helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

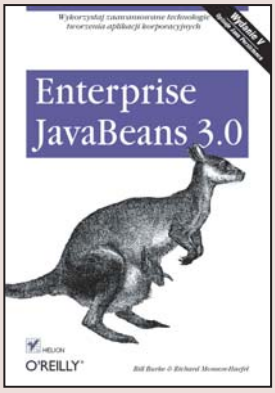
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Enterprise JavaBeans 3.0. Wydanie V



Autorzy: Bill Burke, Richard Monson-Haefel
Tłumaczenie: Mikołaj Szczepaniak, Krzysztof Ostrowski
ISBN: 83-246-0726-9
Tytuł oryginału: Enterprise JavaBeans 3.0 (5th Edition)
Format: B5, stron: 760
Przykłady na ftp: 10150 kB

Wykorzystaj zaawansowane technologie tworzenia aplikacji korporacyjnych

- Poznaj architekturę EJB 3.0
- Stwórz własne komponenty
- Zaprojektuj własne usługi sieciowe na podstawie EJB 3.0

Enterprise JavaBeans to technologia przeznaczona do tworzenia złożonych programów, oparta na języku Java i platformie Java Enterprise Edition. Stosowana jest przy tworzeniu rozbudowanych aplikacji korporacyjnych i pozwala programistom na generowanie mechanizmów automatycznego zarządzania usługami kluczowymi dla systemu. Wersje EJB stosowane do tej pory wymagały od twórców aplikacji implementowania mechanizmów, które nie miały wiele wspólnego z właściwą logiką biznesową tworzonego oprogramowania, co znacznie wydłużało i komplikowało proces produkcji systemu. Najnowsza wersja, oznaczona numerem 3.0, jest pozbawiona tych wad.

Dzięki książce „Enterprise JavaBeans 3.0. Wydanie V" poznasz najnowsze wcielenie technologii EJB. Opisano tu wszystkie rozwiązania, które umożliwiły uproszczenie standardu Enterprise JavaBeans 3.0 względem jego poprzednich wersji. Czytając tę książkę, poznasz nowy interfejs Java Persistence API, który zastąpił stosowane dotychczas komponenty encyjne zwykłymi obiektami Javy, oraz nauczysz się sposobów eliminowania konieczności implementowania interfejsów EnterpriseBean. Dowiesz się, jak stosować adnotacje w miejsce elementów języka XML umieszczanych w deskryptorach wdrożenia. Znajdziesz tu również praktyczne przykłady, dzięki którym błyskawicznie opanujesz nową wersję EJB.

- Architektura EJB 3.0
- Relacje pomiędzy komponentami
- Zapytania i język EJB QL
- Komponenty sesyjne
- Obsługa transakcji
- Implementowanie usług WWW
- Instalacja i konfiguracja serwera JBoss

**Nie trać więcej czasu! Zastosuj technologię,
która ułatwi Ci wytwarzanie systemów korporacyjnych**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 3.

Zarządzanie zasobami i usługi podstawowe

W rozdziale 2. opisano podstawy architektury technologii Enterprise JavaBeans i Java Persistence, w tym relacje łączące klasę komponentu, kontener EJB oraz usługę `EntityManager`. Relacje pomiędzy wymienionymi elementami interesującej nas architektury składają się odpowiednio na wspólny model rozproszonych komponentów serwera oraz na model utrwalania, który może być stosowany zarówno w aplikacjach działających po stronie serwera, jak i w aplikacjach autonomicznych. Same modele w tej formie nie wystarczą do tego, by technologia Enterprise JavaBeans zainteresowała programistów i stała się bardziej funkcjonalna od innych popularnych architektur. Serwery EJB dodatkowo muszą zarządzać zasobami wykorzystywanymi przez komponenty i z reguły oferują możliwość jednoczesnego zarządzania tysiącami lub wręcz milionami obiektów rozproszonych. Właśnie do serwera EJB należy zarządzanie sposobem wykorzystywania przez obiekty rozproszone pamięci, wątków, połączeń z bazą danych, mocy obliczeniowej itp. Co więcej, specyfikacja EJB definiuje interfejsy, które ułatwiają programistom korzystanie z wymienionych mechanizmów.

Serwery EJB oferują pięć podstawowych usług: współbieżność, zarządzanie transakcjami, utrwalanie danych, rozpraszanie obiektów, nazewnictwo oraz bezpieczeństwo. Wymienione usługi stanowią rodzaj infrastruktury niezbędnej do właściwej pracy systemu trójwarstwowego. Specyfikacja Enterprise JavaBeans opisuje też dwie usługi dodatkowe: asynchroniczne przesyłanie komunikatów i licznik czasowy.

W niniejszym rozdziale skoncentrujemy się na elementach funkcjonalności odpowiedzialnych za zarządzanie zasobami oraz na najważniejszych usługach (tzw. usługach podstawowych) oferowanych komponentom Enterprise JavaBeans przez ich serwery.

Zarządzanie zasobami

Wielkie systemy biznesowe charakteryzujące dużą liczbą użytkowników mogą wymagać jednoczesnego istnienia i realizacji właściwych zadań przez tysiące lub wręcz miliony obiektów. Wraz ze wzrostem liczby wspólnych operacji podejmowanych przez te obiekty działania w takich obszarach jak zarządzanie współbieżnością czy przetwarzanie transakcyjne mogą prowadzić do wydłużenia czasu odpowiedzi i — tym samym — frustracji użytkowników. Serwery Enterprise JavaBeans próbują optymalizować pracę systemów EJB, synchronizując wspólne działania rozproszonych obiektów i wymuszając współdzielenie najcenniejszych zasobów.

Jest pewna zależność pomiędzy liczbą klientów a liczbą obiektów rozproszonych, których istnienie jest warunkiem koniecznym sprawnej obsługi tych klientów. Obsługa większej liczby klientów z oczywistych względów wymaga większej liczby obiektów rozproszonych. Istnieje punkt, od którego wzrost liczby klientów powoduje spadek wydajności i przepustowości całego systemu. Specyfikacja Enterprise JavaBeans opisuje dwa mechanizmy, których zadaniem jest ułatwienie zarządzania dużą liczbą komponentów w czasie wykonywania: mechanizm zarządzania pulą egzemplarzy oraz mechanizm aktywacji. Twórcy technologii EJB dodatkowo zdecydowali się użyć do zarządzania połączeniami z zasobami architektury Java EE Connector Architecture (czyli konektory Java EE Connectors). Wraz ze wzrostem liczby obiektów rozproszonych i klientów z natury rzeczy musi rosnąć także liczba połączeń z zasobami. Konektory Java EE Connectors mogą być wykorzystywane przez kontener EJB do zarządzania połączeniami z bazami danych, systemami przesyłania komunikatów, systemami ERP, istniejącymi systemami informatycznymi oraz pozostałymi typami zasobów.

Zarządzanie pulą egzemplarzy

Koncepcja tworzenia puli zasobów nie jest żadną nowością. Przykładowo większość baz danych tworzy i zarządza pulą połączeń, która umożliwia obiektom biznesowym wchodzącym w skład danego systemu współdzielenie dostępu do zasobów bazy danych. W ten sposób można stosunkowo łatwo ograniczyć liczbę potrzebnych połączeń, co z kolei pozwala zmniejszyć poziom wykorzystania zasobów i — tym samym — przekłada się na wyższą przepustowość. Architekturę Java EE Connector Architecture (JCA), która jest często wykorzystywana przez kontenery EJB właśnie do zarządzania pulą połączeń z bazami danych i innymi zasobami, szczegółowo omówimy w dalszej części tego rozdziału. Większość kontenerów EJB stosuje mechanizmy zarządzania pulą zasobów dla komponentów serwera — ta technika często jest określana mianem **zarządzania pulą egzemplarzy** (ang. *instance pooling*). Pula egzemplarzy ogranicza liczbę egzemplarzy komponentów (a więc także zasobów) niezbędnych do obsługi żądań generowanych przez oprogramowanie klienckie.

Jak wiemy, aplikacje klienckie współpracują z komponentami sesyjnymi za pośrednictwem interfejsów zdalnych i lokalnych implementowanych przez odpowiednie obiekty EJB. Oznacza to, że oprogramowanie klienckie nigdy nie ma bezpośredniego dostępu do właściwych egzemplarzy klas komponentów sesyjnych. Podobnie aplikacje klienckie systemu JMS nigdy nie współpracują bezpośrednio z komponentami sterowanymi komunikatami JMS (komponentami JMS-MDB). Komunikaty wysyłane przez te aplikacje są kierowane do systemu kontenera EJB, który odpowiada za ich dostarczanie do odpowiednich egzemplarzy komponentów sterowanych komunikatami.

Zarządzanie pulą egzemplarzy jest możliwe, ponieważ aplikacje klienckie nigdy nie uzyskują bezpośredniego dostępu do komponentów. W związku z tym utrzymywanie odrębnej kopii każdego z komponentów EJB dla każdego klienta nie jest konieczne. Serwer EJB może z powodzeniem realizować te same zadania, utrzymując mniejszą liczbę egzemplarzy komponentów EJB, ponieważ pojedyncze egzemplarze tych komponentów mogą być wielokrotnie wykorzystywane do obsługi różnych żądań. Mimo że opisywane podejście wielu programistom może się wydawać niebezpieczne, dobrze zaprojektowany i zaimplementowany mechanizm zarządzania pulą egzemplarzy może znacznie ograniczyć ilość zasobów potrzebnych do obsługi wszystkich żądań generowanych przez oprogramowanie klienckie.

Cykl życia bezstanowego komponentu sesyjnego

Aby jak najlepiej zrozumieć sposób działania puli egzemplarzy, warto przeanalizować cykl życia bezstanowego komponentu sesyjnego. Ogólnie bezstanowy komponent sesyjny może się znajdować w trzech stanach:

Brak stanu
: W tym stanie znajdują się te egzemplarze komponentów, które nie zostały jeszcze zainicjalizowane. Warto identyfikować ten specyficzny stan, ponieważ dobrze reprezentuje sytuację z początku i końca cyklu życia egzemplarza komponentu.

