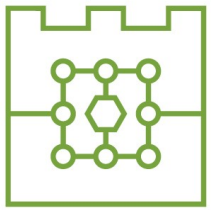
Zaawansowane techniki programowania

Rhuan Rocha - Java EE 8 Design Patterns and Best Practices(2020, Packt)

Omówienie czwartego i piątego rozdziału



Politechnika Krakowska
Wydział Informatyki
i Telekomunikacji

Integration patterns

Koncept warstwy Integracji

JAVA EE podzielona jest na trzy warstwy: prezentacji, biznesową oraz warstwę integracji. Wszystkie trzy współpracują ze sobą w ramach realizacji aplikacji trójwarstwowej.

Patrząc na aplikację twórca musi się zastanowić w jaki sposób odczytywać i zapisywać dane oraz jak jej poszczególne elementy będą się ze sobą komunikować. Aby rozdzielić te zagadnienia od warstwy biznesowej stworzono warstwę integracji.

Warstwa integracji jest odpowiedzialna za rozdzielenie logiki warstwy biznesowej od logiki trwałości danych w obrębie całej aplikacji. Ta warstwa obejmuje funkcjonalność zajmującą się komunikacją z zewnętrznymi zasobami jak np. baza danych, lub system plików oraz umożliwia zapisywanie i odczytywanie danych z tych źródeł co znacznie ułatwia komunikację między aplikacjami i komponentami.

Dodatkowo warstwa integracji ukrywa całą komunikację przed warstwą biznesową która po prostu otrzymuje dane nie dostając informacji o poziomie złożoności komunikacji pomiędzy komponentami oraz w jaki sposób są one zbudowane.

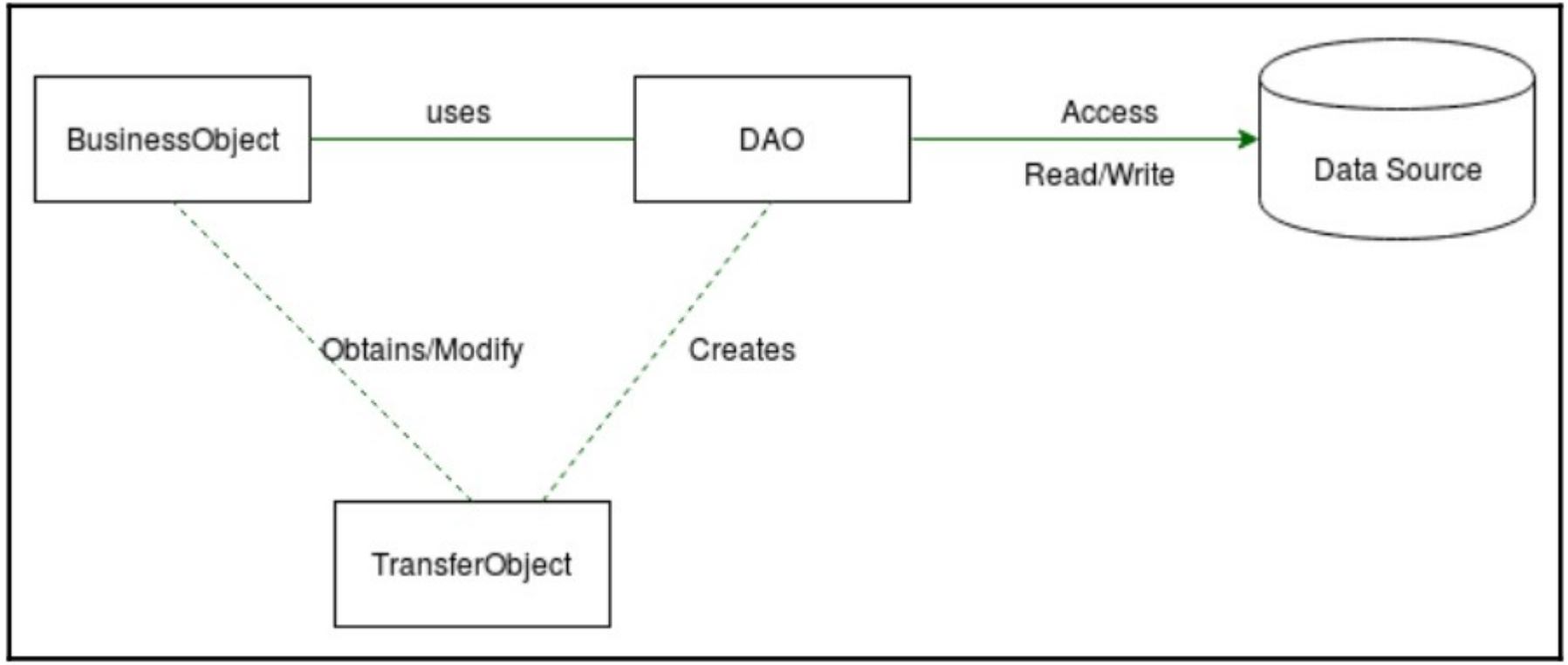
Data-Access Object pattern

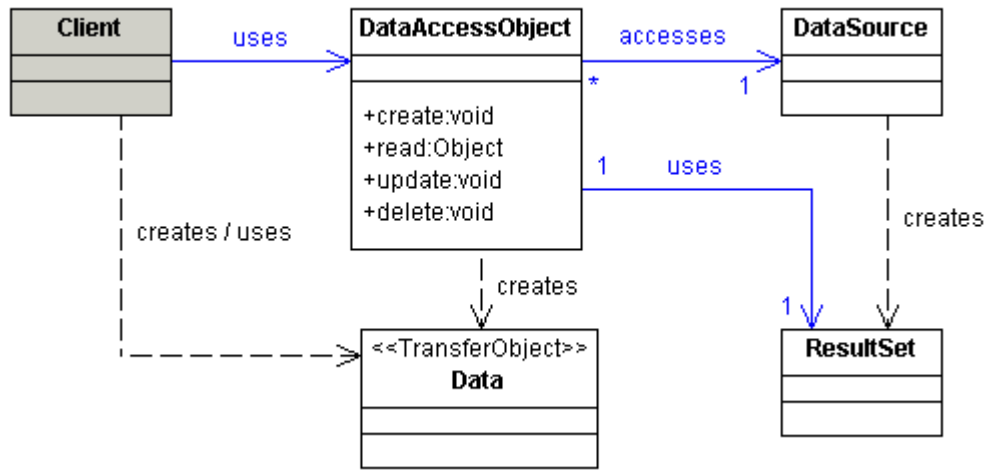
Obecnie aplikacje niemal zawsze muszą być integrowane z jakimś źródłem danych aby odczytywać, zapisywać usuwać i aktualizować dane. Jednym z takich źródeł może być baza danych lub system plików. Każde z zewnętrznych źródeł danych ma swoją strukturę i złożoność która nie powinna być widoczna dla logiki biznesowej.

