i klient może skorzystać z każdego z nich.

Jeśli komponent `DoradcaBean` zwraca poradę w żaden sposób nie związaną z poradami, które komponent zwrócił wcześniej (jak również, która nie zależy od niczego, o czym *komponent* dowiedział się od klienta), to nie ma żadnego powodu, by był on komponentem stanowym. W tym przypadku, za każdym razem gdy klient wywoła metodę `getPorada()` zdefiniowaną w interfejsie komponentu (używając w tym celu obiektu `EJBObject`), będzie ją w stanie wykonać dowolny komponent `DoradcaBean`.

Z drugiej strony, gdyby komponent `DoradcaBean` został zmodyfikowany w taki sposób, by zwracał losową, lecz nie powtarzającą się poradę, to `DoradcaBean` musiałby być komponentem stanowym, gdyż tylko w ten sposób mógłby przechowywać informacje o poradach udzielonych wcześniej i nie powtarzać się.

P. Jak długo stanowy komponent session bean przechowuje informacje o stanie związane z konkretnym klientem?

O. Jedynie przez okres trwania sesji. Sesja trawa do momentu gdy klient poinformuje komponent, iż zakończył używać komponentu (co może zrobić, wywołując metodę `remove()` dostępną w interfejsie komponentu) lub do momentu wystąpienia awarii serwera bądź do upłynięcia limitu czasu istnienia komponentu (tym zagadnieniem zajmiemy się w rozdziale poświęconym komponentom *session bean*).

P. A zatem, stanowe komponenty session bean nie są skalowalne?

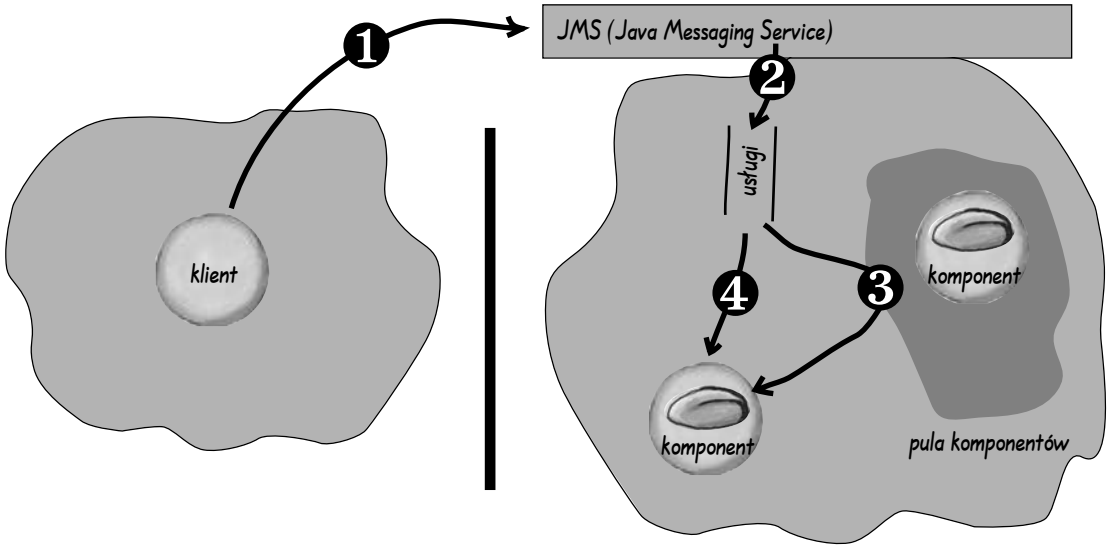
O. Nie, takie komponenty są skalowalne. Jednak nie zapewniają równie dużej skalowalności, co komponenty bezstanowe.

P. Ale jak to możliwe, że stanowe komponenty session bean są skalowalne, skoro zawsze dla jednego klienta musi istnieć jeden taki komponent?