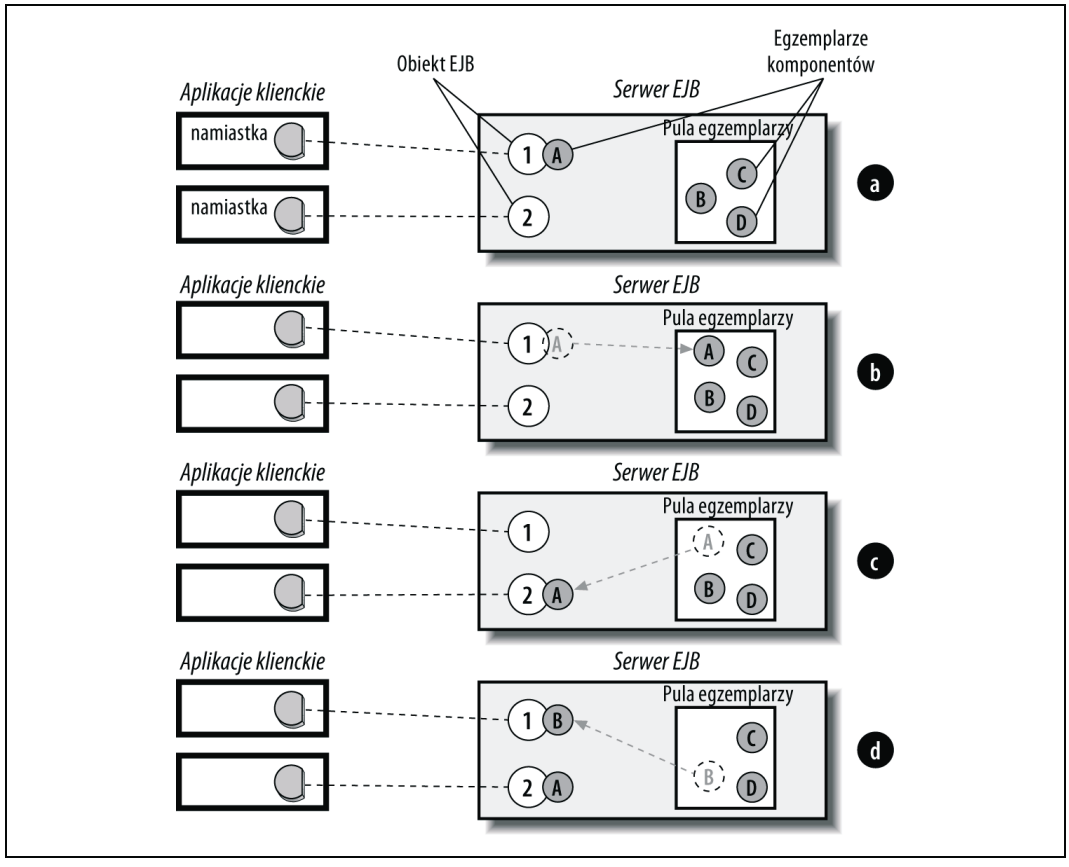
W puli
: Kiedy egzemplarz komponentu znajduje się w tym stanie, wiemy, że został zainicjalizowany przez kontener, ale jeszcze nie związano go z żadnym żądaniem wygenerowanym przez oprogramowanie klienckie.

Stan gotowości
: Egzemplarz komponentu EJB znajdujący się w tym stanie został związany z konkretnym żądaniem EJB i jest gotowy do przetworzenia przychodzących wywołań metod biznesowych.

Ponieważ bezstanowe komponenty sesyjne nie utrzymują swojego stanu pomiędzy kolejnymi wywołaniami metod, każde z tych wywołań jest niezależne od pozostałych — może wykonywać swoje zadania bez konieczności odwoływania się do zmiennych egzemplarzy. Oznacza to, że dowolny egzemplarz bezstanowego komponentu sesyjnego może obsługiwać żądania dowolnego obiektu EJB pod warunkiem, że rodzaj i format tych żądań jest właściwy. Kontener EJB może więc wymieniać egzemplarze komponentu sesyjnego nawet pomiędzy kolejnymi wywołaniami metod.

Każdy producent kontenera EJB implementuje mechanizm zarządzania pulą egzemplarzy w nieco inny sposób, jednak wszystkie strategie tworzenia tego rodzaju pul mają na celu takie zarządzanie kolekcjami egzemplarzy komponentów, które w czasie wykonywania zagwarantuje możliwie szybki dostęp do tych egzemplarzy. Podczas konstruowania puli egzemplarzy kontener EJB tworzy wiele egzemplarzy klasy pojedynczego komponentu i utrzymuje je w wewnętrznej kolekcji do chwili, w której okażą się potrzebne. W odpowiedzi na żądania metod biznesowych generowanych przez aplikacje klienckie kontener EJB przypisuje poszczególnym klientom egzemplarze komponentu wchodzące w skład dostępnej puli. Po zakończeniu przetwarzania żądania, kiedy odpowiedni obiekt EJB nie jest już potrzebny, następuje jego zwrócenie do puli egzemplarzy. Serwer EJB utrzymuje pule egzemplarzy dla każdego wdrożonego typu bezstanowego komponentu sesyjnego. Warto pamiętać, że każdy egzemplarz wchodzący w skład puli egzemplarzy jest elementem **równoprawnym** — wszystkie egzemplarze są traktowane w identyczny sposób. Egzemplarze są wybierane z puli i przydzielane kolejnym żądaniom EJB w sposób całkowicie przypadkowy (żaden z egzemplarzy nie ma pozycji uprzywilejowanej względem pozostałych).

Na rysunku 3.1 przedstawiono schemat wymiany egzemplarzy pomiędzy kolejnymi wywołaniami metod bezstanowego komponentu sesyjnego. Na rysunku 3.1a widać egzemplarz *A* obsługujący wywołanie metody biznesowej przekazane przez 1. obiekt EJB. Po zakończeniu obsługi tego żądania egzemplarz *A* wraca do puli egzemplarzy (patrz rysunek 3.1b). Kiedy do systemu EJB dociera wywołanie metody 2. obiektu EJB, egzemplarz *A* jest wiązany z tym obiektem na czas trwania bieżącej operacji (patrz rysunek 3.1c). W czasie, gdy egzemplarz *A* obsługuje żądanie 2. obiektu EJB, 1. obiekt EJB otrzymuje wygenerowane przez oprogramowanie klienckie wywołanie innej metody — nowe żądanie jest obsługiwane przez egzemplarz *B* (patrz rysunek 3.1d).



Rysunek 3.1. Strategia wymiany egzemplarzy bezstanowych komponentów sesyjnych

Opisana strategia wymiany egzemplarzy bezstanowych komponentów sesyjnych umożliwia efektywną obsługę kilkuset aplikacji klienckich za pomocą zaledwie kilku egzemplarzy bezstanowego komponentu sesyjnego, ponieważ czas potrzebny do wykonania większości wywołań metod z reguły jest znacznie krótszy od przerw dzielących kolejne wywołania. Egzemplarz komponentu, który kończy obsługę żądania wygenerowanego przez obiekt EJB, natychmiast jest dostępny dla dowolnego innego obiektu EJB, który tego potrzebuje. Takie rozwiązanie umożliwia znacznie mniejszej liczbie bezstanowych komponentów sesyjnych obsługę większej liczby żądań, co automatycznie przekłada się na mniejsze wykorzystanie zasobów i wyższą wydajność.

Jeśli dany komponent sesyjny żąda wstrzyknięcia egzemplarza interfejsu `javax.ejb.EJBContext`, zaraz po umieszczeniu jego egzemplarza w odpowiedniej puli następuje przekazanie referencji do właściwego obiektu kontekstu (więcej informacji na ten temat można znaleźć w rozdziale 14.). Interfejs `EJBContext` może być wykorzystywany przez komponenty do komunikowania się z ich środowiskiem EJB. Interfejs `EJBContext` jest szczególnie przydatny w czasie, gdy dany egzemplarz komponentu znajduje się w stanie gotowości.

W czasie obsługi żądania przez egzemplarz komponentu interfejs `EJBContext` nabiera nieco innego znaczenia, ponieważ oferuje informacje o aplikacji klienckiej korzystającej z danego komponentu. Co więcej, interfejs `EJBContext` zapewnia egzemplarzowi komponentu dostęp

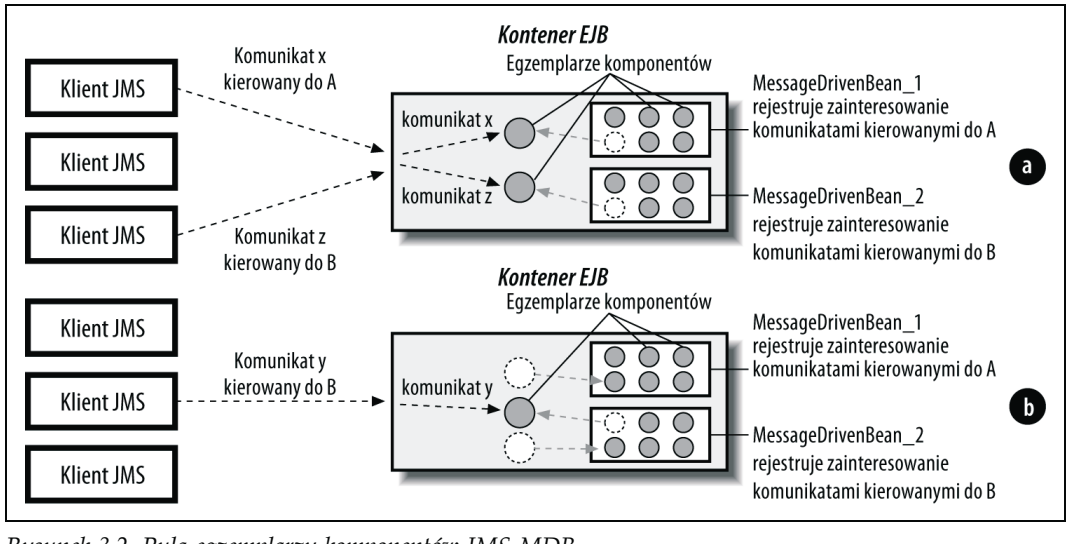
do jego namiastki pośrednika EJB, co nie jest bez znaczenia w sytuacji, gdy dany komponent musi przekazywać referencje do samego siebie i (lub) pozostałych komponentów EJB. Oznacza to, że interfejs `EJBContext` w żadnym razie nie jest strukturą statyczną — jest dynamicznym interfejsem kontenera EJB.

Bezstanowe komponenty sesyjne deklarujemy jako „bezstanowe", stosując adnotację `@javax.ejb.Stateless` w kodzie źródłowym lub odpowiednie zapisy w deskryptorze wdrożenia. Od momentu wdrożenia klasy naszego bezstanowego komponentu sesyjnego kontener zakłada, że pomiędzy kolejnymi wywołaniami metod nie jest utrzymywany stan konwersacji. Bezstanowe komponenty sesyjne mogą zawierać zmienne egzemplarzy, jednak z uwagi na możliwość obsługi wielu różnych obiektów EJB przez pojedynczy egzemplarz takiego komponentu tego rodzaju zmienne nie powinny być wykorzystywane do reprezentowania stanu konwersacji.

Komponenty sterowane komunikatami i pula egzemplarzy

Podobnie jak bezstanowe komponenty sesyjne, komponenty sterowane komunikatami nie utrzymują stanów właściwych dla poszczególnych żądań i jako takie doskonale nadają się do składowania w puli egzemplarzy.