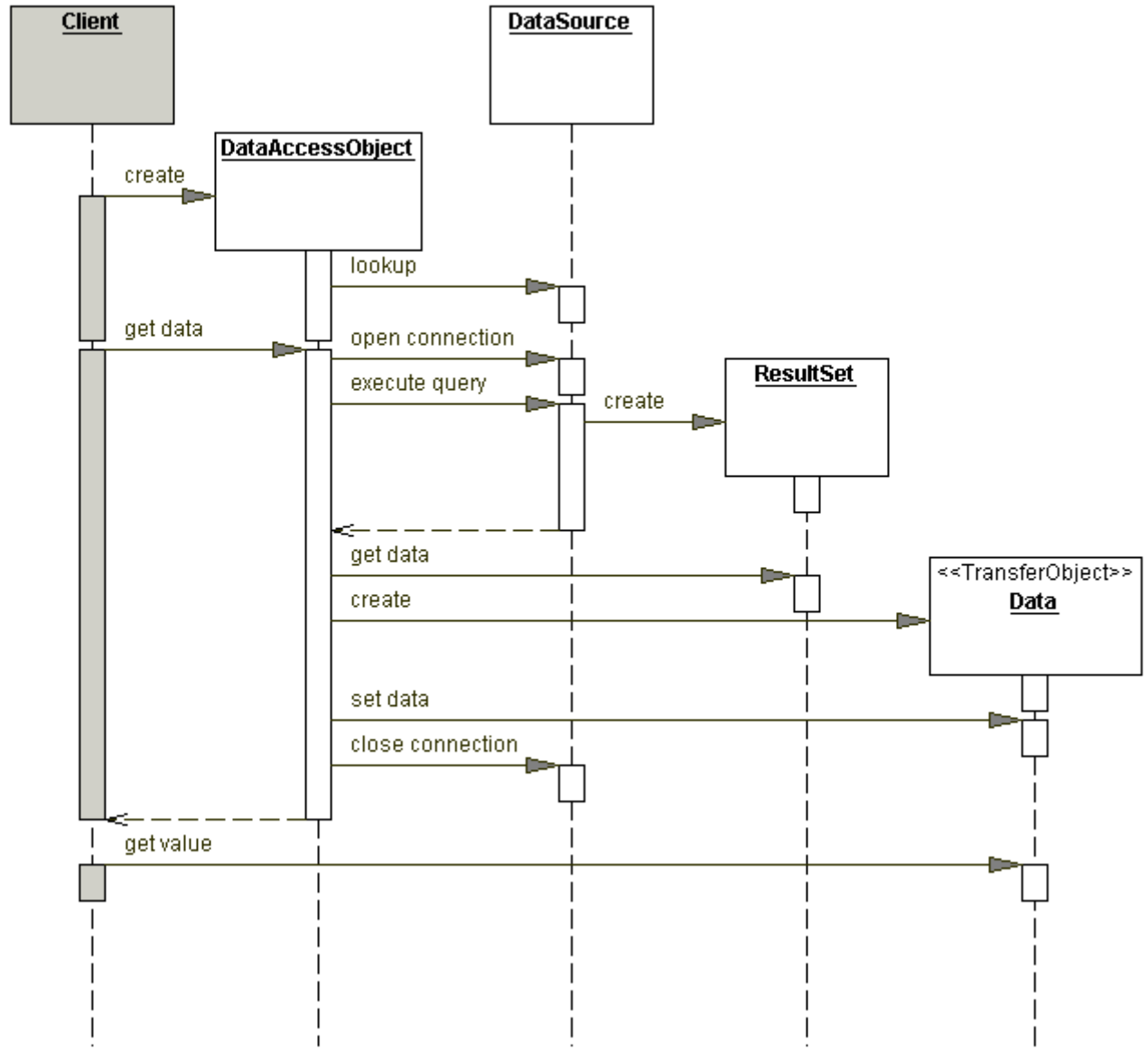
Dlatego powstał Data-Access Object pattern (DAO), który jest wzorcem służącym do ukrywania źródeł danych przed logiką biznesową. Wzorzec ten hermetyzuje dostęp do danych i tym samym oddziela logikę integracji od warstwy Biznesowej. Ponadto programista może modyfikować kod tego wzorca i ta modyfikacja nie będzie widoczna w warstwie Biznesowej. Innymi słowy DAO dostarcza jednolity interfejs służący do komunikacji między aplikacją a źródłem danych.

Wykorzystujemy DAO gdy:

- Chcemy oddzielić logikę trwałości danych od reszty aplikacji.
- Chcemy zapewnić jednolity interfejs API dostępu do różnych typów danych takich jak repozytoria XML, pliki proste, relacyjne i nierelacyjne bazy danych.
- Chcemy uporządkować logikę dostępu do danych i zapewnić hermetyzację, aby ułatwić konserwację i przenośność.


BusinessObject
uses
DAO
Access
Read/Write
Data Source
Obtains/Modify
Creates
TransferObject


Client
uses
DataAccessObject
accesses
DataSource
*
1
+create:void
+read:Object
+update:void
+delete:void
1
uses
creates / uses
creates
creates
1
<<TransferObject>>
Data
ResultSet


Client
DataSource
DataAccessObject
create
lookup
get data
open connection
execute query
ResultSet
create
get data
<<TransferObject>>
Data
create
set data
close connection
get value

Zalety DAO

- Dostarcza prostą metodę separacji integracji i logiki biznesowej w aplikacji umożliwiającą modyfikację jednej i drugiej bez ingerowania w siebie nawzajem.
- Wszystkie szczegóły realizacji komunikacji są ukryte przed resztą aplikacji.
- Można je zaimplementować na wiele różnych sposobów od prostych interfejsów po całe niezależne framework'i.

Wady DAO

- Potencjalna duplikacja kodu. Jeśli aplikacja wymaga kilku klas DAO istnieje szansa że w każdej z nich będzie identyczny kod zawierający implementację CRUD. Można temu zapobiec implementując ogólne DAO zajmujące się tymi operacjami.

Domain-store pattern

DAO umożliwia rozdzielenie logiki dostępu do danych od logiki biznesowej, lecz jest wzorcem bezstanowym. Wiele problemów zawiera skomplikowaną relację między danymi która potrzebna jest do przeprowadzenia procesu zapisu. Aby rozwiązać ten problem powstał wzorzec Domain-Store.