O. To prawda, że dla każdego klienta musi istnieć jeden komponent, lecz nie każdy komponent musi aktywnie korzystać z zasobów. Jeśli pomiędzy kolejnymi wywołaniami metod biznesowych stanowego komponentu *session bean* zgłaszanymi przez tego samego klienta występują długie odstępy czasu, to komponent może być chwilowo wprowadzony w stan dezaktywacji (ang. *passivation*). Ten stan pozwala na zachowanie informacji o kliencie (co jest oczywiste) przy jednoczesnym zmniejszeniu liczby komponentów istniejących na serwerze. Komponent wyjdzie ze stanu dezaktywacji i ponownie przystąpi do pełnienia swych obowiązków (czyli będzie aktywny), kiedy klient zgłosi wywołanie metody biznesowej.

Przegląd architektury — komponenty message-driven bean

Komponenty *message-driven bean* nie mają tak zwanego „widoku klienta". Oznacza to, że nie posiadają one interfejsów (ani zdalnych, ani lokalnych), które poinformowałyby klientów jakie metody udostępnia taki komponent. Innymi słowy, komponenty tego typu nie posiadają ani obiektu bazowego, ani obiektu `EJBObject`. Nie mają ani interfejsu obiektu bazowego, ani interfejsu komponentu.

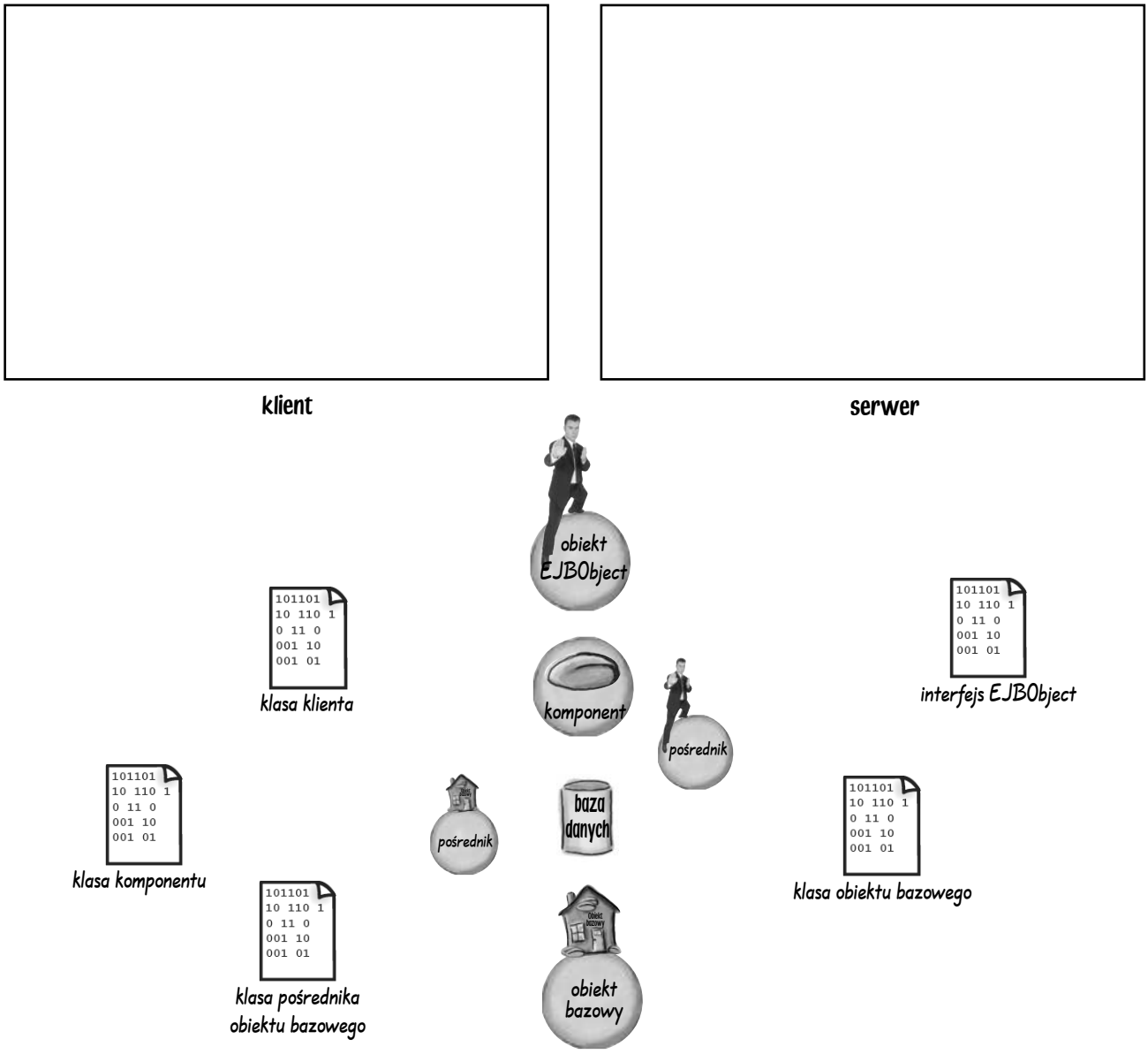


1. *Klient przesyła komunikat do usługi JMS.*
2. *Usługa JMS dostarcza komunikat kontenerowi.*
