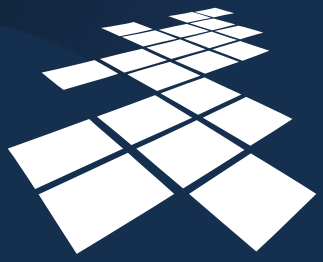
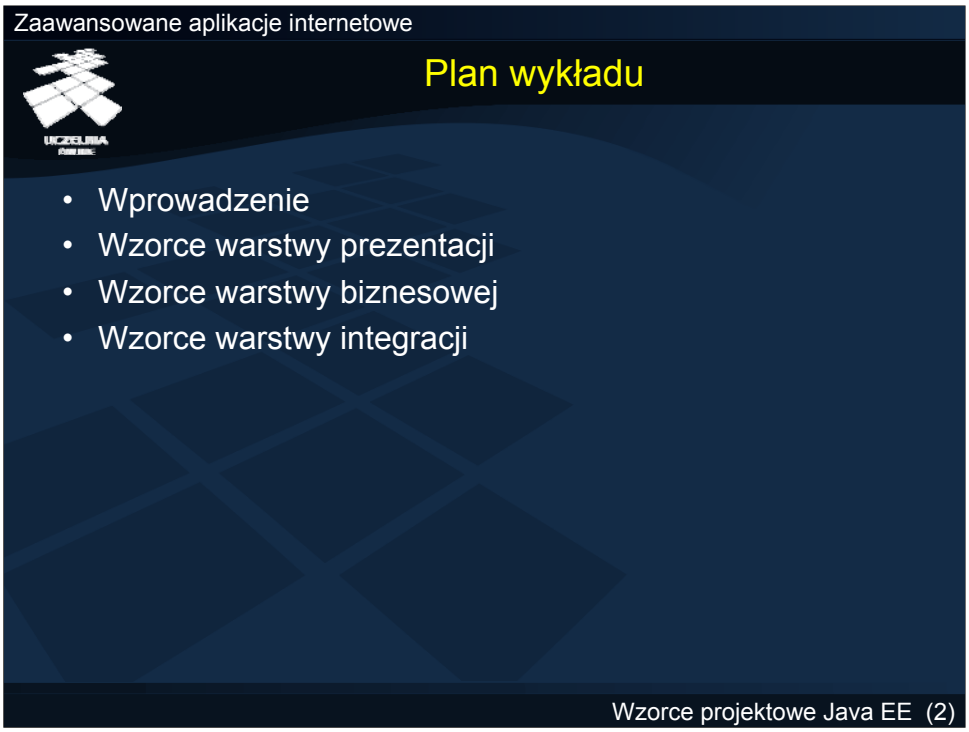
Wzorce projektowe Java EE

wykład prowadzi
Mikołaj Morzy



UCZELNIA
ONLINE



Celem wykładu jest przedstawienie katalogu wzorców projektowych Java EE. W pierwszej części wykładu zaprezentowano pojęcie wzorca, kategoryzację wzorców, zalety stosowania wzorców oraz podstawy podejścia warstwowego, na którym bazuje architektura Java EE, a które jest niezbędne dla poprawnego zrozumienia idei i znaczenia stosowania wzorców projektowych. Kolejne części wykładu stanowią prezentację ponad dwudziestu wzorców projektowych, rozwiązujących problemy najczęściej spotykane w aplikacjach Java EE. Prezentacja została podzielona logicznie według warstw: jako pierwsze prezentowane są wzorce warstwy prezentacji, następnie wzorce warstwy biznesowej, a na koniec wzorce warstwy integracji.

Wzorzec

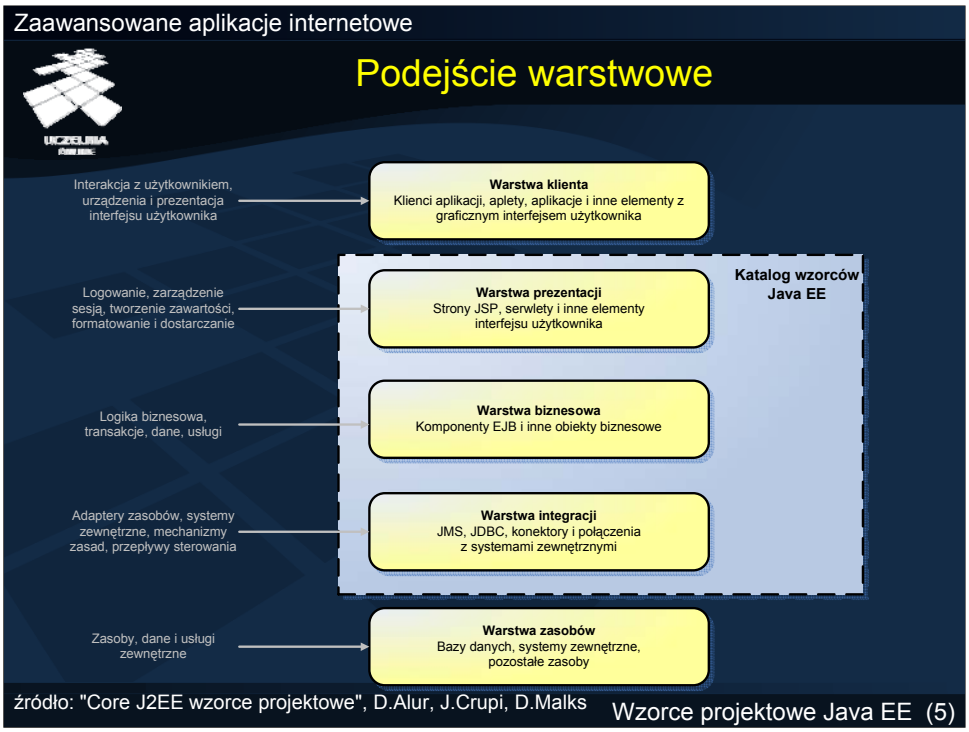
- Każdy wzorzec składa się z trzech części, które wyrażają związek między konkretnym kontekstem, problemem i rozwiązaniem. *Christopher Alexander*
- Wzorzec to pomysł, który okazał się użyteczny w jednym rzeczywistym kontekście i prawdopodobnie będzie także użyteczny w innym. *Martin Fowler*
- Wzorzec a strategia