W większości kontenerów EJB dla każdego typu komponentów sterowanych komunikatami jest tworzona osobna pula egzemplarzy odpowiedzialnych za obsługę komunikatów przychodzących. Komponenty JMS-MDB rejestrują swoje zainteresowanie określonymi rodzajami komunikatów (kierowanych w określone miejsce, rodzaj adresu wykorzystywanego podczas wysyłania i odbierania komunikatów). Kiedy asynchroniczny komunikat jest wysyłany przez klienta systemu JMS, komunikat trafia do kontenera EJB zawierającego komponenty, które zarejestrowały się jako odbiorcy tego rodzaju komunikatów. Kontener EJB odpowiada za określenie, który komponent JMS-MDB powinien otrzymać nowy komunikat, po czym wybiera z puli jeden egzemplarz tego komponentu, któremu zostanie zlecona obsługa danego komunikatu. Kiedy wybrany egzemplarz komponentu JMS-MDB zakończy przetwarzanie tego komunikatu (gdy wywołana metoda `onMessage()` zwróci sterowanie), kontener EJB zwróci ten egzemplarz do odpowiedniej puli. Sposób przetwarzania żądań aplikacji klienckich przez kontener EJB przedstawiono na rysunku 3.2.



Rysunek 3.2. Pula egzemplarzy komponentów JMS-MDB

Na rysunku 3.2a przedstawiono sytuację, w której pierwszy klient JMS dostarcza komunikat kierowany pod adres *A*, natomiast ostatni (trzeci) klient JMS dostarcza komunikat kierowany pod adres *B*. Kontener EJB wybiera egzemplarz komponentu `MessageDrivenBean_1`, który ma przetworzyć komunikat kierowany pod adres *A*, oraz egzemplarz komponentu `MessageDrivenBean_2`, który ma przetworzyć komunikat kierowany pod adres *B*. Egzemplarze obu wymienionych komponentów są na czas przetwarzania odpowiednich komunikatów usuwane z puli.

Na rysunku 3.2b przedstawiono sytuację, która ma miejsce chwilę później — środkowy (drugi) klient JMS wysyła komunikat kierowany pod adres *B*. Na tym etapie pierwsze dwa komunikaty zostały już przetworzone, zatem należy przyjąć, że kontener EJB zwrócił oba egzemplarze komponentów sterowanych komunikatami do właściwych pul. W reakcji na otrzymanie nowego komunikatu kontener wybiera nowy egzemplarz komponentu `MessageDrivenBean_2`, którego zadaniem będzie przetworzenie tego komunikatu.

Komponenty JMS-MDB zawsze są wdrażane z myślą o przetwarzaniu komunikatów kierowanych w określone miejsce docelowe. Przykładowo na rysunku 3.2 egzemplarze komponentu `MessageDrivenBean_1` przetwarzają tylko komunikaty kierowane pod adres *A*, natomiast egzemplarze komponentu `MessageDrivenBean_2` przetwarzają wyłącznie komunikaty z adresem docelowym *B*. Warto pamiętać, że istnieje możliwość jednoczesnego przetwarzania wielu komunikatów kierowanych pod ten sam adres. Przykładowo jeśli system EJB nagle otrzyma sto komunikatów kierowanych z adresem docelowym A, kontener EJB będzie musiał wybrać z puli sto egzemplarzy komponentu `MessageDrivenBean_1`, które przetworzą te komunikaty przychodzące (każdemu egzemplarzowi zostanie przydzielony inny komunikat).

W specyfikacji Enterprise JavaBeans 2.1 i kolejnych wersjach rozszerzono rolę komponentów sterowanych komunikatami, rezygnując z ograniczania ich możliwości do jednego systemu przesyłania komunikatów — obecnie komponenty MDB oprócz systemu JMS mogą współpracować także z innymi usługami i interfejsami API przesyłania komunikatów. Ten odważny krok otworzył komponenty sterowane komunikatami na niemal dowolne rodzaje zasobów, włącznie z systemami przesyłania komunikatów alternatywnymi względem systemu JMS, takimi systemami ERP jak SAP oraz istniejącymi systemami informatycznymi (np. IMS). Niezależnie od rodzaju zasobów reprezentowanych przez dany komponent sterowany komunikatami, jego egzemplarze są składowane w puli w dokładnie taki sam sposób jak egzemplarze komponentów JMS-MDB.

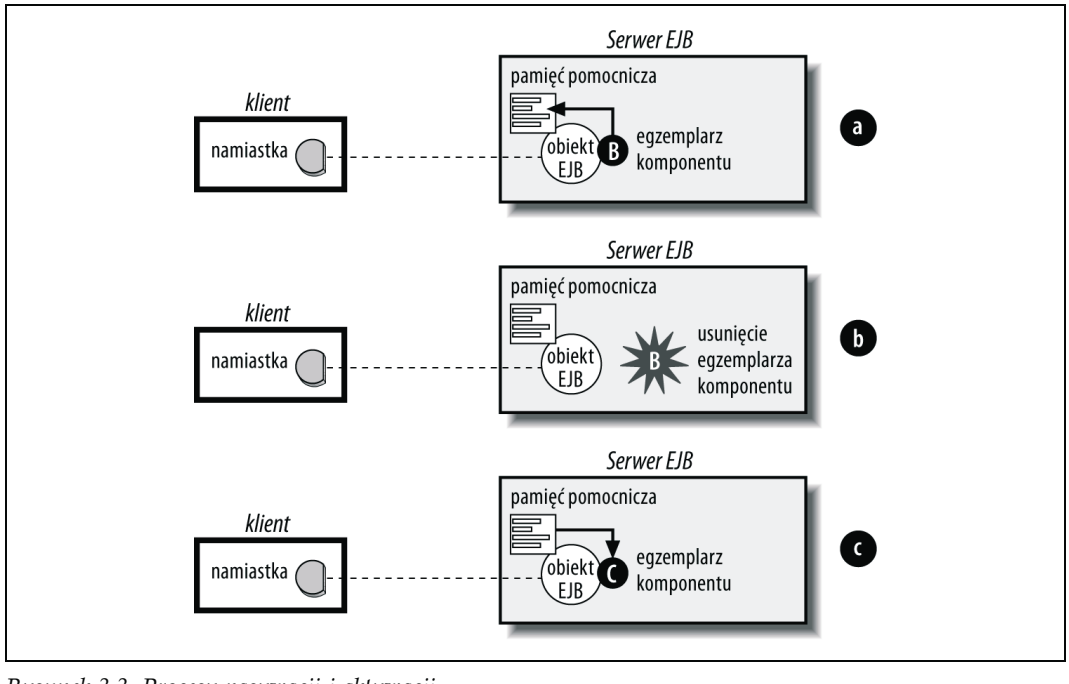
Mechanizm aktywacji

W przeciwieństwie do pozostałych rodzajów komponentów EJB stanowe komponenty sesyjne utrzymują swój stan pomiędzy kolejnymi wywołaniami metod. Tzw. **stan konwersacji** (ang. *conversational state*) reprezentuje konwersację z klientem stanowego komponentu sesyjnego. Integralność takiego stanu konwersacji wymaga jego utrzymywania przez cały czas trwania obsługi żądań klienta przez dany komponent. Inaczej niż w przypadku bezstanowych komponentów sesyjnych i komponentów sterowanych komunikatami egzemplarze stanowych komponentów sesyjnych nie są składowane w ramach puli. Zamiast tego stanowe komponenty sesyjne wykorzystują mechanizm aktywacji, który także pozwala na pewne oszczędności w zakresie wykorzystywanych zasobów. Kiedy serwer EJB musi zachować jakieś zasoby, może po prostu usunąć stanowe komponenty sesyjne z pamięci. Stan konwersacji usuniętego komponentu jest serializowany w pamięci pomocniczej. Kiedy klient wywołuje metodę

obiektu EJB, kontener EJB tworzy egzemplarz stanowego komponentu sesyjnego i odtwarza w nowym egzemplarzu stan zachowany w chwili usuwania z pamięci jego poprzednika.

Pasywacja (ang. *passivation*) polega na rozłączeniu stanowego komponentu sesyjnego od obiektu EJB oraz na zachowaniu jego stanu w pamięci pomocniczej. Pasywacja wymaga składowania stanu konwersacji pomiędzy egzemplarzem komponentu a odpowiednim obiektem EJB. Po wykonaniu pasywacji egzemplarz komponentu jest bezpiecznie usuwany zarówno z obiektu EJB, jak i z pamięci. Aplikacje klienckie w ogóle nie są informowane o procesach pasywacji. Warto pamiętać, że oprogramowanie klienckie wykorzystuje zdalną referencję do komponentu implementowaną przez namiastkę pośrednika EJB, zatem klient może być połączony z obiektem EJB także po przeprowadzeniu pasywacji.

Aktywacja (ang. *activation*) polega na przywróceniu stanu konwersacji egzemplarza stanowego komponentu sesyjnego z obiektem EJB. W momencie wywołania metody obiektu EJB, który był przedmiotem pasywacji, kontener automatycznie tworzy nowy egzemplarz odpowiedniego komponentu i przypisuje jego polom dane zapisane w czasie pasywacji. Obiekt EJB może następnie delegować wspomniane wywołanie metody do nowo utworzonego egzemplarza komponentu sesyjnego. Procesy aktywacji i pasywacji stanowego komponentu sesyjnego przedstawiono na rysunku 3.3. Rysunek 3.3a ilustruje sytuację, w której komponent podlega pasywacji. Stan konwersacji egzemplarza *B* z obsługiwanym obiektem EJB jest odczytywany i utrwalany. Na rysunku 3.3b przedstawiono moment pasywacji i zapisywania stanu. Od tej pory dany obiekt EJB nie jest już związany ze wspomnianym egzemplarzem komponentu. Aktywację tego komponentu przedstawiono na rysunku 3.3c. Rysunek ten prezentuje sytuację, w której nowy egzemplarz, nazwany *C*, jest tworzony, wypełniany danymi reprezentującymi stan zapisany w czasie pasywacji i ostatecznie wiązany z tym samym obiektem EJB, z którym przed pasywacją współpracował egzemplarz *B*.



Rysunek 3.3. Procesy pasywacji i aktywacji

Ponieważ sama klasa stanowego komponentu sesyjnego nie musi oferować możliwości serializacji, konkretne rozwiązania stosowane w mechanizmach aktywacji i pasywacji tego rodzaju komponentów zależą od producenta kontenera EJB. Warto pamiętać, że np. właściwości `transient` mogą nie być traktowane przez mechanizmy odpowiedzialne za aktywację i pasywację dokładnie tak, jak tego oczekujemy. Przykładowo w czasie deserializacji obiektu Javy jego polom przejściowym zawsze są przypisywane wartości początkowe właściwe dla ich typów. Pola typu `Integer` mają przypisywaną wartość `0`, pola typu `Boolean` mają przypisywaną wartość `false`, referencje do obiektów mają przypisywaną wartość `null` itd. W systemach EJB pola przejściowe aktywowanych komponentów nie mają przywracanych wartości początkowych, tylko albo zachowują swoje wartości oryginalne, albo wartości dowolne. Stosując pola przejściowe, należy zachowywać szczególną ostrożność, ponieważ ich stan po dokonaniu pasywacji i aktywacji zależy od implementacji.