3. *Kontener pobiera z puli komponent message-driven bean.*
4. *Kontener przekazuje komunikat do komponentu (wywołując jego metodę* `onMessage()` *należącą do interfejsu* `MessageListener`*).*

Rozwiązania ćwiczeń

Gdzie umieścić poszczególne elementy?

Umieść poszczególne obiekty i klasy w odpowiednich miejscach — po stronie klienta, na serwerze lub w obu tych miejscach jednocześnie (tak, tak, możesz to zrobić). Notatka: nie wszystkie elementy układanki zostały narysowane na tej stronie, jeśli zatem skończysz i przypomnisz sobie o czymś (klasie lub obiekcie), co mogłoby się znaleźć na schemacie, to narysuj to!



Ćwiczenia

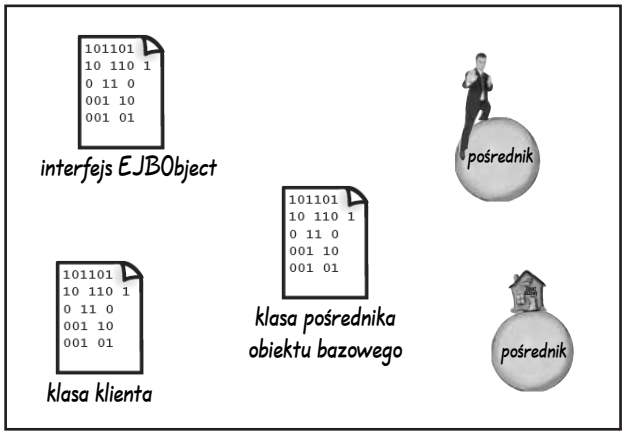
Zorganizuj swoje komponenty

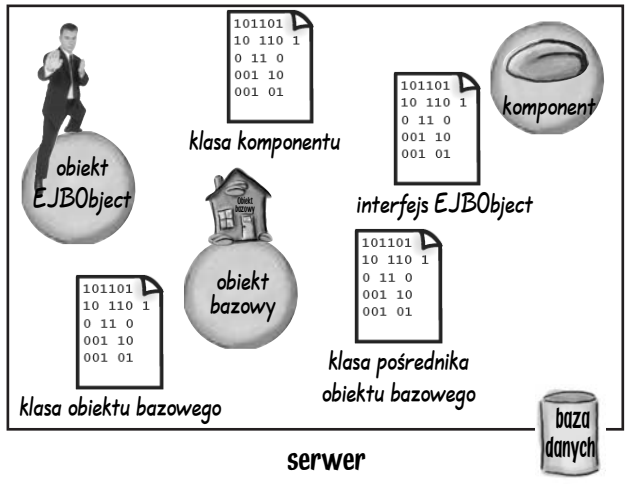
Uzupełnij tabelę, umieszczając znaczki (a jeszcze lepiej — wyjaśnienia) w komórkach, dla których stwierdzenie podane w danym wierszu jest prawdziwe dla danego typu komponentu. Jedną z komórek wypełniliśmy za Ciebie. Jeśli w którymś momencie nie będziesz wiedział jak uzupełnić tabelę, to przejrzyj poprzednie dwa rozdziały. W kilku miejscach, być może, będziesz musiał zgadywać. Nic nie szkodzi — nim zakończysz lekturę niniejszej książki i tak wszystko się wyjaśni. Wiemy, że sobie poradzisz. (Przypomnij sobie piosenkę z filmu „Rocky".)