Pojęcie wzorca występującego w inżynierii i architekturze wprowadził w latach 70-tych XX wieku Ch.Alexander. Na grunt inżynierii oprogramowania ideę wzorców przenieśli w roku 1994 E.Gamma, R.Helm, R.Johnson i J.Vlissides (tak zwana "Banda Czworga", ang. "Gang of Four"). Od tego czasu wzorce projektowe na stałe zagościły w informatyce, a w szczególności w projektowaniu i implementacji aplikacji. Wzorce projektowe nabierają szczególnego znaczenia w kontekście aplikacji internetowych, gdzie wielokrotnie spotykane są te same problemy architektoniczne. Najczęściej wzorzec definiujemy jako trzyczęściową regułę reprezentującą związek między problemem, kontekstem, w którym problem wystąpił, a sprawdzonym rozwiązaniem. Celem wzorca jest wielokrotne użycie pewnego rozwiązania problemu występującego wielokrotnie w tym samym lub podobnym kontekście. Istotne jest rozróżnienie między wzorcem a strategią. Wzorzec to sposób rozwiązania problemu abstrahujący od konkretnej implementacji, natomiast strategia wyraża najczęściej spotykaną implementację wzorca i umożliwia rozszerzanie wzorca. Dla danego wzorca może istnieć wiele równorzędnych strategii implementacji. Mówimy też, że wzorce występują na wyższym poziomie abstrakcji niż strategie.

Cechy wzorca

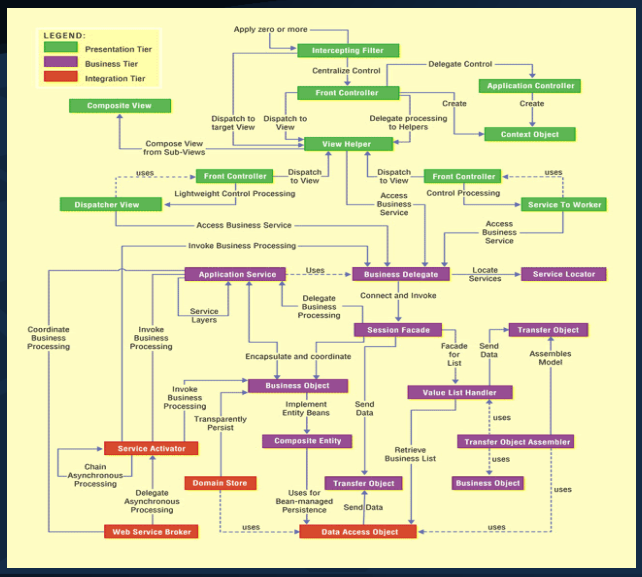
- Odkrywane dzięki doświadczeniu
- Zapisywane w formacie strukturalnym
- Zapobiegają wyważaniu otwartych drzwi
- Wykorzystywane wielokrotnie
- Obrazują najlepsze sprawdzone rozwiązania
- Nieustannie ewoluują
- Mogą być łączone by rozwiązywać bardziej złożone problemy

Wzorce są odkrywane dzięki doświadczeniu. Katalog wzorców Java EE został opracowany przez inżynierów Sun Java Center na podstawie pracy nad wieloma projektami aplikacji internetowych. Wzorce były identyfikowane i dodawane do katalogu po wielokrotnym natknięciu się na ten sam problem, kontekst i rozwiązanie w wielu różnych systemach. Wzorce zapisuje się w formacie szablonów niezależnych od konkretnej implementacji. Wykorzystanie wzorca pozwala uniknąć popularnego błędu: poświęcenia czasu, pieniędzy i zasobów na wyważanie otwartych drzwi (rozwiązywaniu problemów, dla których podano już optymalne sposoby rozwiązania). Kluczem do sukcesu jest niewątpliwie wielokrotne wykorzystanie wzorca w różnych systemach. Wzorce są tak projektowane, aby można było je stosować (po niewielkich przeróbkach) w różnych systemach. Wzorce nie są statyczne. Nieustanna ewolucja zarówno samych wzorców, jak i katalogu wzorców, obrazuje zmiany pojawiające się w aplikacjach internetowych w odpowiedzi na nowe technologie i możliwości. Wreszcie wzorce mogą być łączone w większe całostki w celu rozwiązywania dużych, nietypowych problemów.