Proces aktywacji jest obsługiwany przez metody zwrotne cyklu życia komponentu. W przeciwieństwie do specyfikacji Enterprise JavaBeans 2.1 specyfikacja Enterprise JavaBeans 3.0 nie nakłada na programistów klas bezstanowych komponentów sesyjnych obowiązku implementowania metod zwrotnych, które nie są potrzebne (niezaimplementowane metody oczywiście nie są udostępniane przez interfejs komponentu sesyjnego). Za pośrednictwem metod zwrotnych (oznaczonych stosownymi adnotacjami) programiści komponentów mogą uzyskiwać sygnały o zdarzeniach mających związek z cyklem życia ich komponentów. Przykładowo metoda oznaczona adnotacją `@javax.ejb.PostActivate` (jeśli istnieje) jest wywoływana natychmiast po pomyślnym zakończeniu procesu aktywacji egzemplarza komponentu. W ciele tej metody można np. „wyzerować" wartości pól przejściowych przypisując im wartości początkowe. Metoda oznaczona adnotacją `@javax.ejb.PrePassivate` (jeśli istnieje) jest wywoływana bezpośrednio przed przystąpieniem do pasywacji egzemplarza danego komponentu. Metody wyróżnione tą parą adnotacji są szczególnie przydatne, jeśli egzemplarze naszego komponentu utrzymują połączenia z zasobami, które powinny być zamykane lub zwalniane przed przeprowadzeniem pasywacji oraz ponownie uzyskiwane lub odtwarzane po aktywacji. Ponieważ egzemplarze stanowych komponentów sesyjnych w czasie pasywacji są usuwane z pamięci, otwarte połączenia z zasobami z natury rzeczy nie mogą być zachowywane. Do wyjątków należą zdalne referencje do pozostałych komponentów oraz kontekstu sesji (reprezentowanego przez egzemplarz interfejsu `SessionContext`), które muszą być zachowywane wraz z serializowanym stanem komponentu i odtwarzane w momencie jego aktywowania. Specyfikacja Enterprise JavaBeans dodatkowo wymaga zachowywania w procesie pasywacji referencji do kontekstu środowiska JNDI, interfejsów komponentu, usługi `EntityManager` oraz obiektu `UserTransaction`.

Architektura Java EE Connector Architecture

Architektura Java EE Connector Architecture definiuje interfejs łączący korporacyjne systemy informacyjne (ang. *Enterprise Information Systems — EIS*) z systemami kontenerów Javy EE (w tym kontenerami EJB i serwletów). EIS jest ogólnym terminem stosowanym w odniesieniu do systemów informacyjnych, włącznie z serwerami relacyjnych baz danych, oprogramowaniem pośredniczącym (np. MQSeries lub SonicMQ), systemami w architekturze CORBA, systemami ERP (np. SAP, PeopleSoft czy JD Edwards) oraz istniejącymi systemami informatycznymi (np. IMS czy CICS).

Java EE definiuje (oprócz architektury Enterprise JavaBeans) szereg standardowych interfejsów API dla rozwiązań korporacyjnych, w tym takie interfejsy jak JDBC, JMS, JNDI, Java IDL czy JavaMail. Każdy z tych interfejsów API oferuje niezależne od producentów mechanizmy dla ściśle określonego rodzaju korporacyjnych systemów informatycznych. Przykładowo interfejs JDBC służy do wymiany informacji z relacyjnymi bazami danych, JMS jest oprogramowaniem pośredniczącym systemu przesyłania komunikatów, JNDI jest zbiorem usług nazewnictwa i usług katalogowych, JavaMail stworzono z myślą o obsłudze poczty elektronicznej, natomiast Java IDL jest interfejsem opracowanym z myślą o architekturze CORBA. Obowiązek implementowania obsługi tych interfejsów API jest gwarancją przenośności komponentów EJB pomiędzy środowiskami różnych producentów.

Mimo że interfejsy API dla rozwiązań korporacyjnych w założeniu mają być niezależne od producentów konkretnych rozwiązań, produkty kryjące się za tymi interfejsami zawsze wykazują cechy właściwe wyłącznie dla ich producentów. Kiedy komponent EJB korzysta z tego rodzaju interfejsu API, kontener EJB odpowiada za właściwe zarządzanie pulą utrzymywanych połączeń z systemem EIS, włączanie tego systemu do obsługiwanych transakcji, propagowanie danych uwierzytelniających itp. Wymienione zadania często wymagają od kontenera EJB współpracy z systemami EIS z wykorzystaniem technik, których nie przewidzieli lub nie udokumentowali twórcy tych uniwersalnych interfejsów API. W efekcie każdy producent rozwiązań pisanych w Javie EE musi tworzyć własny, niestandardowy kod współpracujący z ewentualnymi systemami EIS. W związku z tym producenci rozwiązań Java EE wybierają dla każdego standardowego interfejsu API obsługiwane systemy EIS. Takie podejście ma poważny wpływ na zakres obsługi poszczególnych systemów EIS przez producentów rozwiązań EJB — przykładowo producent *A* może obsługiwać połączenia za pośrednictwem interfejsu JDBC z systemami baz danych Oracle, natomiast producent *B* może implementować obsługę tylko połączeń z bazami danych DB2.

Konektory JCA 1.5

Specyfikacja Enterprise JavaBeans 2.0 nakładała na twórców komponentów EJB obowiązek implementowania obsługi architektury Java EE Connector Architecture, która stanowiła ogromny krok na drodze do rozwiązania opisywanych problemów. Z drugiej strony, proponowane rozwiązanie okazały się niewystarczające. W szczególności nie udało się rozwiązać problemu braku obsługi tzw. **trybu wpychania** (ang. *push model*) w systemach przesyłania komunikatów, co było o tyle istotne, że wiele systemów EIS (np. JMS) „wpychało" dane klientom mimo braku odpowiednich żądań. Zarówno specyfikacja Enterprise JavaBeans 2.1, jak i specyfikacja Enterprise JavaBeans 3.0 wymagają obsługi architektury Java EE Connector Architecture 1.5, która obsługuje tryb wpychania. Z myślą o obsłudze trybu wpychania twórcy architektury JCA 1.5 wykorzystali model programowania komponentów sterowanych komunikatami. W szczególności architektura JCA 1.5 definiuje interfejs kontener-konektor, który umożliwia przetwarzanie komunikatów przychodzących (wysyłanych asynchronicznie przez system EIS) przez komponenty sterowane komunikatami. Przykładowo producent *X* mógłby opracować konektor Java EE dla agenta **Mail Delivery Agent** (MDA) pełniący funkcję oprogramowania odpowiedzialnego za dostarczanie wiadomości poczty elektronicznej. W ramach tego procesu producent *X* może wówczas zdefiniować interfejs nasłuchiwania komunikatów nazwany `EmailListenr`, który powinien być implementowany przez komponenty poczty elektronicznej sterowane komunikatami odpowiedzialne za przetwarzanie wiadomości poczty elektronicznej. Agent MDA „wpycha" otrzymywane z internetu wiadomości poczty elektronicznej

do kontenera EJB, który z kolei deleguje otrzymywane komunikaty do egzemplarzy odpowiednich komponentów sterowanych komunikatami. Programista aplikacji powinien następnie napisać komponent poczty elektronicznej sterowany komunikatami oznaczony adnotacją `@javax.ejb.MessageDriven` i implementujący wspominany już interfejs `com.producent.EmailListener`. Ostatecznie opracowany i wdrożony komponent poczty elektronicznej sterowany komunikatami może przetwarzać komunikaty przychodzące.

Usługi podstawowe

Istnieje wiele wartościowych usług opracowanych z myślą o aplikacjach rozproszonych. W niniejszej książce szczegółowo przeanalizujemy osiem najważniejszych usług nazywanych **usługami podstawowymi** (ang. *primary services*) z uwagi na konieczność ich implementowania przez wszystkie kompletne platformy Enterprise JavaBeans. Usługi podstawowe oferują mechanizmy w takich obszarach jak współbieżność, przetwarzanie transakcyjne, utrwalanie danych, obsługa obiektów rozproszonych, asynchroniczne przesyłanie komunikatów, licznik czasowy, nazewnictwo i bezpieczeństwo. Serwery EJB automatycznie zarządzają wszystkimi wymienionymi usługami podstawowymi. Takie rozwiązanie zwalnia programistów aplikacji z trudnego obowiązku samodzielnego implementowania wszystkich lub części spośród wymienionych rozwiązań. Zamiast tracić czas na implementowanie tego rodzaju mechanizmów, programiści mogą się koncentrować wyłącznie na definiowaniu logiki aplikacji opisującej określony wycinek działalności biznesowej — za dostarczanie wszelkich niezbędnych usług na poziomie systemowym odpowiada serwer EJB. W kolejnych punktach omówimy poszczególne usługi podstawowe i opiszemy wymagany zakres oferowanej funkcjonalności w ramach tych usług (zgodnie ze specyfikacją EJB).

Współbieżność

Chociaż współbieżność (ang. *concurrency*) jako taka jest istotna z perspektywy programistów komponentów wszystkich typów, w przypadku pewnych rodzajów komponentów ma nieco inne znaczenie niż w przypadku innych typów.

Współbieżność w pracy komponentów sesyjnych i encyjnych

Komponenty sesyjne nie obsługują dostępu współbieżnego. To ograniczenie jest w pełni uzasadnione, jeśli uwzględnimy faktyczny charakter stanowych i bezstanowych komponentów sesyjnych. Stanowy komponent sesyjny jest rozszerzeniem pojedynczego klienta i pracuje wyłącznie w jego imieniu. W związku z tym oferowanie współbieżnego dostępu do tego rodzaju komponentów całkowicie mijałoby się z ich właściwym przeznaczeniem, ponieważ i tak są wykorzystywane przez te aplikacje klienckie, które je utworzyły. Także bezstanowe komponenty sesyjne nie muszą oferować współbieżności, ponieważ i tak nie utrzymują stanu, który wymagałby współdzielenia (współbieżnego dostępu). Zasięg operacji wykonywanych przez bezstanowe komponenty sesyjne ogranicza się do zasięgu odpowiednich wywołań metod. Ponieważ ani stanowe, ani bezstanowe komponenty sesyjne nie reprezentują danych współdzielonych, w ich przypadku serwer EJB nie musi implementować usług zarządzających współbieżnością.