Domain-Store to wzorzec który oddziela logikę trwałości od modelu obiektowego co sprawia że aplikacja może wybrać mechanizm trwałości w zależności od stanu obiektu. To rozwiązanie jest zarówno transparentne jak i oddzielone od logiki biznesowej.

Wewnątrz wzorca Domain-Store znajduje się DAO ukryte przed aplikacją stworzone do komunikacji i manipulowania danymi wewnątrz źródła danych. Dodatkowo Domain-Store pozwala na zapewnienie trwałości obiektom biznesowym bez korzystania z kontenera EJB.

Implementacja Domain-Store może być zrealizowana na dwa sposoby. Pierwszy polega na przygotowaniu prostych klas Java z własną strategią trwałości. Takie rozwiązanie nadaje się tylko do prostych i niewielkich modeli obiektowych gdyż przygotowanie efektywnego, wydajnego i pozbawionego błędów mechanizmu trwałości jest zadaniem bardzo trudnym i czasochłonnym. Drugi sposób to wykorzystanie prostych klas Java wspomaganych przez gotowe już systemy trwałości takie jak np. Hibernate lub JDO(Java Data Objects) i ten sposób jest najczęściej wybierany.

Wykorzystujemy wzorzec Domain-store gdy:

- Nie chcemy umieszczać logiki trwałości w obiektach biznesowych.
- Nie chcemy używać Entity Beans.
- Nasza aplikacja może w całości być umieszczona w kontenerze WEB.
- Struktura klas wykorzystuje dziedziczenie, a klasy związane są ze sobą skomplikowanymi relacjami.

Diagram klas

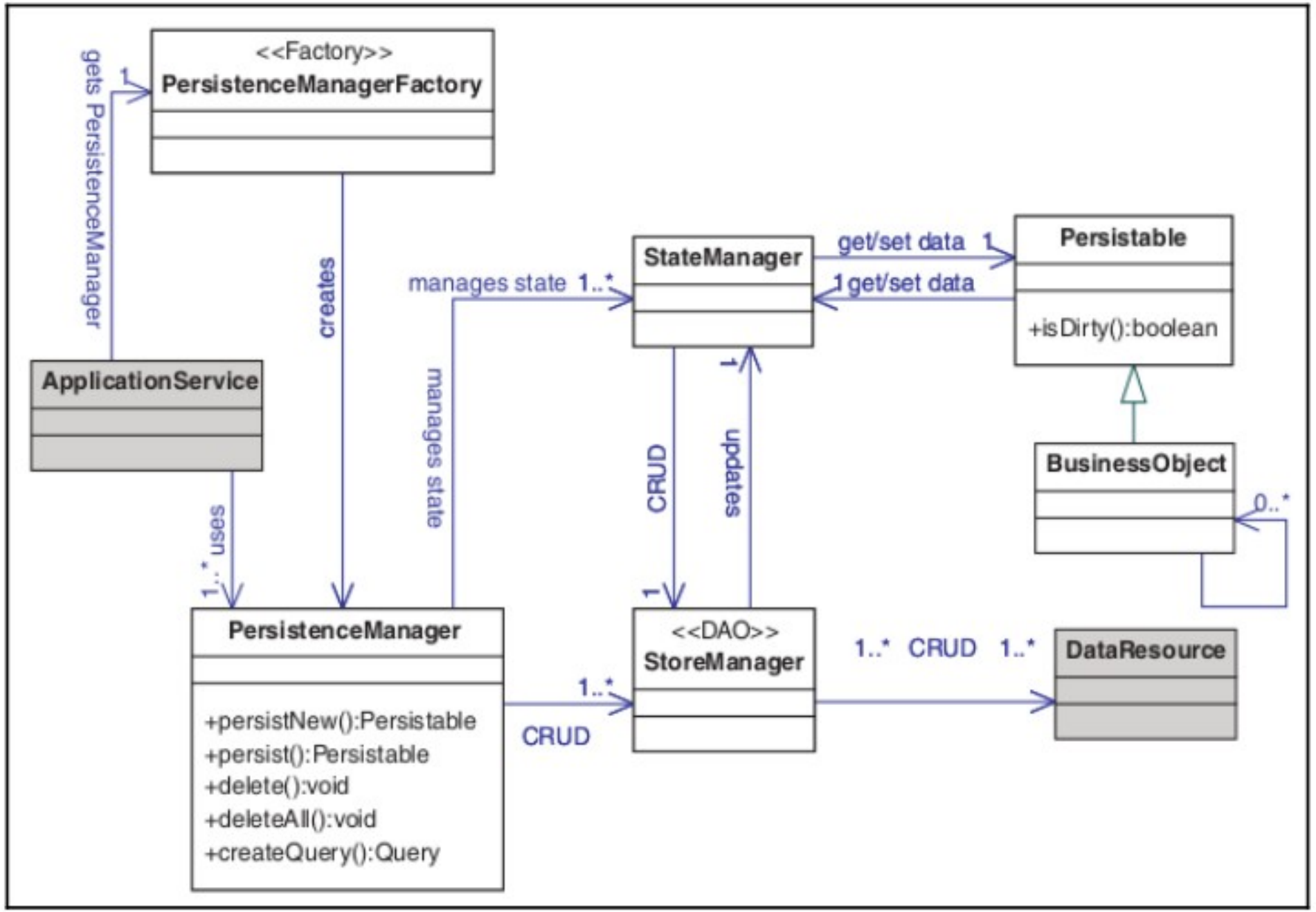
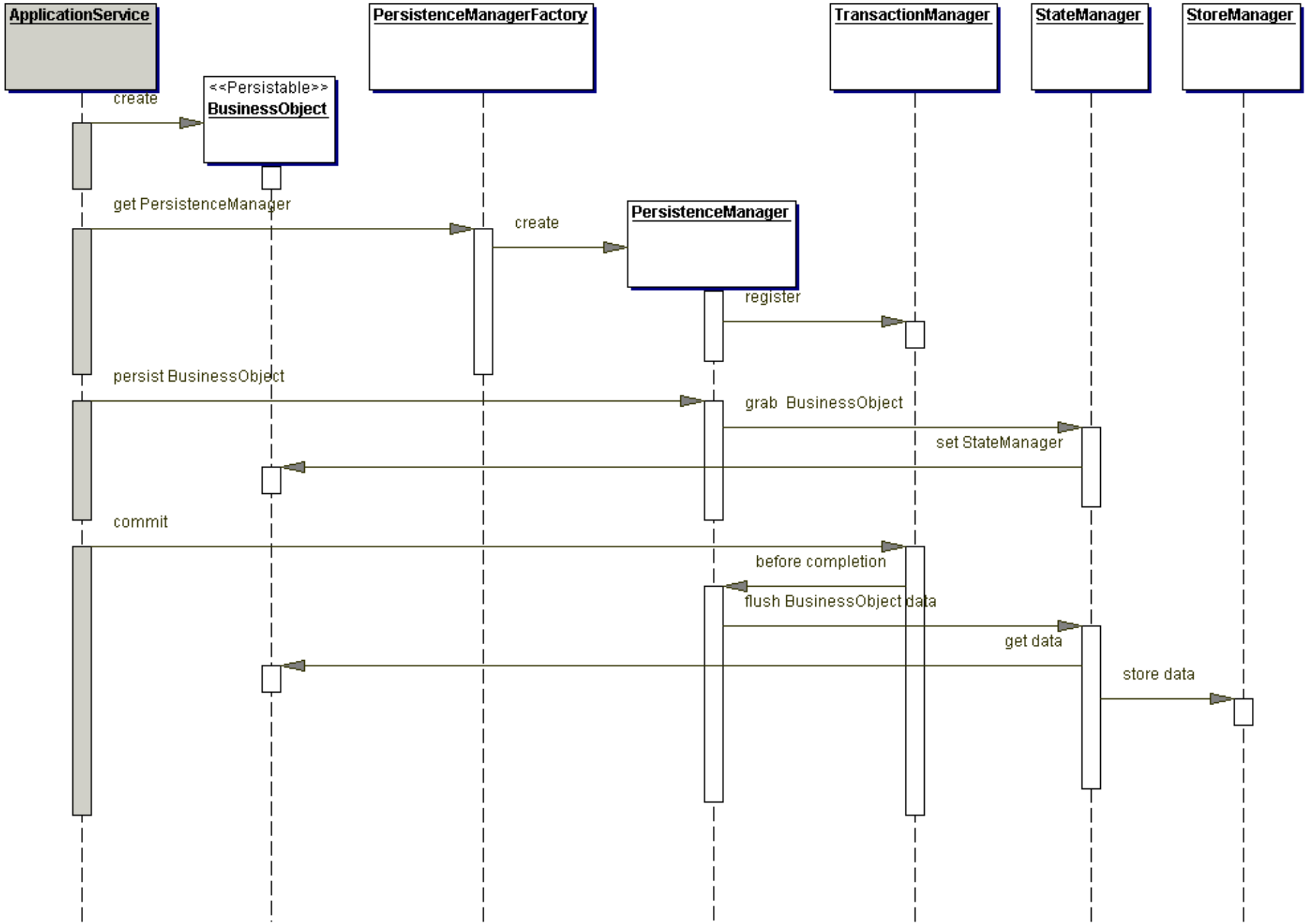