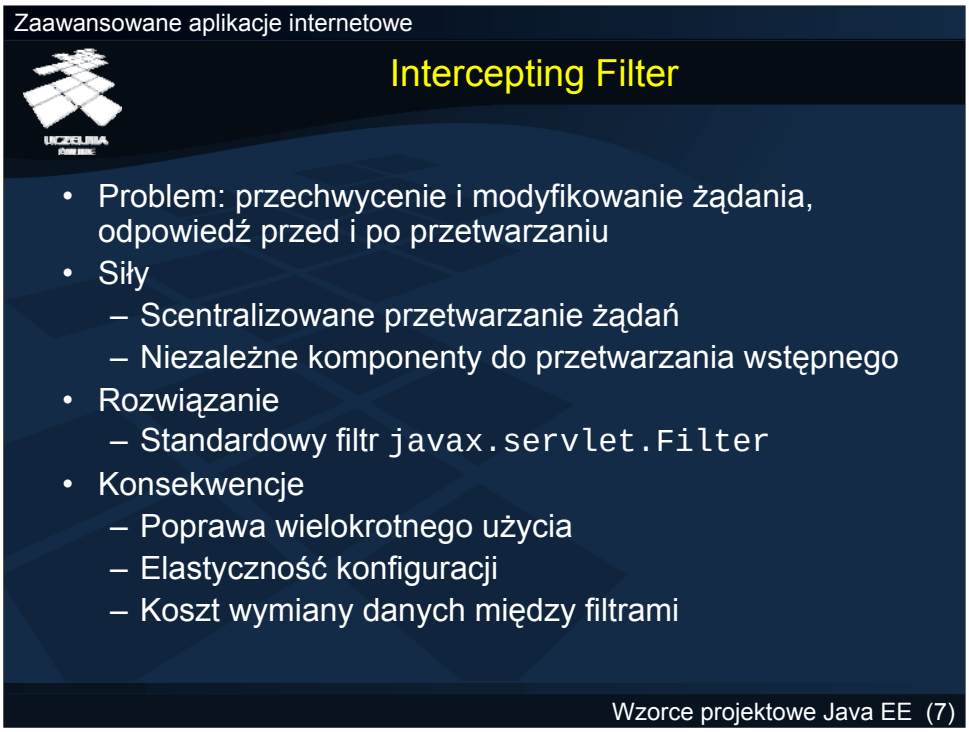
Platforma Java EE jest platformą wielowarstwową. Warstwy odpowiadają logicznemu podziałowi zadań w systemie. Każda warstwa posiada zdefiniowany zakres odpowiedzialności i jest logicznie oddzielona od innych warstw. Stopień integracji między poszczególnymi warstwami jest znikomy a wprowadzenie każdej dodatkowej zależności między warstwami jest uważane za błąd projektowy. Jednym z celów wzorców projektowych jest zachowanie niezależności pomiędzy warstwami. Warstwa klienta reprezentuje urządzenia i aplikacje, które posiadają dostęp do systemu. Najczęściej klientem jest przeglądarka internetowa, aplikacja Java, telefon komórkowy. W warstwie prezentacji zawiera się logika obsługująca: odbieranie żądań klientów, logowanie, zarządzanie sesją, zarządzanie dostępem do usług biznesowych, budową i dostarczeniem odpowiedzi do klienta. W tej warstwie najczęściej wykorzystujemy serwlety i JSP. Warstwa biznesowa udostępnia usługi biznesowe i zawiera logikę obsługi procesów biznesowych i transakcji. Preferowaną implementacją tej warstwy są komponenty EJB. Warstwa integracji odpowiada za komunikację z systemami zewnętrznymi: bazą danych, aplikacją zewnętrzną, monitorem transakcji. W tej warstwie najczęściej korzystamy z komponentów używających JDBC i JCA. W warstwie zasobów umieszczamy zewnętrzne źródła danych, systemy zewnętrzne, systemy B2B, systemy autoryzacji kart kredytowych, itp. Katalog wzorców Java EE obejmuje wzorce przedstawiające rozwiązania dla warstwy prezentacji, biznesowej i integracji.