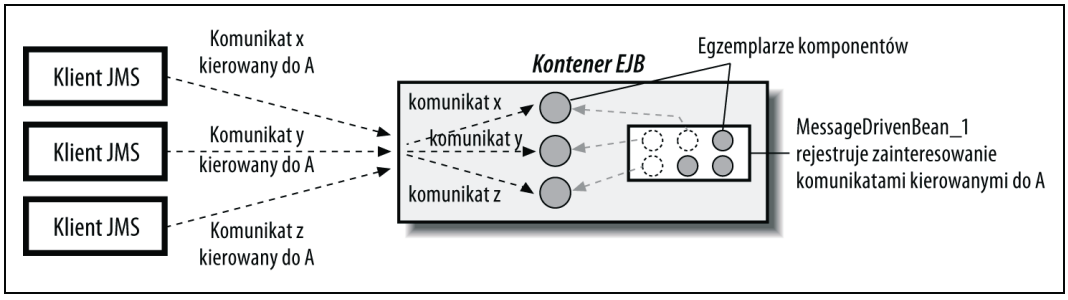
Ponieważ za obsługę współbieżności odpowiadają serwery EJB, metody samych komponentów nie muszą gwarantować bezpieczeństwa przetwarzania wielowątkowego. W rzeczywistości specyfikacja Enterprise JavaBeans wręcz zakazuje programistom komponentów EJB stosowania słowa kluczowego `synchronized`. Zakaz używania w kodzie podstawowych konstrukcji synchronizujących pracę wątków skutecznie uniemożliwia programistom podejmowanie prób samodzielnego sterowania synchronizacją i — tym samym — przekłada się na większą wydajność egzemplarzy komponentów w czasie wykonywania. Co więcej, specyfikacja Enterprise JavaBeans wprost zakazuje komponentom tworzenia własnych wątków. Innymi słowy, programista nie może z poziomu swoich komponentów tworzyć nowych wątków. Zachowywanie pełnej kontroli nad komponentem należy do kontenera EJB, który musi właściwie zarządzać współbieżnością, przetwarzaniem transakcyjnym oraz utrwalaniem danych. Dowolność w kwestii tworzenia wątków przez programistę komponentu uniemożliwiłaby kontenerowi nie tylko śledzenie działań komponentu, ale także właściwe zarządzanie podstawowymi usługami.

Komponenty encyjne reprezentują dane współdzielone i jako takie mogą być przedmiotem dostępu współbieżnego. Komponenty encyjne zaliczamy do tzw. komponentów współdzielonych. Przykładowo w systemie EJB linii żeglugowych Titan Cruises są reprezentowane trzy statki: *Paradise*, *Utopia* i *Valhalla*. Komponent encyjny *Ship* reprezentujący statek *Utopia* w dowolnym momencie może być adresatem żądań generowanych przez setki aplikacji klienckich. Aby współbieżny dostęp do komponentu encyjnego był możliwy, mechanizm odpowiedzialny za utrwalanie danych musi odpowiednio chronić dane reprezentowane przez taki komponent współdzielony (mimo następującego jednocześnie dostępu wielu aplikacji klienckich do jednego logicznego komponentu encyjnego).

Specyfikacja Java Persistence przewiduje, że kontener odpowiedzialny za ochronę współdzielonych danych komponentów encyjnych powinien tworzyć po jednej kopii egzemplarza komponentu encyjnego dla każdej z wykonywanych transakcji. Właśnie istnienie migawki danej encji dla każdej transakcji umożliwia prawidłową i bezpieczną obsługę współbieżnego, wielowątkowego dostępu. Warto się więc zastanowić, jak to możliwe, że kontener skutecznie chroni komponenty encyjne przed fałszywymi odczytami oraz próbami jednoczesnej aktualizacji przez wiele transakcji. Jednym z możliwych rozwiązań jest stosowanie strategii tzw. współbieżności optymistycznej z wykorzystaniem prostego mechanizmu pól reprezentujących wersje. Innym rozwiązaniem jest użycie opcji `SERIALIZED` dla poziomu izolacji interfejsu JDBC. Implementacje tworzone przez rozmaitych producentów mogą wykorzystywać własne, niestandardowe mechanizmy bezpośredniego blokowania dostępu do informacji na poziomie bazy danych. Wszystkie te zagadnienia zostaną szczegółowo omówione w rozdziale 16.

Współbieżność w pracy komponentów sterowanych komunikatami

W przypadku komponentów sterowanych komunikatami określenie **współbieżność** odnosi się do jednoczesnego przetwarzania więcej niż jednego komunikatu. Gdyby komponent sterowany komunikatami mógł w danej chwili przetwarzać tylko jeden komunikat, jego przydatność dla rzeczywistych aplikacji byłaby znikoma, ponieważ systemy złożone z takich komponentów nie mogłyby sobie radzić z dużymi obciążeniami (wyrażanymi w liczbie komunikatów generowanych w określonych przedziałach czasowych). Na rysunku 3.4 przedstawiono sytuację, w której do przykładowego systemu EJB docierają trzy komunikaty wygenerowane jednocześnie przez trzy różne aplikacje klienckie — trzy egzemplarze pojedynczego komponentu JMS-MDB, które zarejestrowały swoje zainteresowanie tego rodzaju komunikatami, mogą jednocześnie przetwarzać wspomniane komunikaty.



Rysunek 3.4. Przetwarzanie współbieżne z wykorzystaniem komponentów sterowanych komunikatami

Także te komponenty sterowane komunikatami, które implementują inne interfejsy API niż JMS, mogą korzystać z tych samych usług współbieżności co komponenty JMS-MDB. Egzemplarze wszystkich rodzajów komponentów sterowanych komunikatami są składowane w puli i wykorzystywane do współbieżnego przetwarzania komponentów przychodzących — dzięki temu istnieje możliwość jednoczesnej obsługi setek lub wręcz tysięcy komunikatów generowanych przez aplikacje klienckie[1].

Transakcje

Transakcja jest jednostką pracy lub zbiorem zadań wykonywanych sekwencyjnie w odpowiedzi na jedno żądanie. Transakcje są atomowe, co oznacza, że pojedynczą transakcję można uznać za prawidłowo wykonaną wtedy i tylko wtedy, gdy uda się zrealizować wszystkie zadania składające się na tę transakcję. W poprzednim rozdziale wielokrotnie mówiliśmy o komponencie EJB *TravelAgent* w kontekście sposobu kontrolowania przez komponenty sesyjne interakcji z pozostałymi komponentami. Poniżej przedstawiono metodę `bookPassage()`, którą opisano w rozdziale 2.:

```
public Reservation bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
     if (customer == null || cruise == null || cabin == null) {
         throw new IncompleteConversationalState( );
     }
     try {
          Reservation reservation =
             new Reservation(customer,cruise,cabin,price,new Date( ));
          entityManager.persist(reservation);
          process.byCredit(customer,card,price);
          return reservation;
     } catch(Exception e) {
         throw new EJBException(e);
     }
}
```

Metoda `bookPassage()` realizuje dwa zadania, które muszą być wykonane albo razem, albo wcale — tymi zadaniami jest odpowiednio utworzenie nowego egzemplarza *Reservation* oraz przetworzenie płatności. Kiedy komponent EJB *TravelAgent* jest wykorzystywany do rezerwacji miejsca w kajucie dla nowego pasażera, musi zostać skutecznie przeprowadzone za-

[1] W praktyce jednoczesne przetwarzanie czegokolwiek w sytuacji, gdy nie dysponujemy wieloma procesorami, jest bardzo trudne, jednak na poziomie pojęciowym nasze stwierdzenie jest prawdziwe. Wiele wątków w ramach tej samej wirtualnej maszyny Javy lub w ramach wielu maszyn wirtualnych korzystających z tego samego procesora (fizycznego układu obliczeniowego) mogą skutecznie imitować przetwarzanie współbieżne.

równo pobranie stosownej kwoty z karty kredytowej pasażera, jak i utworzenie nowej encji reprezentującej samą rezerwację. Gdyby inny komponent EJB, *ProcessPayment*, pobrał kwotę z karty kredytowej pasażera w sytuacji, gdy próba utworzenia nowej encji *Reservation* zakończyła się niepowodzeniem, należałoby to działanie uznać za niewłaściwe. Podobnie nie powinniśmy tworzyć nowej rezerwacji, jeśli nie uda się pobrać stosownej kwoty z karty kredytowej pasażera. Oznacza to, że serwer EJB musi uważnie monitorować tego rodzaju transakcje, aby zagwarantować właściwe wykonywanie wszystkich zadań.

Transakcje są zarządzane automatycznie, zatem programiści komponentów EJB nie muszą stosować żadnych interfejsów API odpowiedzialnych za zarządzanie zaangażowaniem tworzonych komponentów w przetwarzanie transakcyjne. Aby zasygnalizować serwerowi EJB sposób, w jaki powinien zarządzać komponentem w czasie działania, wystarczy w czasie wdrażania zadeklarować odpowiednie atrybuty transakcyjne. Okazuje się jednak, że specyfikacja EJB definiuje mechanizm umożliwiający bezpośrednie zarządzanie transakcjami tym komponentom, w przypadku których jest to niezbędne. Atrybuty transakcyjne ustawiane w czasie wdrażania, techniki zarządzania transakcjami wprost przez programistów komponentów EJB oraz pozostałe zagadnienia związane z przetwarzaniem transakcyjnym zostaną omówione w rozdziale 16.

Trwałość

Komponenty encyjne reprezentują zachowania i dane właściwe dla osób, miejsc lub przedmiotów. W przeciwieństwie do komponentów sesyjnych i komponentów sterowanych komunikatami komponenty encyjne mają charakter trwały, co oznacza, że ich stan jest zapisywany (utrwalany) w bazie danych. Mechanizm utrwalania umożliwia zachowywanie encji w sposób umożliwiający uzyskiwanie dostępu zarówno do zachowań, jak i do danych komponentów encyjnych w dowolnym czasie (bez konieczności samodzielnego odzyskiwania tych zachowań i danych w razie ewentualnej awarii).

Java Persistence

W specyfikacji Enterprise JavaBeans 3.0 całkowicie zweryfikowano dotychczasowe podejście do problemu utrwalania — zdecydowano się nawet na wyłączenie tej problematyki do odrębnej, przebudowanej specyfikacji Java Persistence. O ile specyfikacja Enterprise JavaBeans 2.1 proponowała model, w którym utrwalanie było realizowane na poziomie komponentów, zgodnie z nową specyfikacją Java Persistence trwałe komponenty encyjne mają postać zwykłych obiektów Javy (nazywanych obiektami POJO). Encje można tworzyć poza środowiskiem kontenera EJB. Sam proces ich tworzenia niczym się nie różni od procesów tworzenia wszystkich innych obiektów Javy z wykorzystaniem standardowego operatora `new()`. Co więcej, komponent encyjny może być w dowolnym momencie włączany do kolekcji komponentów zarządzanych przez kontener lub wykluczany z tego zbioru. Za wiązanie egzemplarzy komponentów encyjnych z pamięcią trwałą (najczęściej bazą danych) odpowiada usługa `EntityManager`. Usługa `EntityManager` oferuje metody umożliwiające tworzenie, odnajdywanie, odczytywanie, usuwanie i aktualizowanie komponentów encyjnych. Po połączeniu egzemplarza komponentu encyjnego z pamięcią trwałą kontener EJB odpowiada za zarządzanie trwałym stanem tego komponentu i automatyczną synchronizację z odpowiednim źródłem danych.

Szczególnie interesującym aspektem modelu proponowanego w specyfikacji Java Persistence jest możliwość odłączania egzemplarzy komponentów encyjnych od kontenera EJB. Egzemplarze komponentów EJB z reguły są odłączane od kontenera w chwili zakończenia wykonywania bieżącej transakcji. Warto pamiętać, że tak odłączone egzemplarze można swobodnie przesyłać za pośrednictwem sieci zdalnym aplikacjom klienckim lub wręcz zapisywać na dysku. Stan tego rodzaju egzemplarzy może być modyfikowany, a same egzemplarze komponentów można ponownie łączyć z kontenerem EJB za pomocą metody `EntityManager.merge()`. W momencie ponownego łączenia egzemplarza komponentu encyjnego z kontenerem EJB wszelkie zmiany dokonane na danym komponencie automatycznie są synchronizowane z zawartością pamięci trwałej. Nowy model utrwalania danych umożliwia programistom komponentów EJB rezygnację ze starej koncepcji obiektów transferu danych (ang. *Data Transfer Object*), co w znacznym stopniu upraszcza architekturę konstruowanych aplikacji. Szczegółowe omówienie tego zagadnienia można znaleźć w rozdziale 5.