Diagram sekwencji



Zalety Domain-Store

- Rozdziela logikę trwałości od logiki biznesowej.
- Brak implementacji mechanizmów trwałości w obiektach biznesowych.
- Aplikacja w pełni działająca w kontenerze Web (Java EE).
- Można wykorzystać gotowe systemy trwałości jak np. Hibernate.

Wady

- Implementacja własnego systemu trwałości jest czasochłonnym i skomplikowanym zajęciem.
- Długa i skomplikowana struktura implementacji.
- Częściowo wyparte przez JPA (Java Persistance API), obecnie wykorzystywane głównie do źródeł danych nie będących relacyjnymi bazami danych.

Czym jest wzorzec service-activator?

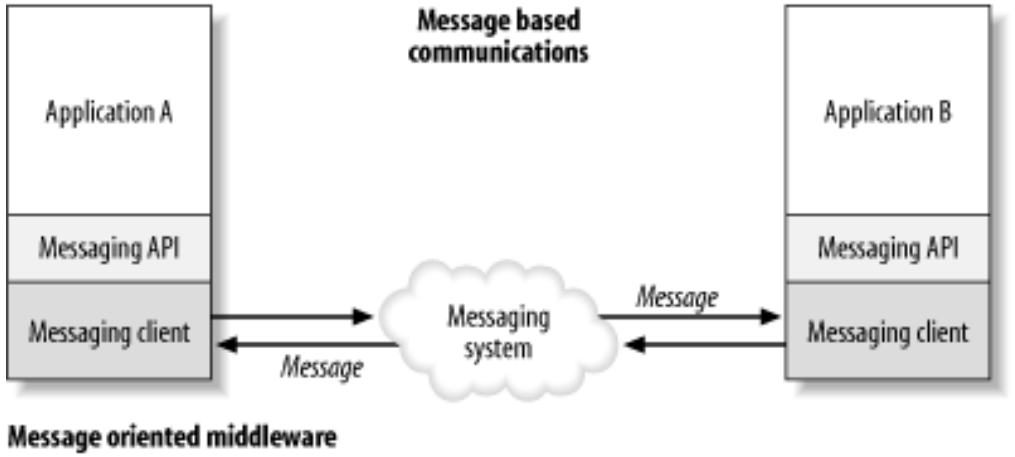
- W aplikacjach korporacyjnych większość przetwarzania odbywa się w sposób synchroniczny. Klient wywołuje usługę biznesową i czeka, aż usługa biznesowa powróci z przetwarzania. Jednak przetwarzanie biznesowe w niektórych przypadkach wymaga znacznej ilości czasu i zasobów. W przypadku tych długotrwałych procesów nie jest możliwe, aby klienci aplikacji czekali na zakończenie przetwarzania biznesowego, więc w tym przypadku chcemy wywoływać usługi asynchronicznie.

Czym jest wzorzec service-activator?

- Niektóre złożone przypadki użycia zajmują dużo czasu. Zamiast blokować użytkowników, możemy uruchomić je asynchronicznie. JMS jest dobrym przykładem wzorca aktywatora usług — JMS (Java Message Service) to interfejs API, który zapewnia możliwość tworzenia, wysyłania i czytania wiadomości. Zapewnia luźno połączoną, niezawodną i asynchroniczną komunikację.
- Aktywator usług jest zaimplementowany jako usługa nasłuchiwania i delegowania JMS, która może nasłuchiwać i odbierać komunikaty JMS.

Message-Oriented Middleware (MOM)

- Message-Oriented Middleware (MOM) to infrastruktura programowa lub sprzętowa wspierająca wysyłanie i odbieranie komunikatów między systemami rozproszonymi. MOM umożliwia dystrybucję modułów aplikacji na heterogenicznych platformach i zmniejsza złożoność tworzenia aplikacji obejmujących wiele systemów operacyjnych i protokołów sieciowych. Oprogramowanie pośredniczące tworzy rozproszoną warstwę komunikacji, która izoluje programistę aplikacji od szczegółów różnych systemów operacyjnych i interfejsów sieciowych. Interfejsy API, które obejmują różne platformy i sieci, są zazwyczaj obsługiwane przez MOM.