Mapa wzorców projektowych



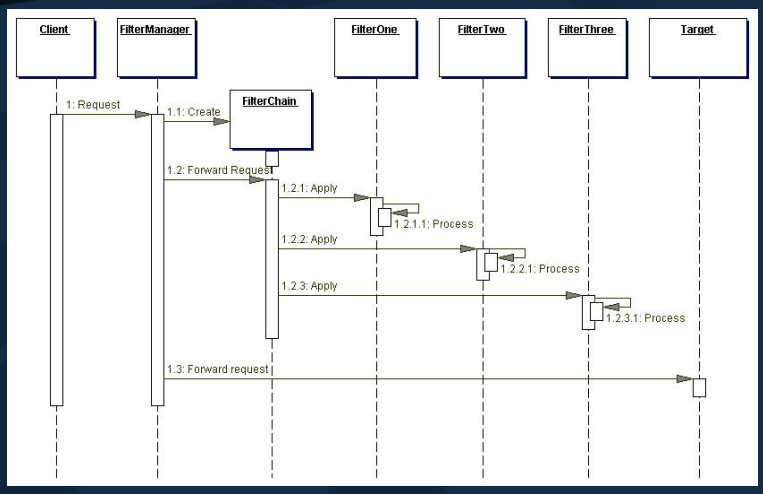
źródło: www.corej2eepatterns.com

Każdy wzorzec projektowy posiada własny problem, kontekst i rozwiązanie. Złożone problemy wymagają jednak złożonych rozwiązań. Żaden wzorzec nie istnieje w izolacji od innych wzorców, lecz wymaga ich współdziałania. Powyższy slajd obrazuje mapę związków występujących między wzorcami. Przykładowo, wzorzec Front Controller centralizuje logikę przetwarzania w warstwie prezentacji, otrzymując wstępnie przygotowane żądania klientów z wzorca Intercepting Filter. Stan aplikacji jest zapisywany przez współpracujący z wzorcem Front Controller wzorzec Context Objext. Front Controller korzysta także z wzorca View Helper do budowania ostatecznego widoku zwracanego do klienta. Współpraca z warstwą biznesową, w zależności od stopnia zaawansowania logiki, odbywa się przez współpracujące wzorce Service to Worker lub Dispatcher View. Jak widać, wzorzec Front Controller wykorzystuje wiele dodatkowych wzorców do skutecznego działania.



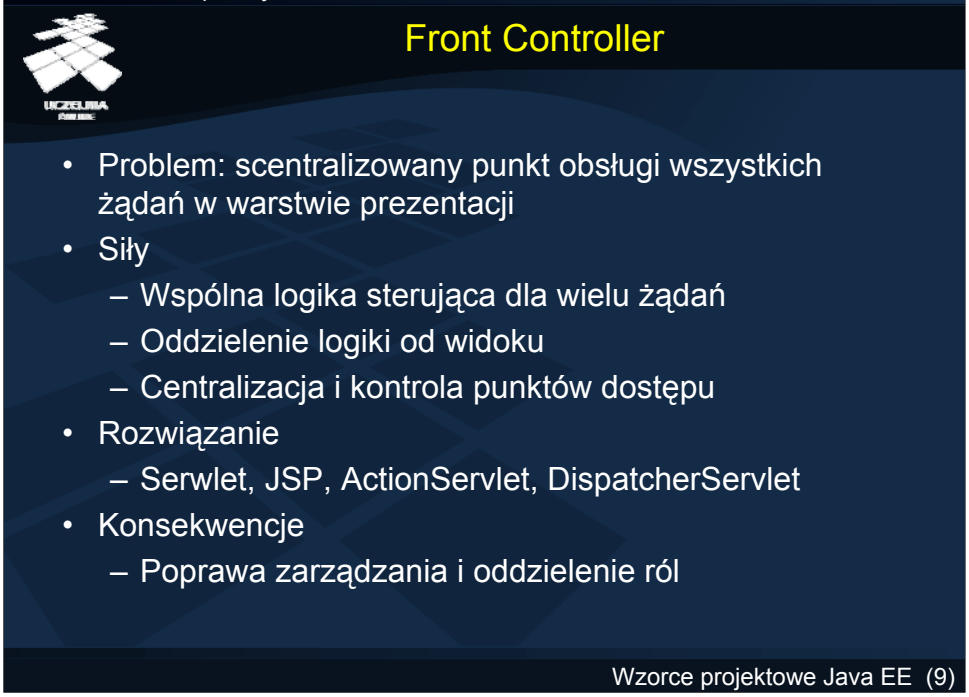
Wzorzec Intercepting Filter rozwiązuje problem przechwytywania i modyfikowania żądań przychodzących od klientów, oraz modyfikacji odpowiedzi zwracanych do klientów. Przykłady możliwych zastosowań to sprawdzenie poprawności sesji klienta, sprawdzenie parametrów klienta HTTP, zarządzanie kodowaniem danych, kompresja/dekompresja danych. Podstawowe zalety stosowania wzorca Intercepting Filter polegają na scentralizowaniu przetwarzania żądań, uniknięciu duplikacji kodu, uniezależnieniu głównego modułu przetwarzania żądań od zadań przetwarzania wstępnego i końcowego, oraz dostarczeniu niezależnych komponentów przetwarzania wstępnego, które mogą być dowolnie konfigurowane, włączane lub wyłączane. Istotne jest, że Intercepting Filter nie zajmuje się przetwarzaniem żądań pod kątem logiki aplikacji. Najczęstszą strategią implementacji jest wykorzystanie mechanizmu filtrów serwletów i implementacja interfejsu javax.servlet.Filter. Konfiguracja dla takiej strategii mieści się w deskryptorze wdrożenia web.xml. Wzorzec Intercepting Filter może też znaleźć zastosowanie poza warstwą prezentacji, np. do obsługi komunikacji z usługami sieciowymi w warstwie integracji i wstępnym przetwarzaniu komunikatów SOAP. Konsekwencje użycia wzorca Intercepting Filter obejmują możliwość wielokrotnego użycia tego samego filtru w różnych miejscach aplikacji i poprawę elastyczności poprzez możliwość deklaratywnego zarządzania filtrami i łańcuchami filtrów przez deskryptor wdrożenia web.xml. Wzorzec Intercepting Filter wiąże się jednak także z dodatkowym narzutem związanym z koniecznością obsługi komunikacji między poszczególnymi filtrami.