Utrwalanie obiektowo-relacyjne

Utrwalanie obiektowo-relacyjne (ang. *object-to-relational — O/R*) wiąże się z koniecznością odwzorowywania stanu komponentów encyjnych w tabelach i kolumnach relacyjnej bazy danych. Ponieważ relacyjne bazy danych są wykorzystywane przez 99 procent aplikacji korzystających z baz danych, grupa ekspertów zaangażowana w prace nad nową specyfikacją Enterprise JavaBeans (EJB 3.0 Expert Group) ustaliła, że opracowanie mechanizmu odwzorowań obiektowo-relacyjnych będzie dużo lepszym rozwiązaniem niż podejmowanie skazanych na niepowodzenie prób tworzenia jednej, uniwersalnej architektury utrwalania. Wskutek tej decyzji powstała specyfikacja Java Persistence obejmująca bogaty zbiór mechanizmów odpowiedzialnych za odwzorowywanie komponentów encyjnych w relacyjnych bazach danych z uwzględnieniem takich technik jak dziedziczenie, odwzorowania w wielu tabelach, zarządzanie wersjami czy obsługa rozszerzonego języka zapytań EJBQL. Ponieważ odwzorowania obiektowo-relacyjne są precyzyjnie definiowane przez konkretną specyfikację, współczesne aplikacje EJB oferują **dużo większą** przenośność pomiędzy rozwiązaniami różnych producentów, ponieważ w znacznym stopniu wyeliminowano konieczność stosowania metadanych właściwych dla poszczególnych produktów.

Przeanalizujmy teraz prosty przykład stosowania techniki odwzorowań obiektowo-relacyjnych w praktyce. W systemie informatycznym linii żeglugowych Titan Cruises klasa `Cabin` modeluje kajutę na statku. Klasa `Cabin` definiuje trzy pola składowe: `name`, `deckLevel` oraz `id`. Definicję klasy `Cabin` przedstawiono poniżej:

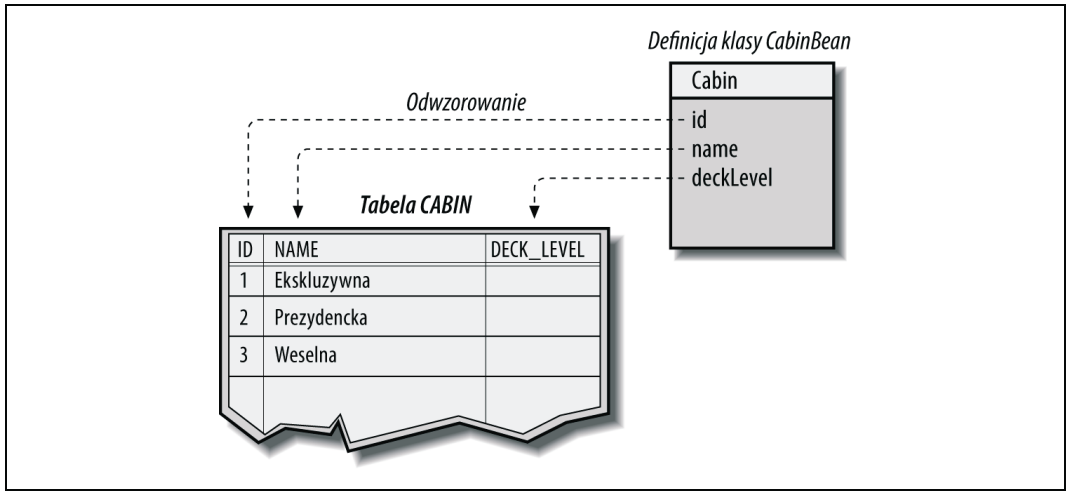
```
@Entity
@Table(name="CABIN")
public class Cabin {
    private int id;
    private String name;
    private int deckLevel;

    @Column(name="NAME")
    public String getName( ) { return name; }
    public void setName(String str) { name = str; }

    @Column(name="DECK_LEVEL")
    public int getDeckLevel( ) { return deckLevel; }
    public void setDeckLevel(int level) { deckLevel = level; }
```

```
    @Id
    @Column(name="ID")
    public int getId( ) { return id; }
    public void setId(int id) { this.id = id; }
}
```

W prezentowanym przykładzie metody akcesorów reprezentują pola komponentu encyjnego zarządzane przez kontener EJB. Skoro stosujemy mechanizm odwzorowań obiektowo-relacyjnych, możemy przyjąć, że pola komponentu encyjnego odpowiadają kolumnom relacyjnej bazy danych. Metadane opisujące odwzorowania obiektowo-relacyjne należy definiować w formie adnotacji poprzedzających zarówno metody akcesorów (`@Column` oraz `@Id`), jak i klasę komponentu (`@Table`). Przykładowo pole `deckLevel` klasy `Cabin` jest odwzorowywane w kolumnie `DECK_LEVEL` tabeli nazwanej `CABIN` i wchodzącej w skład relacyjnej bazy danych linii Titan Cruises. Na rysunku 3.5 przedstawiono graficzny schemat tego odwzorowania.



Rysunek 3.5. Schemat odwzorowania obiektowo-relacyjnego komponentów encyjnych

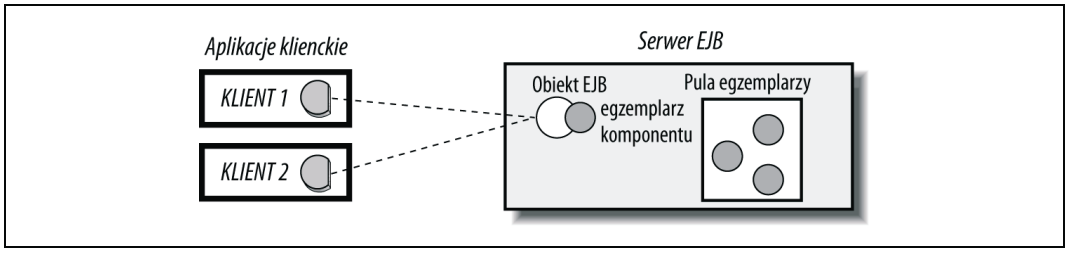
Kiedy pola komponentu encyjnego zostaną już odwzorowane w relacyjnej bazie danych, kontener bierze na siebie odpowiedzialność za utrzymywanie zgodności stanu danego egzemplarza komponentu z zawartością odpowiednich tabel bazy danych. Proces utrzymywania tej zgodności bywa nazywany **synchronizacją** stanu egzemplarza komponentu. W przypadku klasy `Cabin` egzemplarze komponentu encyjnego są odwzorowywane w odrębnych wierszach tabeli `CABIN` relacyjnej bazy danych. Oznacza to, że modyfikacja egzemplarza komponentu encyjnego *Cabin* wymaga jego zapisania we właściwym wierszu bazy danych. Warto pamiętać, że niektóre z komponentów są odwzorowywane w więcej niż jednej tabeli bazy danych. Tego rodzaju odwzorowania są znacznie bardziej skomplikowane i często wymagają stosowania między innymi złączeń SQL-a i wielokrotnych aktualizacji — złożone odwzorowania obiektowo-relacyjne przeanalizujemy w dalszej części tej książki.

Specyfikacja Java Persistence dodatkowo definiuje pola relacji komponentów encyjnych, które umożliwiają wchodzenie tego rodzaju komponentów w relacje „jeden do jednego", „jeden do wielu" oraz „wiele do wielu" z pozostałymi komponentami. Co więcej, komponenty encyjne same mogą utrzymywać kolekcje innych komponentów encyjnych lub pojedyncze referencje. Model Java Persistence szczegółowo omówiono w rozdziałach od 5. do 10.

Obiekty rozproszone

Kiedy mówimy o interfejsach komponentów oraz innych klasach i interfejsach technologii EJB wykorzystywanych przez oprogramowanie klienckie, mamy na myśli perspektywę klienta danego systemu EJB. **Perspektywa klienta EJB** (ang. *EJB client view*) nie obejmuje egzemplarzy klas komponentów sesyjnych, kontenera EJB, mechanizmu wymiany egzemplarzy ani żadnych innych szczegółów związanych z implementacją poszczególnych komponentów sesyjnych. Z perspektywy klientów zdalnych komponent jest definiowany przez interfejs zdalny lub interfejs punktu końcowego[2]. Wszystkie inne elementy, włącznie z mechanizmem wykorzystywanym do obsługi obiektów rozproszonych, są niewidoczne. Jeśli wykorzystywany serwer EJB prawidłowo obsługuje perspektywę danego klienta EJB, w komunikacji pomiędzy tymi węzłami może być stosowany dowolny protokół obiektów rozproszonych. Specyfikacja Enterprise JavaBeans 3.0 mówi, że każdy serwer EJB musi obsługiwać protokół Java RMI-IIOP, co nie oznacza, że serwery EJB nie mogą obsługiwać także innych protokołów (w tym interfejsu Java RMI API oraz protokołu CORBA IIOP). Specyfikacja EJB 3.0 dodatkowo nakłada na producentów serwerów obowiązek implementowania obsługi protokołu SOAP 1.2 za pośrednictwem interfejsu JAX-RPC API.

Niezależnie od wykorzystywanego protokołu serwer EJB musi obsługiwać żądania klientów Javy z wykorzystaniem odpowiedniego interfejsu API tych klientów, zatem stosowany protokół powinien oferować możliwość odwzorowywania do modelu programowania Java RMI-IIOP lub JAX-RPC. Na rysunku 3.6 przedstawiono przykład interfejsu EJB API języka Java obsługiwanego przez różne protokoły obiektów rozproszonych.