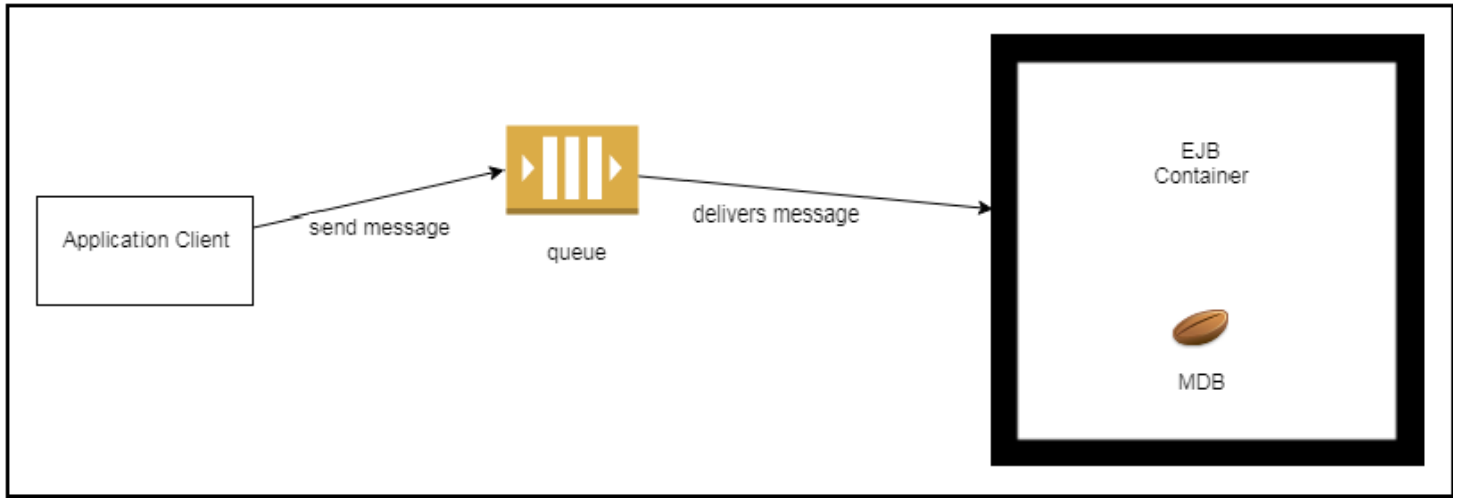


Java Message Service (JMS)

- Java Message Service (JMS) to interfejs programowania aplikacji (API), który zapewnia interfejs MOM dla klientów, którzy chcą wykonać zadanie asynchronicznie. JMS stał się częścią EJB w specyfikacji EJB 2.0 i wprowadzono nowy moduł sesyjny: message-driven bean (MDB)

Message-Driven Bean (MDB)

- Ziarna MDB to bezstanowe ziarno sesji, które służy do nasłuchiwania nadchodzących żądań lub obiektów w kolejce JMS. Należy pamiętać, że MDB może zaimplementować dowolny rodzaj wiadomości, ale jest częściej używany do obsługi komunikatów JMS.
- Komponent bean sterowany komunikatami nasłuchuje komunikatów, które zostały wysłane do kolejki lub do tematu.



Enterprise JavaBeans (EJB)

- Enterprise JavaBeans (EJB) to technologia działająca po stronie serwera, która jest jednym z elementów specyfikacji JEE. EJB jest podobny do podzbioru możliwości JEE w kontekście zarządzania ziarnami udostępniającymi usługi takie jak transakcyjność, trwałość, rozproszenie, bezpieczeństwo, wielodostęp itp. Jedyną rzeczą wymaganą od programisty korzystającego ze specyfikacji EJB jest dostosowanie się do pewnego interfejsu EJB (wymogów implementacyjnych), którego zastosowanie zwalnia użytkownika EJB (dostawcy ziarna lub całego modułu ziaren EJB) z konieczności opracowywania własnych metod obsługi komponentów.

Enterprise JavaBeans (EJB)

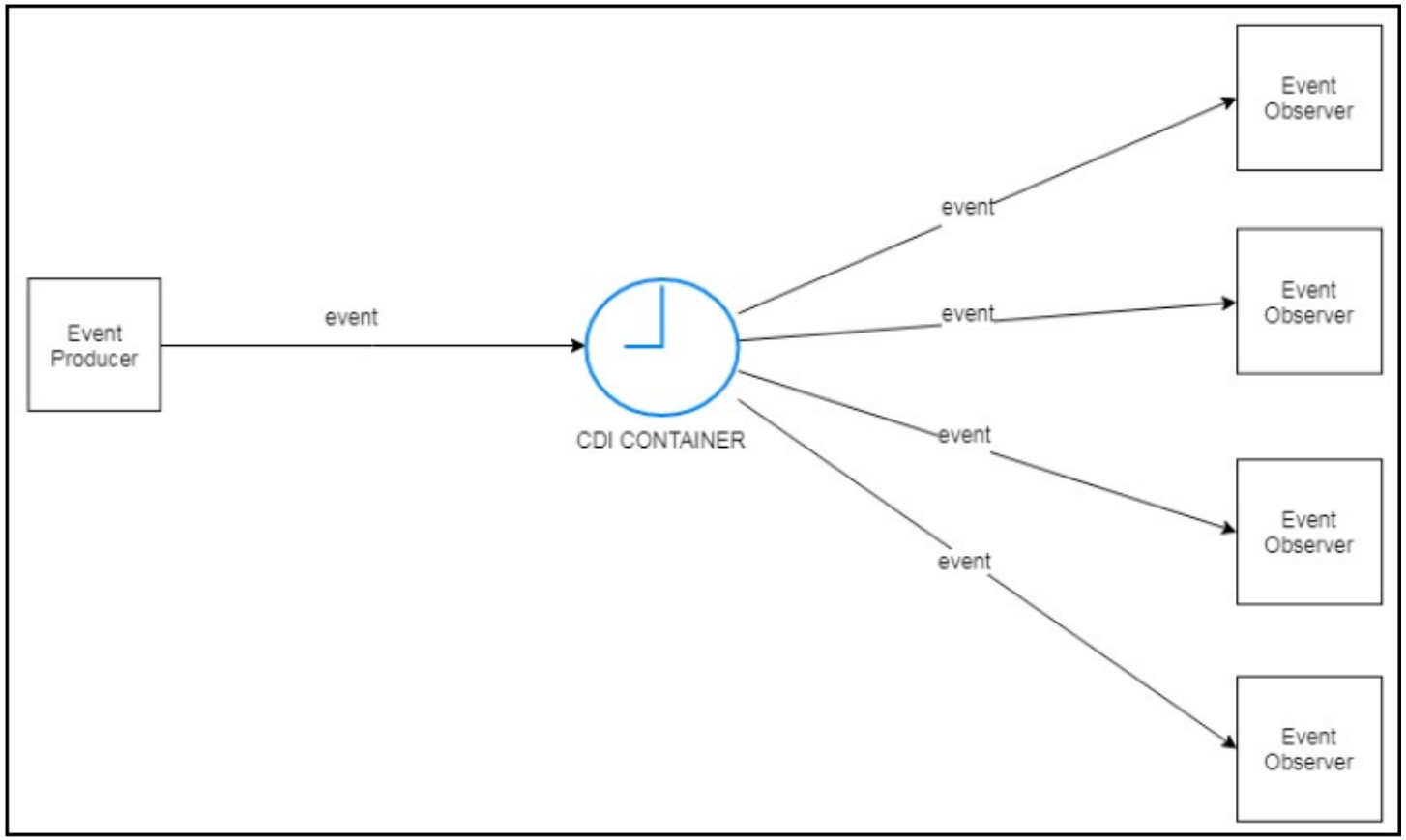
- Idea EJB opiera się na tworzeniu komponentów (ziaren EJB), które mogą być osadzane na serwerze aplikacji (tzw. kontenerze EJB), który z kolei udostępnia je do wykonania lokalnie (dostęp z części aplikacji uruchomionej na tej samej wirtualnej maszynie) lub zdalnie poprzez protokół RMI over IIOP.
- Wyróżnia się trzy główne rodzaje ziaren EJB:
  - sesyjne EJB (ang. session EJB) bezstanowe i stanowe,
  - sterowane komunikatami EJB (ang. message-driven EJB),
  - encyjne EJB (ang. entity EJB)