Intercepting Filter



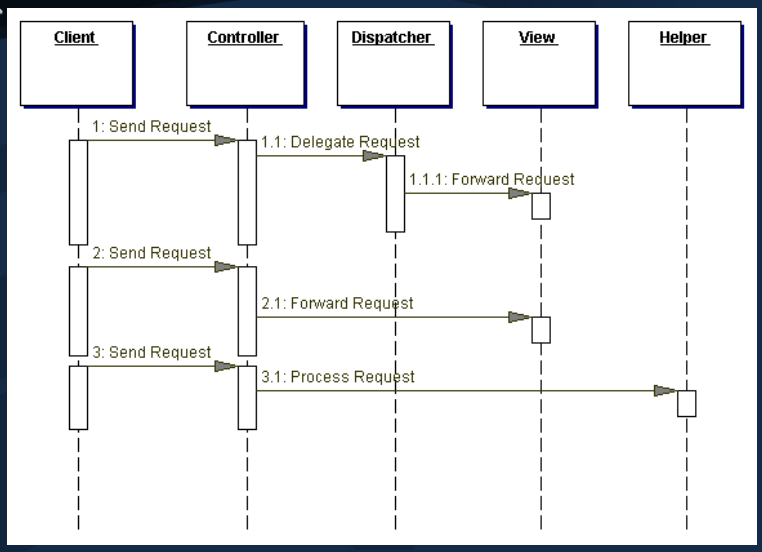
źródło: java.sun.com

Slajd przedstawia diagram interakcji dla wzorca Intercepting Filter. Sterowanie filtrami odbywa się na podstawie deskryptora wdrożenia web.xml, gdzie adresy URL są odwzorowywane na odpowiednie filtry. W momencie zgłoszenia żądania przez klienta zarządca filtrów przygotowuje wymagane filtry, łączy je w łańcuch i następnie przekazuje żądanie do pierwszego filtru w łańcuchu. Filtr wykonuje właściwe sobie przetwarzanie wstępne i zwraca sterowanie do procesu zarządzającego łańcuchem filtrów. Stąd przetworzone żądanie jest przekazywane do kolejnego filtra w łańcuchu, skąd po przetworzeniu sterowanie wraca do procesu zarządzającego łańcuchem filtrów. Po wykonaniu całego przetwarzania wstępnego zdefiniowanego w łańcuchu żądanie zostaje przekazane do obiektu docelowego. Droga powrotna odpowiedzi jest analogiczna. Filtry składające się na łańcuch można swobodnie zmieniać bez konieczności wprowadzania jakichkolwiek modyfikacji do kodu aplikacji.



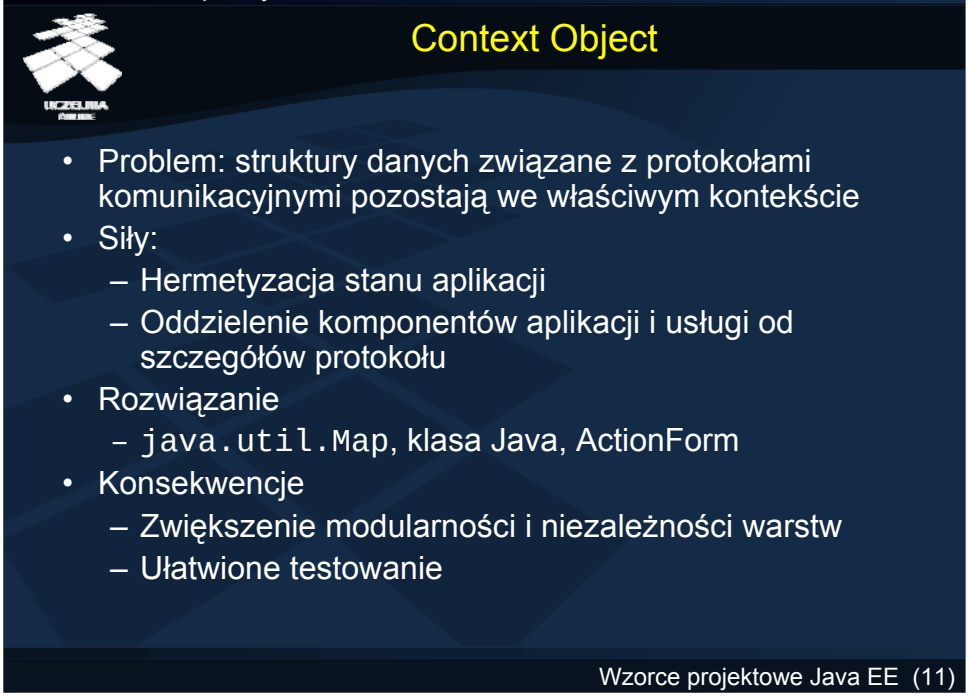
Aplikacja wymaga scentralizowanego punktu obsługi żądań klientów. Bez takiego punktu obsługą żądań zajmują się indywidualne widoki, na skutek czego procedury obsługi żądań bardzo trudno jest uspójnić. Wzorzec Front Controller wprowadza scentralizowany punkt dostępowy obsługi żądań. Dzięki temu ujednoliceniu ulega logika sterująca przetwarzaniem żądań, dodatkowo logika ta zostaje oddzielona od widoku zwiększając modułowość i elastyczność aplikacji. Istotną zaletą wprowadzenia wzorca Front Controller jest także możliwość całkowitego scentralizowania i poddania kontroli punktów dostępu do aplikacji. Istnieje wiele strategii implementacji tego wzorca, najczęściej wykorzystuje się własne serwlety, choć można także zaimplementować Front Controller jako stronę JSP (rozwiązanie nie jest polecane) albo własną klasę Java. Wiele środowisk oferujących szkielety aplikacji Java EE dostarcza gotowych implementacji wzorca. Przykładowo, w środowisku Apache Struts tę rolę pełni klasa ActionServlet, a w środowisku Spring Framework wzorzec Front Controller jest zrealizowany pod postacią klasy DispatcherServlet. Podstawowe konsekwencje stosowania wzorca Front Controller to centralizacja i poprawa zarządzania przepływem sterowania i obsługą żądań oraz możliwość wyraźnego oddzielenia ról tworzenia widoków i przetwarzania żądań.