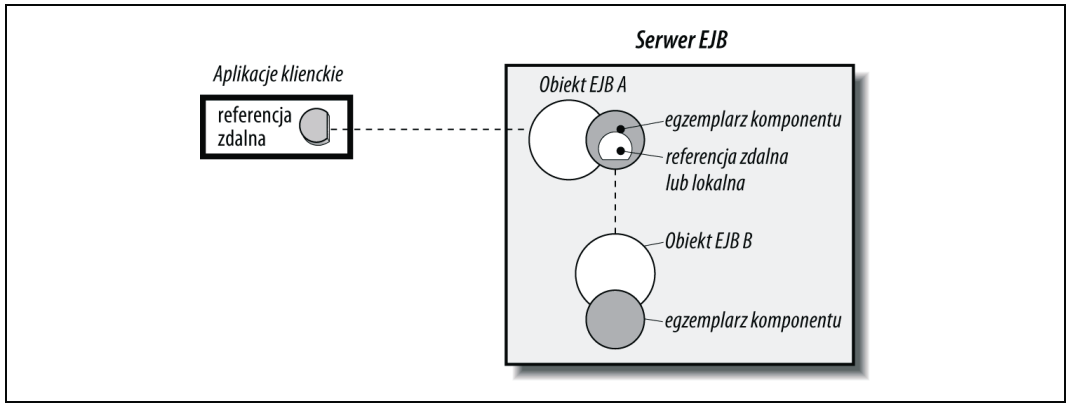
Rysunek 3.6. Perspektywa klienta systemu EJB obsługiwana przez różne protokoły

Specyfikacja Enterprise JavaBeans przewiduje możliwość uzyskiwania przez oprogramowanie klienckie napisane w językach innych niż Java dostępu do komponentów EJB (pod warunkiem, że producent serwera EJB zaimplementuje odpowiednie mechanizmy). Przykładem takiego rozwiązania jest mechanizm odwzorowań EJB-CORBA opracowany przez firmę Sun[3]. Dokument opublikowany przez Sun Microsystems opisuje język definiowania interfejsów architektury CORBA (ang. *CORBA Interface Definition Language — CORBA IDL*), za pomocą którego można uzyskiwać dostęp do komponentów EJB z poziomu klientów technologii CORBA. Aplikacje klienckie architektury CORBA można pisać w dowolnych językach programowania, włącznie z takimi językami jak C++, Smalltalk, Ada czy COBOL. Wspomniany mechanizm odwzorowujący dodatkowo obejmuje szczegółowe rozwiązania w zakresie obsługi

[2] Nieco inaczej jest w przypadku komponentów encyjnych, ponieważ egzemplarze klas tego rodzaju komponentów mogą być odłączane od kontenera i wysyłane do zdalnego klienta (pod warunkiem, że wspomniane klasy implementują interfejs `java.io.Serializable`).

[3] *Enterprise JavaBeans™ to CORBA Mapping, Version 1.1*, Sanjeev Krishnan, Sun Microsystems, 1999.

perspektywy klientów Java EJB, odwzorowań systemu nazewnictwa architektury CORBA w system nazewnictwa serwerów EJB oraz rozproszonymi transakcjami obejmującymi swoim zasięgiem zarówno obiekty CORBA, jak i komponenty EJB. Innym ciekawym przykładem jest odwzorowanie EJB-SOAP zbudowane na bazie technologii JAX-RPC. Odwzorowanie EJB-SOAP umożliwia aplikacjom klienckim SOAP napisanym w takich językach jak Visual Basic .NET, C# czy Perl uzyskiwanie dostępu do bezstanowych komponentów sesyjnych. Na rysunku 3.7 przedstawiono możliwe rozwiązania w zakresie uzyskiwania dostępu do serwera EJB z poziomu innych obiektów rozproszonych.



Rysunek 3.7. Komponent EJB udostępniany różnym rozproszonym aplikacjom klienckim

Asynchroniczne przesyłanie komunikatów

Przed wydaniem specyfikacji Enterprise JavaBeans 2.0 asynchroniczne przesyłanie komunikatów nie było zaliczane do zbioru usług podstawowych, ponieważ implementacja obsługi tego rodzaju komunikatów nie była konieczna do stworzenia kompletnej platformy EJB. Z drugiej strony, wprowadzenie komponentów sterowanych komunikatami spowodowało istotny wzrost znaczenia asynchronicznego przesyłania komunikatów za pomocą systemu JMS, które ostatecznie zostało uznane za jedną z usług podstawowych.

Obsługa przesyłania komunikatów wymaga od kontenera EJB implementowania niezawodnych mechanizmów kierowania komunikatów pochodzących od oprogramowania klienckiego JMS do właściwych komponentów JMS-MDB. Niezawodne kierowanie komunikatów do miejsc docelowych wymaga czegoś więcej niż semantyki podobnej do tej znanej z poczty elektronicznej czy nawet interfejsu JMS API. Systemy korporacyjne z natury rzeczy wymagają solidnych mechanizmów przekazywania komunikatów, co w przypadku systemu JMS wiąże się z koniecznością ponownego wysyłania komunikatów, których dostarczenie okazało się niemożliwe[4]. Co więcej, komunikaty w systemach EJB mogą mieć charakter trwały, co oznacza, że mogą być składowane na dysku lub w bazie danych do momentu, w którym będzie je można dostarczyć do właściwych adresatów. Komunikaty trwałe muszą być zachowywane

[4] Większość producentów kontenerów EJB ogranicza maksymalną liczbę prób ponownego wysyłania niedostarczonych komunikatów. Jeśli liczba takich prób przekroczyła określony próg, odpowiedni komunikat można umieścić w specjalnym repozytorium „martwych komunikatów", gdzie mogą być przeglądane przez administratora.

mimo ewentualnych awarii systemowych — komunikaty trwałe muszą zostać dostarczone nawet wówczas, jeśli awarii ulegnie serwer EJB (choćby miało to nastąpić po ponownym uruchomieniu tego serwera). Co ciekawe, przesyłanie komunikatów w systemie EJB ma charakter transakcyjny. Oznacza to, że jeśli komponent JMS-MDB nie zdoła prawidłowo przetworzyć otrzymanego komunikatu, automatycznie trzeba będzie przerwać całą transakcję i wymusić na kontenerze EJB ponowne dostarczenie tego samego komunikatu do innego egzemplarza komponentu sterowanego komunikatami.

Okazuje się, że komunikaty JMS mogą być wysyłane także przez komponenty sterowane komunikatami, bezstanowe komponenty sesyjne oraz komponenty encyjne. W niektórych przypadkach możliwość wysyłania komunikatów jest w przypadku standardowych komponentów Enterprise JavaBeans równie istotna jak w przypadku komponentów JMS-MDB — obsługa obu rozwiązań z reguły jest implementowana w bardzo podobny sposób.

Usługa licznika czasowego EJB

Usługa licznika czasowego EJB (ang. *EJB Timer Service*) może służyć do konstruowania harmonogramów wysyłania powiadomień do komponentów EJB w określonych odstępach czasu. Liczniki czasowe znajdują zastosowanie w wielu różnych aplikacjach. Przykładowo systemy bankowe mogą wykorzystywać tego rodzaju liczniki w roli zabezpieczeń kredytów hipotecznych, a konkretnie mechanizmu weryfikującego terminowość dokonywanych spłat. System obsługujący handel akcjami na giełdzie papierów wartościowych może korzystać z tej usługi do ustanawiania okresów ważności składanych zleceń. System obsługujący roszczenia klientów firmy ubezpieczeniowej może stosować liczniki czasowe do automatycznego inicjowania okresowych kontroli pod kątem ewentualnych wyłudzeń. Liczniki czasowe mogą być stosowane we wszelkich aplikacjach wymagających samokontroli i przetwarzania wsadowego.

Liczniki czasowe mogą być ustawiane w komponentach encyjnych, bezstanowych komponentach sesyjnych oraz komponentach sterowanych komunikatami. Warto pamiętać, że komponenty sesyjne i encyjne same ustawiają liczniki czasowe. Przykładowo, w chwili przyznania kredytu hipotecznego komponent encyjny reprezentujący ten kredyt może ustawić licznik umożliwiający weryfikację uregulowania kolejnej raty, który będzie zerowany po każdej prawidłowej wpłacie. Niektóre kontenery Enterprise JavaBeans oferują obsługę liczników czasowych wykorzystywanych także przez komponenty sterowane komunikatami, jednak w tego rodzaju przypadkach konfiguracja następuje w czasie wdrażania, a same liczniki odpowiadają za przetwarzanie wsadowe w określonych odstępach czasu. Usługę licznika czasowego szczegółowo omówimy w rozdziale 13.

Nazewnictwo

Wszystkie usługi nazewnicze odpowiadają za realizację podobnych działań — oferują swoim klientom mechanizmy ułatwiające lokalizowanie obiektów lub zasobów rozproszonych. Efektywna realizacja tego zadania wymaga od usług nazewniczych dwóch rzeczy: wiązania obiektów oraz udostępniania interfejsu API dla operacji wyszukiwania. **Wiązanie obiektów** (ang. *object binding*) polega na przypisywaniu obiektom rozproszonym nazw wyrażonych w języku naturalnym lub identyfikatorów. Przykładowo obiektowi `TravelAgentRemote` można przypisać nazwę *TravelAgentRemote* lub *agent*. Przypisana nazwa bądź identyfikator w praktyce pełni funkcję wskaźnika bądź indeksu określonego obiektu rozproszonego. **Interfejs API**

wyszukiwania oferuje oprogramowaniu klienckiemu dostęp do elementów funkcjonalności stosowanego systemu nazewnictwa. Najkrócej mówiąc, interfejsy wyszukiwania umożliwiają klientom łączenie się z usługami rozproszonymi oraz żądanie zdalnych referencji do potrzebnych obiektów.

Specyfikacja Enterprise JavaBeans wymusza na producentach kontenerów EJB stosowanie interfejsu JNDI w roli API żądania obsługującego wyszukiwania komponentów generowanego przez aplikacje klienckie Javy. Interfejs JNDI obsługuje niemal wszystkie rodzaje usług nazewnictwa i katalogów. Mimo że wielu programistów uważa ten interfejs za nadmiernie skomplikowany, jego wywołania na poziomie aplikacji Javy EE z reguły mają dość prostą formę. Aplikacje klienckie Javy mogą wykorzystywać interfejs JNDI zarówno do inicjowania połączeń z serwerem EJB, jak i do lokalizowania konkretnych komponentów EJB. Przykładowo poniższy fragment kodu demonstruje sposób, w jaki za pomocą interfejsu JNDI API można zlokalizować i uzyskać referencję do komponentu EJB *TravelAgent*:

```
javax.naming.Context jndiContext = new javax.naming.InitialContext( );
Object ref = jndiContext.lookup("TravelAgentRemote");
TravelAgentRemote agent = (TravelAgentRemote)
    PortableRemoteObject.narrow(ref, TravelAgentRemote.class);

Reservation res = agent.bookPassage(...);
```

Właściwości przekazywane za pośrednictwem parametrów konstruktora klasy `InitialContext` sygnalizują interfejsowi JNDI API, gdzie należy szukać serwera EJB i który sterownik JNDI załadować. Metoda `Context.lookup()` określa na potrzeby usługi JNDI nazwę obiektu, który ma zostać zwrócony przez wskazany wcześniej serwer EJB. W tym przypadku interesuje nas interfejs zdalny komponentu EJB *TravelAgent*. Kiedy już będziemy dysponowali wspomnianym interfejsem, będziemy mogli wywoływać metody obsługujące takie operacje jak rezerwowanie miejsc w kajutach.

Istnieje wiele różnych rodzajów usług katalogowych i nazewniczych — producenci kontenerów EJB mogą co prawda swobodnie wybierać te rozwiązania, które w największym stopniu spełniają ich wymagania, jednak wszystkie serwery muszą dodatkowo obsługiwać usługę nazewniczą architektury CORBA.