Asynchroniczne zdarzenia – producenci i konsumenci

- Inną alternatywą, która pojawiła się na ewolucyjnej skali platformy JEE, był mechanizm zdarzeń, który jest częścią specyfikacji CDI. Mechanizm składa się z producentów i konsumentów wydarzeń, co oznacza, że jeden składnik uruchamia wydarzenie, a inny komponent aplikacji odbiera zdarzenie, działając jako słuchacz lub obserwator.
- Do specyfikacji JEE8 ten mechanizm zdarzeń był wykonywany synchronicznie. Z wprowadzeniem CDI 2.0 w specyfikacji JEE8, zawarto takie usprawanienia w API eventów jak użycie asynchroniczne.

Asynchroniczne zdarzenia – producenci i konsumenci

Poniższy schemat przedstawia asynchroniczny mechanizm wysyłania i odbierania zdarzeń:



Zdarzenia asynchroniczne - przykład

- Załóżmy, że klient chce wysyłać e-maile do studentów, zapraszając ich do seminarium technologicznego. Załóżmy, że wysłane e-maile generują informacje statystyczne do późniejszej analizy.
- W naszym przypadku klient nie musi czekać na natychmiastową odpowiedź. Możemy stworzyć producenta zdarzenia i dwóch obserwatorów zdarzenia:
  - Obserwatora odpowiedzialnego za wysyłkę samej wiadomości e-mail
  - Obserwatora odpowiedzialnego za kontrolę statystyczną.
- Te dwa procesy są od siebie niezależne.

Programowanie aspektowe (Aspect-Oriented Programing,AOP)

Czym jest programowanie aspektowe?

- Programowanie aspektowe to paradygmat tworzenia programów, który pozwala na rozdzielenie logiki biznesowej od kodu właściwego (modularyzacja kodu).
- Zapobiegnięcie negatywnym skutkom tworzenia „warkocza”
- Oddzielenie od siebie części kodu które nie są związane ze sobą funkcjonalnie i umieszczenie w osobnych aspektach.
- Punkty interakcji między aspektami

Zastosowanie

- Logowanie, profilowanie
- Autoryzacje i uwierzytelnianie
- Zarządzanie transakcjami
- Konwersja obrazów
- Kompresja danych

Interceptor – metoda przechwytująca

- Interceptory w programowaniu aspektowym odpowiadają za dekorowanie klasy dodatkowymi funkcjami bez ingerencji w metodę.
- Umożliwiają wykonywanie pewnych czynności przed lub po wykonaniu metody
- Służą również do przechwytywania wywołań HTTP oraz ich modyfikacji
- Nie wymagają interfejsów, a jedynie deklaracji metody z argumentem obiektu typu:

javax.interceptor.InvocationContext

CDI – Contex and Dependency Injection

- CDI jest standardem wstrzykiwania zależności w Javie EE
- Definiuje również standard programowania aspektowego (mechanizm interceptorów)
- Wstrzykiwanie zależności jest techniką opierającą się o unikanie silnych powiązań między klasami. Ważną rolą odgrywa korzystanie z interfejsów oraz polimorfizmu
- Podstawową ideą jest odwrócenie sterowania obiektami.
- Zamiana relacji między obiektami i przekazanie pracy „wtryskiwaczowi”

Ziarno - bean

- Ziarnem możemy nazwać obiekt klasy zarządzany po stronie kontenera.
- Ziarna CDI
  - Bezargumentowy konstruktor lub posiadający adnotacje @Inject
  - Konkretna klasa na wysokim poziomie hierarchii lub @Decorate
  - Zdefiniowane jako nie EJB
- Ziarna znajdują się w archiwum ziaren.
- Adnotacja @Vetoed – wykluczenie klasy