Front Controller



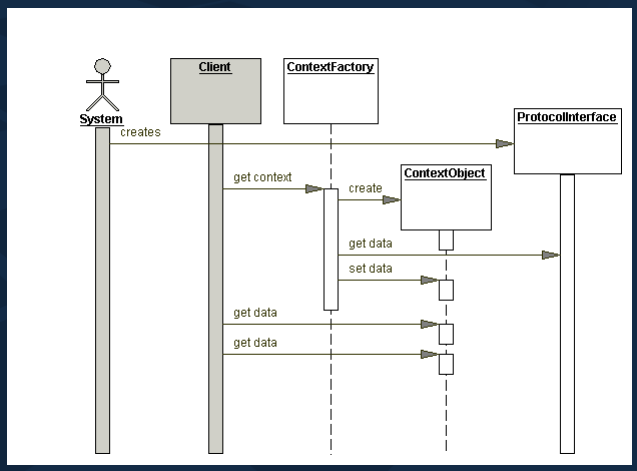
źródło: java.sun.com

Slajd przedstawia diagram interakcji wzorca Front Controller. Obiekt Controller jest miejscem przyjmującym wszystkie żądania klienta. Controller wykorzystuje obiekt Dispatcher do zarządzania widokami i nawigacją, obiekt Dispatcher wybiera właściwy widok i przekazuje tam żądanie od klienta. Dispatcher może być osobną klasą, ale może też być zaimplementowany jako część obiektu Controller. Niektóre żądania mogą być bezpośrednio przekazane przez Controller do właściwego widoku, reprezentowanego przez obiekt View. Wreszcie, część żądań może wymagać bardziej złożonego przetwarzania (np. konieczne może być sprawdzenie poprawności zalogowania się). Do implementacji bardziej złożonej logiki przetwarzania Front Controller wykorzystuje wiele klas Helper odpowiedzialnych za takie przetwarzanie.