	Bezstanowe komponenty *session bean*	**Stanowe komponenty *session bean***	**Komponenty *entity bean***	**Komponenty *message-driven bean***
Wraz z nimi są używane pule.	*Tak. Ponieważ nie przechowują żadnych informacji skojarzonych z konkretnym klientem, nie trzeba mieć jednego takiego komponentu dla każdego klienta.*			
Wiele klientów może mieć referencje do tego samego komponentu.				
Mamy gwarancję, że przetrwają awarię serwera.				
Mają widok klienta.				
Pozwalają na komunikację asynchroniczną.				
Reprezentują proces.				
Reprezentują „rzeczy" w trwałym magazynie (na przykład, w bazie danych).				

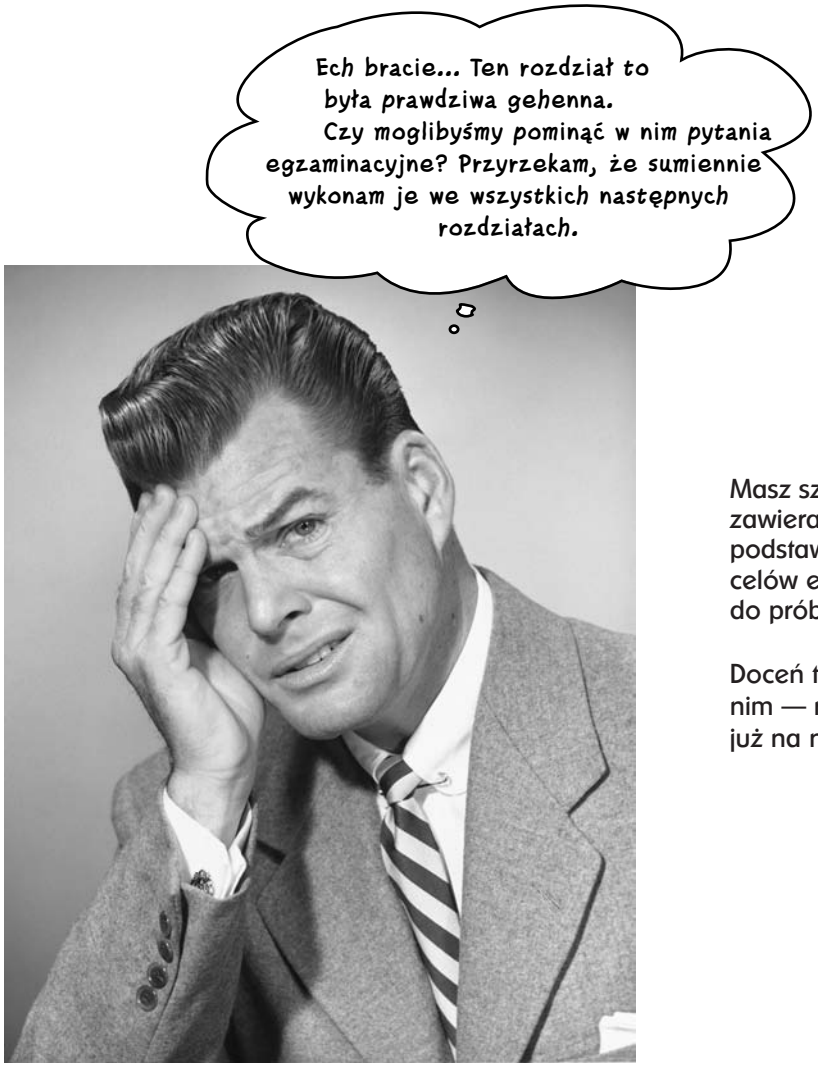
Ćwiczenia

Gdzie umieścić poszczególne elementy?





Notatka: nie znajdziesz rozwiązania ćwiczenia z tabelą komponentów. Chcemy bowiem, żebyś to TY ją uzupełnił. Jest to jeszcze jedna możliwość nauczenia się czegoś, za którą będziesz nas wspominał z wdzięcznością.



Masz szczęście. Ten rozdział zawiera jedynie informacje podstawowe, więc nie ma w nim celów egzaminacyjnych ani pytań do próbnego egzaminu.

Doceń ten moment i rozkoszuj się nim — rozdział trzeci zaczyna się już na następnej stronie…