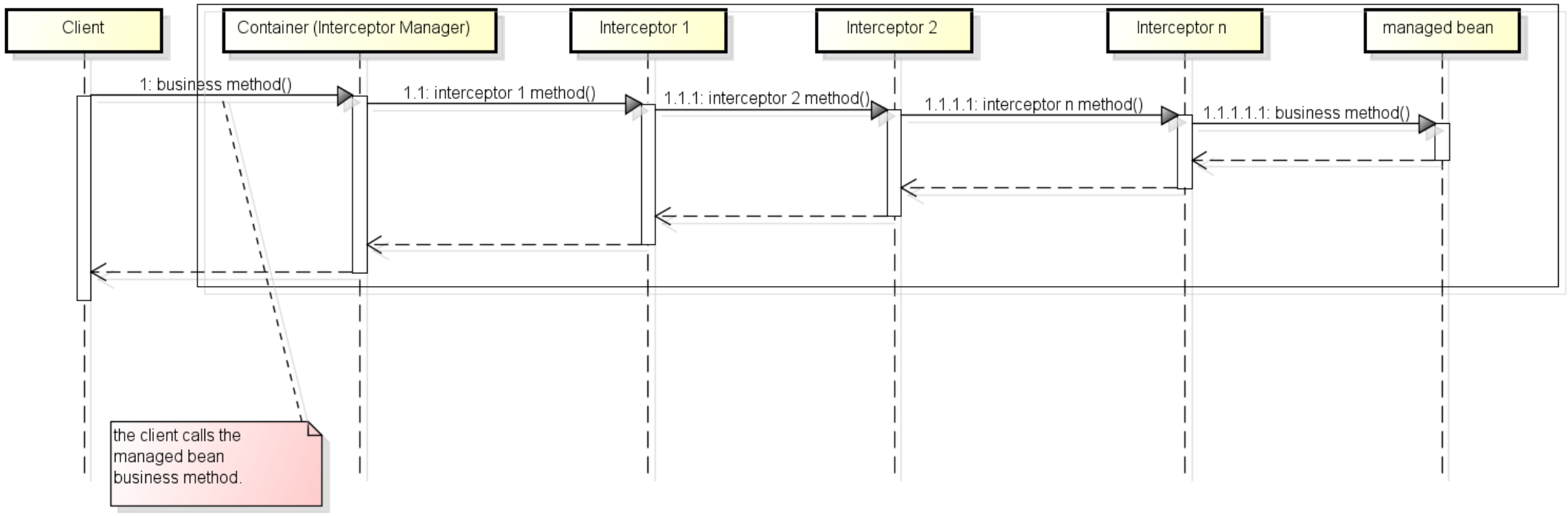
Zakres ziaren CDI

- Cykl życia ziarna CDI jest powiązany z zakresem kontekstowym.
  - @RequestScoped — zakres obejmuje żądanie HTTP użytkownika.
  - @SessionScoped — zakres obejmuje sesję HTTP użytkownika.
  - @ApplicationScoped — stan jest współdzielony przez wszystkich użytkowników w aplikacji.
  - @ConversationScoped — zakres jest kontrolowany przez programistę.

Interceptory cd.

- Klasy interceptorów i metody interceptorów
- Metody przechwytywania są wywoływane gdy metoda klasy posiada adnotację przechwycenia lub gdy metoda wywołania zwrotnego cyklu jest przechwycona
- Prosta aplikacja – interceptory w klasie docelowej
- Złożona aplikacja/Rozrastająca się – interceptory w osobnej klasie
- Przechwytywane elementy – porady
- Porada -> kod przechwytujący
- Punkty interakcji/przecięcia – lokalizacja punktu wywołania kodu

Łańcuch przechwytujący interceptorów



Implementacja interceptora - EJB

- Interceptory w specyfikacji EJB opierają się głównie o metodę biznesową.
- Główną ideą jest to że klasa jest poddana wstrzykiwaniu musi być zarządzana przez kontener EJB
- Dlatego do poprawnego działania potrzebna jest instancja ziarna.
- Dane z ziarna można uzyskać poprzez obiekt *InvocationContext*
- Cykl życia interceptorów jest związany z cyklem przypisanego ziarna
- Klasy interceptorów można przypisać do ziarna lub metod biznesowych za pomocą adnotacji *@Interceptors*

Implementacja interceptora – EJB cd.

- W interceptorach metod biznesowych ważną adnotacja jest *@AroundInvoke*
- *Metody z tą adnotacja są wywoływane na równi metod biznesowych*
- *Pozwala wywołać dowolny składnik z przechwytywanej metody*
- *Posiada dowolny poziom dostępu*
- *Klasa może posiadać tylko jedną metodę @AroundInvoke*
- *Każda z metod interceporta posiada cykl życia opisana eventami:* `AroundConstruct`, `PostConstruct`, i `PreDestroy`

Implementacja interceptora w CDI

- Mechanizm CDI jest nowszy niż EJB przez co bardziej elastyczny w użytkowaniu
- Tak jak EJB używana jest metoda *@AroundInvoke*
- A także cykle życia
- Można przechwytywać metody biznesowe nie tylko do ziaren EJB
- Ziarno CDI wie tylko jaki typ powiązania posiada interceptor
- To co można zrobić z ziarnem EJB można zrobić z CDI, ale nie na odwrót

Dekorator

- **Dekorator** – wzorzec projektowy należący do grupy *wzorców strukturalnych*. Pozwala na dodanie nowej funkcji do istniejących klas dynamicznie podczas działania programu.
- Wzorzec dekoratora polega na opakowaniu oryginalnej klasy w nową klasę „dekorującą”. Zwykle przekazuje się oryginalny obiekt jako parametr konstruktora dekoratora, metody dekoratora wywołują metody oryginalnego obiektu i dodatkowo implementują nową funkcję.

Zalety

- Większa elastyczność niż statyczne dziedziczenie
- Pozwala uniknąć tworzenia przeładowanych funkcjami klas na wysokich poziomach hierarchii

Wady

- Interfejs dekoratora musi być taki sam jak klasy dekorowanej
- Przy użyciu tego wzorca może powstać wiele małych obiektów

Przykład użycia dekoratora

i10

Już od 41 300 zł

i20

Już od 58 800 zł

i30 Hatchback

Już od 67 900 zł

i30 Fastback

Już od 76 400 zł

NOWOŚĆ

i30 Wagon

Już od 70 900 zł

ELANTRA

Już od 74 900 zł

BAYON

Już od 65 700 zł

KONA

Już od 74 900 zł

TUCSON

Już od 99 900 zł

SANTA FE

Już od 231 900 zł

Typ silnika

2WD, Skrzynia manualna
1.2 MPI 5MT
Benzynowy

1.2 MPI 2WD
62 kW (84 KM)
5.2 l/100km

Sprawdź szczegóły

2WD, Skrzynia manualna
1.0 T-GDI 6MT
Benzynowy

1.0 T-GDI 2WD
74 kW (100 KM)
5.2 l/100km

Sprawdź szczegóły

2WD, Skrzynia manualna
1.0 T-GDI 48V 6iMT
Benzynowy

1.0 T-GDI 48V 2WD
74 kW (100 KM)
5 l/100km

Sprawdź szczegóły

Kolor

Wybierz kolor zewnętrzny
i20

Aurora Grey
2100,00 zł

Specjalne

Perłowe

Metaliczne

Dodatki

Classic Plus
Od 61 000,00 zł

Wybrane wyposażenie wersji Classic Plus

- Kolumna kierownicza regulowana w dwóch płaszczyznach
- Lusterka boczne elektrycznie sterowane i podgrzewane
- Klimatyzacja manualna
- Radio z RDS + DAB
- Sterowanie radiem na kierownicy
- System Bluetooth