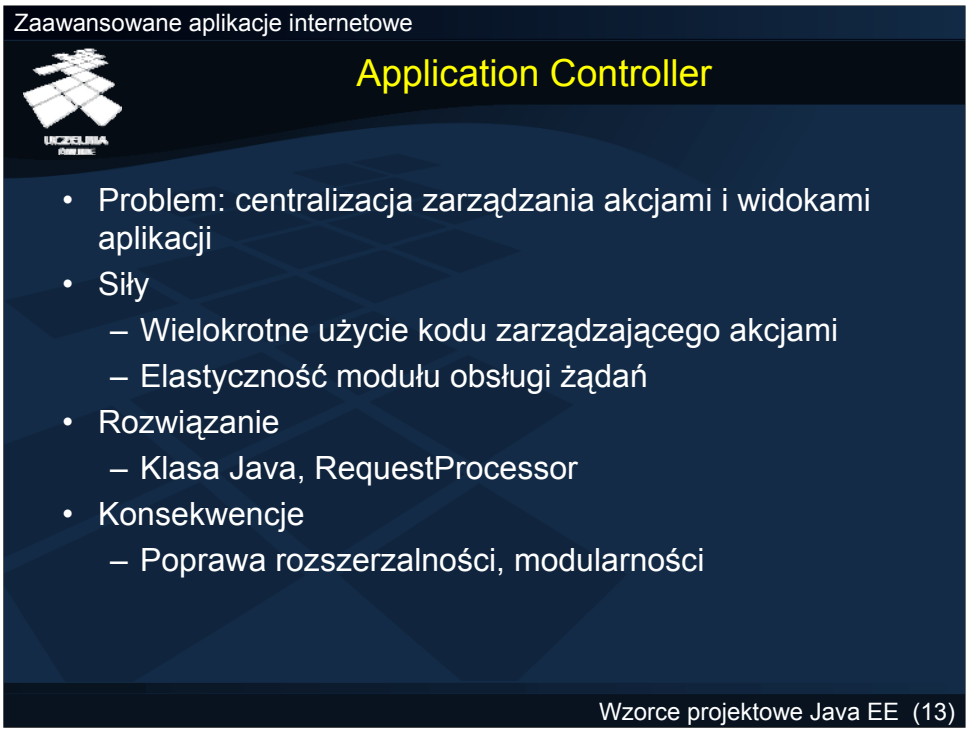


Wzorzec Context Object pozwala na hermetyzację stanu aplikacji bez względu na typ protokołu wykorzystywanego przez klienta lub inny wzorzec. Context Object nadaje się przede wszystkim do transportowania między warstwami aplikacji parametrów żądania od klienta oraz informacji systemowych. Zarówno parametry żądania, jak i informacje systemowe, są pobierane za pomocą pewnego protokołu i w określonym kontekście. Przekazywanie tych informacji do innych warstw niepotrzebnie wiąże te warstwy z używanym protokołem. Przykładowo, żądanie klienta jest zgodne z protokołem HTTP. Przekazanie do warstwy biznesowej danych w strukturze specyficznej dla HTTP niszczy modułowość i uzależnia warstwę biznesową od jednego protokołu. Zamiast tego, należy wykorzystać wzorzec Context Object, który służy jako mechanizm transportu danych między warstwami w sposób niezależny od protokołu. Implementacją wzorca Context Object jest najczęściej zwykła klasa Javy (POJO) lub mapa parametrów (lub własności) java.util.Map. Architektury szkieletowe dostarczają gotowych implementacji tego wzorca, przykładowo, w architekturze Apache Struts rolę Context Object pełnią klasy ActionForm i DynaActionForm. Zaletami użycia wzorca Context Object są zwiększenie modularności i niezależności między warstwami aplikacji oraz ułatwia automatyczne testowanie aplikacji za pomocą narzędzi testujących, np. JUnit. Wadą wzorca Context Objext jest niewielkie zmniejszenie wydajności związane z koniecznością przenoszenia stanu aplikacji do wzorca.

Context Object

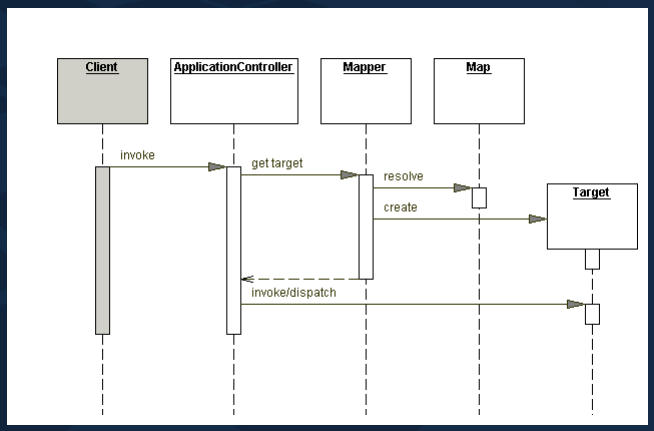


Slajd przedstawia diagram interakcji wzorca Context Object. Wzorzec tworzy nowy obiekt ContextObject za pomocą fabryki ContextFactory. Następnie, fabryka wykorzystuje interfejs ProtocolInterface do pobrania specyficznych danych o protokole i zapisuje te dane w obiekcie ContextObject. Kolejne żądania pobrania danych pochodzące od klientów są już kierowane do obiektu ContextObject, a nie do interfejsu ProtocolInterface, dzięki czemu klient jest nieświadom szczegółów wykorzystywanego protokołu.



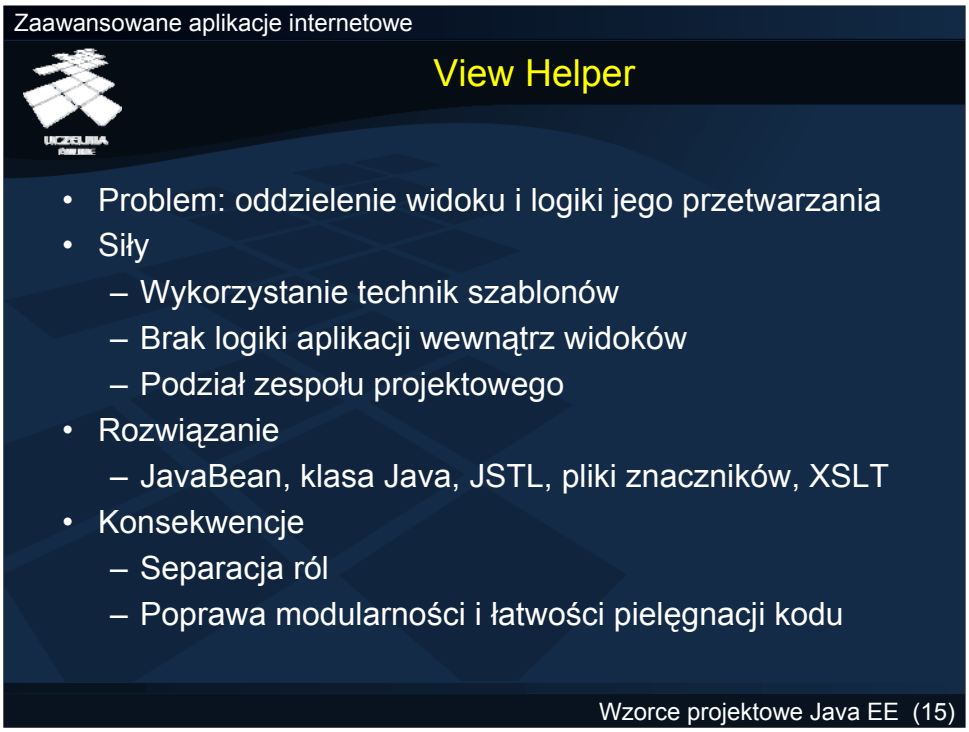
Warstwa prezentacji po otrzymaniu żądania musi wykonać dwie czynności: (1) wybrać właściwą akcję realizującą obsługę żądania – zarządzanie akcjami – oraz (2) wybrać właściwy widok do zwrócenia klientowi – zarządzanie widokami. Tradycyjnie tymi zadaniami zajmuje się Front Controller. Wraz ze wzrostem rozmiaru i stopnia złożoności aplikacji należy wydzielić z Front Controller osobny moduł, który w sposób deklaratywny potrafi zarządzać akcjami i widokami. Takim modułem jest implementacja wzorca Application Controller. Wykorzystanie tego wzorca umożliwia wielokrotne użycie kodu zarządzającego akcjami oraz uelastycznienie modułu obsługi żądań. W przypadku wykorzystania wzorca Application Controller nowe procedury obsługi mogą być dodawane bez konieczności modyfikacji istniejącego kodu, zaś istniejące procedury obsługi mogą być dowolnie modyfikowane. Implementacją tego wzorca jest najczęściej klasa Java, choć architektury szkieletowe często dostarczają gotowych implementacji. Przykładowo, w architekturze Apache Struts rolę Application Controller pełni klasa RequestProcessor. Zysk ze stosowania wzorca Application Controller polega przede wszystkim na poprawie rozszerzalności i modularności aplikacji, co ułatwia testowanie, modyfikowanie i pielęgnację aplikacji.