Bezpieczeństwo

Serwery Enterprise JavaBeans mogą obsługiwać aż trzy rodzaje zabezpieczeń:

Uwierzytelnianie
: Najprościej mówiąc, uwierzytelnianie polega na potwierdzaniu tożsamości danego użytkownika. Najbardziej popularną formą uwierzytelniania jest ekran logowania, w którym użytkownik musi wpisać swoje nazwę i hasło. Użytkownik może korzystać z danego systemu dopiero po akceptacji podanych przez niego danych w systemie uwierzytelniającym. W procesie uwierzytelniania można wykorzystywać także karty identyfikujące, karty z paskiem magnetycznym, certyfikaty bezpieczeństwa i wszelkie inne formy identyfikacji. Chociaż mechanizmy uwierzytelniające mają na celu przede wszystkim zabezpieczanie systemu przed dostępem osób nieuprawnionych, największą niedoskonałością tego rodzaju rozwiązań jest brak kontroli nad dostępem do zasobów systemu przez raz uwierzytelnionego użytkownika.

Autoryzacja
: Autoryzacja (kontrola dostępu) polega na wymuszaniu stosowania określonej polityki bezpieczeństwa określającej, co poszczególni użytkownicy mogą, a czego nie mogą robić w danym systemie. Kontrola dostępu daje nam pewność, że użytkownicy uzyskują dostęp tylko do tych zasobów, które są im rzeczywiście potrzebne. Autoryzacja umożliwia nie tylko ograniczanie dostępu do podsystemów, danych lub obiektów biznesowych, ale też monitorowanie ogólnych zachowań. Przykładowo, niektórzy użytkownicy mogą mieć prawo aktualizacji informacji, podczas gdy pozostali mogą te informacje tylko przeglądać.

Bezpieczna komunikacja
: Kanały komunikacyjne łączące klienta z serwerem dość często muszą gwarantować odpowiedni poziom bezpieczeństwa. Kanał komunikacyjny można zabezpieczyć, np. szyfrując dane przesyłane pomiędzy serwerem a klientem. W takim przypadku wszystkie przesyłane komunikaty są kodowane w sposób uniemożliwiający ich odczytywanie lub modyfikowanie przez nieautoryzowanych użytkowników. Tego rodzaju rozwiązania z reguły wiążą się z koniecznością wymiany pomiędzy klientem a serwerem kluczy kryptograficznych. Klucze umożliwiają uprawnionym odbiorcom komunikatów dekodowanie i odczytywanie ich treści.

Zagadnienia związane z bezpieczeństwem szczegółowo omówimy w rozdziale 17.

Usługi podstawowe i współdziałanie

Możliwość współdziałania jest kluczowym elementem architektury EJB. Specyfikacja Enterprise JavaBeans nie tylko obejmuje konkretne wymagania w kwestii obsługi protokołu Java RMI-IIOP (w roli mechanizmu wywoływania zdalnych metod), ale też definiuje rozwiązania umożliwiające efektywną współpracę w takich obszarach jak przetwarzanie transakcyjne, nazewnictwo i bezpieczeństwo. Specyfikacja EJB wymaga też obsługi technologii JAX-RPC, która sama wymaga obsługi protokołu SOAP 1.1 oraz języka WSDL 1.1, czyli standardów w świecie usług Web Services.

IIOP

Specyfikacja Enterprise JavaBeans nakłada na producentów serwerów EJB obowiązek implementowania standardu Java RMI, który z kolei korzysta z protokołu CORBA 2.3.1 IIOP. Twórcy specyfikacji zdecydowali się na zdefiniowanie tego wymagania z myślą o zapewnieniu możliwości współpracy serwerów aplikacji Javy EE i — tym samym — umożliwieniu komponentom Javy EE (w tym komponentom EJB, aplikacjom, serwletom oraz stronom JSP) pracującym na jednym serwerze Javy EE uzyskiwania dostępu do komponentom EJB działającym na innym serwerze Javy EE. Specyfikacja Java RMI-IIOP definiuje standardy w takich obszarach jak transfer parametrów, zwracanie wartości, generowanie wyjątków oraz odwzorowywanie interfejsów i obiektów wartości w języku CORBA IDL.

Producenci kontenerów EJB mogą oczywiście implementować obsługę innych protokołów niż Java RMI-IIOP, jednak semantyka konstruowanych interfejsów RMI musi pasować do typów obsługiwanych przez protokół RMI-IIOP. Zdecydowano się na to ograniczenie głównie po to, by zapewnić spójność perspektywy klienta EJB niezależnie od stosowanego protokołu zdalnych wywołań.

Możliwość współpracy transakcji zatwierdzanych w trybie dwufazowym i realizowanych w różnych kontenerach jest opcjonalnym, ale bardzo ważnym elementem architektury Enterprise JavaBeans. Takie rozwiązanie daje nam pewność, że transakcje inicjowane przez jeden komponent Javy EE są propagowane do komponentów EJB pracujących w innych kontenerach. Specyfikacja EJB precyzyjnie opisuje zarówno sposób obsługi dwufazowego zatwierdzania obejmującego transakcje realizowane przez wiele kontenerów EJB, jak i sposób współpracy kontenerów transakcyjnych z kontenerami nietransakcyjnymi.

Specyfikacja Enterprise JavaBeans uwzględnia także możliwość współpracy pomiędzy usługami nazewnictwa odpowiedzialnymi za lokalizowanie i odnajdywanie zasobów EJB. Specyfikacja EJB określa, że funkcję łącznika usług nazewniczych pełni moduł `CosNaming` architektury CORBA. Ta sama specyfikacja definiuje zarówno sposób implementowania przez tego rodzaju usługi interfejsów IDL komponentów EJB, jak i sposób korzystania z tych usług przez oprogramowanie klienckie (za pośrednictwem protokołu IIOP).

Specyfikacja Enterprise JavaBeans przewiduje możliwość współpracy w obszarze bezpieczeństwa — określa sposób, w jaki kontenery EJB ustanawiają bezpieczne relacje i wymieniają dane uwierzytelniające w czasie uzyskiwania przez komponenty Java EE dostępu do komponentów EJB wchodzących w skład tych kontenerów. Kontenery EJB muszą obsługiwać protokół SSL 3.0 (ang. *Secure Sockets Layer*) oraz właściwy protokół organizacji IETF — w tym przypadku protokół TLS 1.0 (ang. *Transport Layer Security*) wykorzystywany dla bezpiecznych połączeń pomiędzy klientami i komponentami EJB.

Mimo że historia protokołu IIOP jest dość długa, a sam protokół oferuje możliwość nawiązywania współpracy w rozmaitych obszarach, w praktyce wspomniany protokół nigdy nie odniósł prawdziwego sukcesu rynkowego. Wbrew przewidywaniom swoich twórców protokół IIOP z kilku względów nie zyskał spodziewanej popularności — jego najważniejszą wadą była złożoność. Mimo że IIOP jest protokołem niezależnym od platformy, wielu producentów ma problemy z jego prawidłowym implementowaniem. Co więcej, w IIOP i innych protokołach architektury CORBA znaleziono szereg luk, które mogą powodować poważne problemy we współpracy elementów oprogramowania wdrożonych w docelowych środowiskach. Trudno znaleźć rzeczywiste rozwiązania obejmujące systemy EJB z powodzeniem współpracujące za pośrednictwem protokołu IIOP. Wydaje się, że środowisko programistów dużo większe nadzieje pokłada w standardach SOAP i WSDL jako podstawie dla mechanizmów współpracy tego rodzaju systemów.

SOAP i WSDL

SOAP (ang. *Simple Object Access Protocol*) jest podstawowym protokołem wykorzystywanym przez współczesne usługi Web Services. Protokół SOAP bazuje na języku XML i może być stosowany zarówno w systemach przesyłania komunikatów w technologii RPC, jak i w systemach asynchronicznego przesyłania dokumentów. W praktyce właśnie związki protokołu SOAP z językiem XML decydują o prostocie implementacji korzystających z tego protokołu. Każda platforma (system operacyjny, język programowania, aplikacja itp.), która oferuje możliwość nawiązywania połączeń HTTP i wykonywania analizy składniowej kodu XML-a, może z powodzeniem obsługiwać protokół SOAP. Właśnie dlatego protokół SOAP w tak krótkim czasie zyskał dość szeroką akceptację. Współcześni programiści mają do dyspozycji ponad siedemdziesiąt zestawów narzędzi (bibliotek kodu) związanych z obsługą protokołu SOAP i przystosowanych do pracy w niemal wszystkich środowiskach programowania, włącznie z językiem Java, .NET, JavaScript, C, C++, Visual Basic, Delphi, Perl, Python, Ruby, Smalltalk i innymi.

WSDL (ang. *Web Service Description Language*) jest językiem definiowania interfejsów (IDL) dla usług Web Services. Pojedynczy dokument języka WSDL ma postać pliku XML opisującego zarówno usługi Web Services obsługiwane przez dane przedsiębiorstwo, jak i protokoły, formaty komunikatów i adresy sieciowe tych usług. Dokumenty WSDL charakteryzują się ścisłą strukturą i jako takie mogą być wykorzystywane w procesie automatycznego generowania namiastek technologii RPC i innych interfejsów programowych dla komunikacji z usługami Web Services. Mimo że dokumenty WSDL mogą opisywać usługi dowolnego typu, z reguły są wykorzystywane do opisywania usług Web Services stosujących protokół SOAP.

Język WSDL i protokół SOAP bardzo często są stosowane łącznie. Stanowią bloki składające się na szersze standardy odpowiedzialne za zapewnianie współpracy w takich obszarach jak bezpieczeństwo, przetwarzanie transakcyjne, koordynację, przesyłanie komunikatów i wielu, wielu innych. Działania różnych grup zaangażowanych w wytwarzanie protokołów infrastrukturalnych na bazie protokołu SOAP i języka WSDL w wielu aspektach się pokrywają, stąd mnóstwo sprzecznych ze sobą i niedojrzałych standardów. Z protokołem SOAP i językiem WSDL wiązano wielkie nadzieje, jednak na tym etapie trudno ostatecznie wyrokować, czy skutecznie wyeliminują wszelkie problemy związane ze współpracą usług Web Services, czyli prawdziwą zmorę twórców systemów korporacyjnych. SOAP, WSDL i protokoły infrastrukturalne zbudowane na bazie tych standardów najprawdopodobniej pójdą dalej niż IIOP, DCOM i inne technologie, co nie oznacza, że będą rozwiązaniami przełomowymi. Same usługi Web Services zostaną szczegółowo omówione w rozdziałach 18. i 19.

Co dalej?

Pierwsze trzy rozdziały tej książki miały na celu wprowadzenie podstawowej wiedzy niezbędnej do wytwarzania komponentów i aplikacji Enterprise JavaBeans. Chociaż wciąż nie przystąpiliśmy do analizy szczegółów, zademonstrowaliśmy większość spośród zagadnień, z którymi będziemy mieli do czynienia w kolejnych rozdziałach. Począwszy od rozdziału 4. będziemy krok po kroku opracowywali własne komponenty i zapoznawali się z możliwościami ich praktycznego stosowania w aplikacjach EJB.