Pełna lista wyposażenia

Comfort
Od 67 500,00 zł

Wybrane wyposażenie wersji Comfort

- Światła do jazdy dziennej LED
- Felgi aluminiowe 16"
- Fotele przednie podgrzewane
- Kierownica obszyta skórą
- Podgrzewana kierownica
- Cyfrowe zegary z 10,25" kolorowym wyświetlaczem
- Tylne czujniki parkowania
- Kamera cofania z dynamicznymi liniami pomocniczymi
- System multimedialny z 8" ekranem dotykowym i cyfrowym radiem DAB

Pełna lista wyposażenia

Premium
Od 81 900,00 zł

Wybrane wyposażenie wersji Premium

- Reflektory LED z funkcją doświetlania zakrętów
- Światła tylne LED
- Felgi aluminiowe 17"
- Fotele przednie i tylne podgrzewane
- Klimatyzacja automatyczna
- Elektrochromatyczne lusterko wsteczne
- Podążanie na pasie ruchu (LFA)
- Czujnik deszczu
- Ładowarka bezprzewodowa do smartfonów
- Bezkluczykowy dostęp

Pełna lista wyposażenia

UML

Diagram UML dekoratora

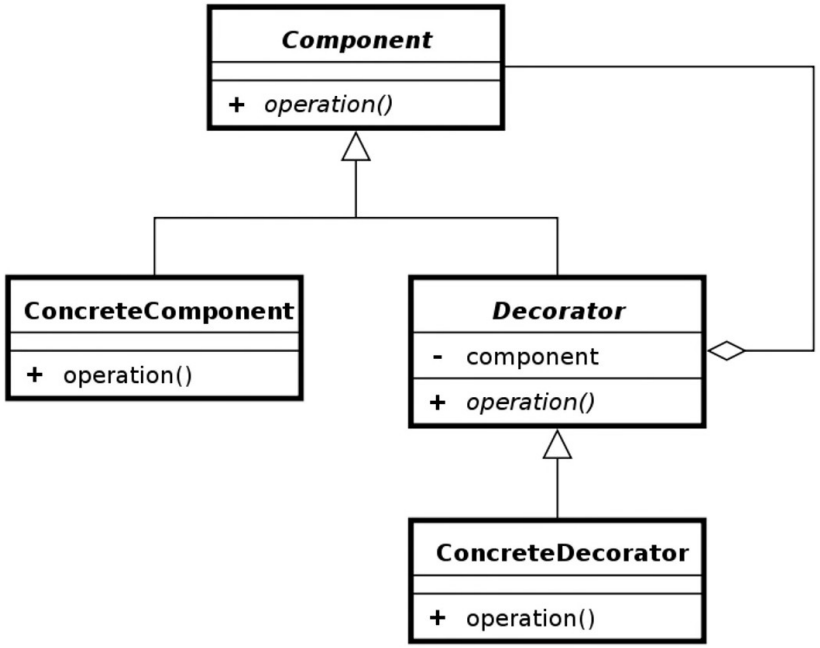
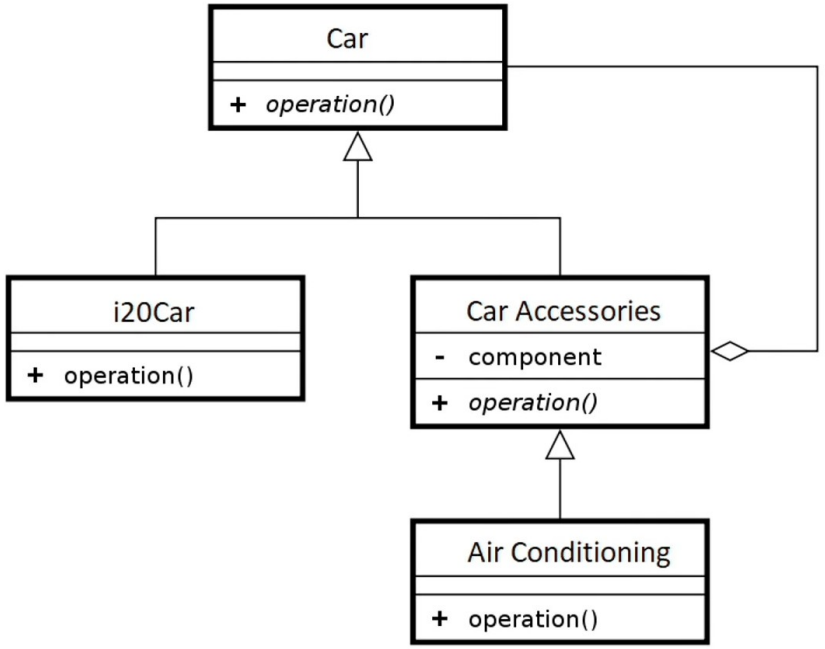


Diagram UML na przykładzie z samochodami



Podsumowanie opracowanych rozdziałów

Podsumowanie rozdziału czwartego

- Czym są warstwy i wzorce integracji.
- Sposób implementacji warstw wzorcowych.
- Jakie wzorce wyróżniamy (Wzorzec obiektu dostępu do danych, magazynu domeny i wzorzec aktywatora usług)
- Wzorzec aktywatora usług - Java Message Service (JMS)
- Metody asynchroniczne - Enterprise Java Beans EJB
- Infrastruktura programowa Message-Oriented Middleware (MOM)
- Bezstanowe ziarno sesji Message-Driven Bean (MDB)
- Asynchroniczne zdarzenia

Podsumowanie rozdziału piątego

- Programowanie aspektowe (Aspect-Oriented Programing(AOP))
- Interceptor - metoda przechwytująca
- Rodzaje interceptorów (Enterprise JavaBeans (EJB), Contex and Dependency Injection (CDI))
- Obiekt klasy zarządzany po stronie kontenera - Ziarno (bean)
- Łańcuch przechwytujący interceptorów
- Dekorator (Decorator design pattern)

Dziękujemy za uwagę!

Kamil Świątek (kamil.swiatek@student.pk.edu.pl)
Łukasz Januszewski (lukasz.januszewski@student.pk.edu.pl)
Michał Wąs (michal.was78@student.pk.edu.pl)
Przemysław Skoczewski (p.skoczewski@student.pk.edu.pl)
Jakub Skoczewski (jakub.Skoczewski@student.pk.edu.pl)