Application Controller



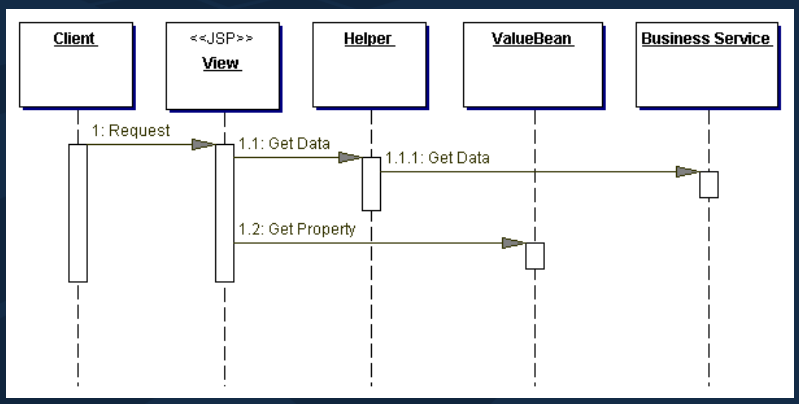
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Application Controller. Klient wywołuje obiekt ApplicationController, który pobiera nazwę akcji do wykonania i wykorzystuje obiekt Mapper odwzorowujący nazwy akcji na obiekty docelowe do uzyskania informacji o właściwym obiekcie docelowym Target. Obiekt Mapper wykorzystuje obiekt Map do pobrania referencji do docelowej akcji (lub widoku) i tworzy obiekt docelowy, a następnie zwraca referencję do obiektu docelowego do obiektu ApplicationController, który wywołuje właściwą akcję.



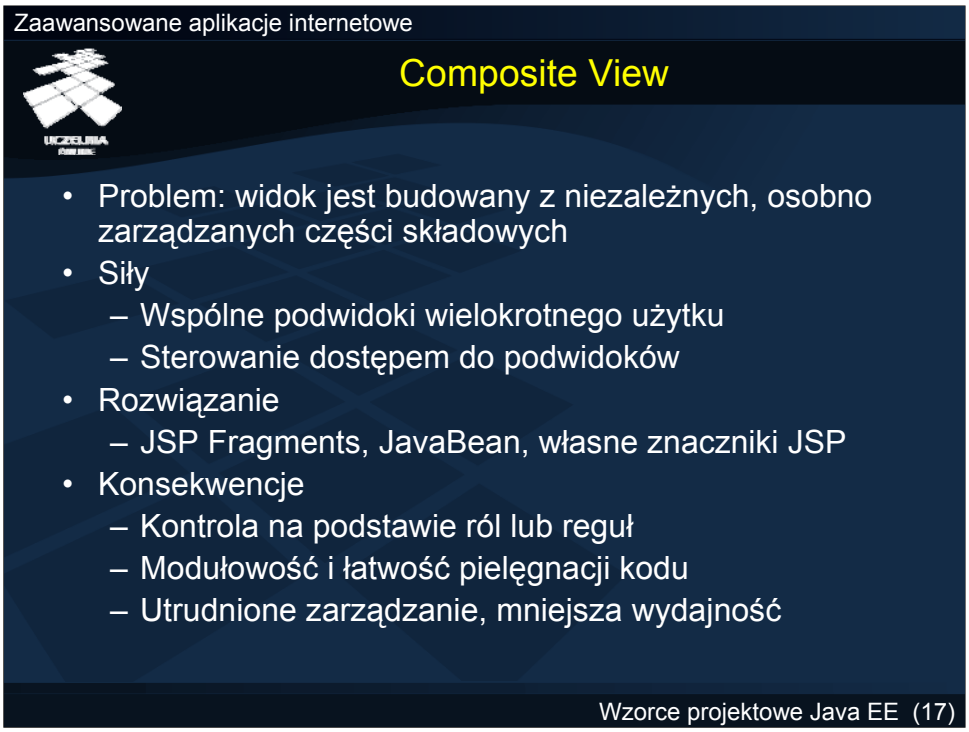
Logika sterowania, formatowanie i dostęp do danych przemieszane wewnątrz komponentów widoku stanowią jeden z najczęstszych błędów projektowych. Zastosowanie wzorca Front Controller wydziela logikę sterowania, natomiast model i komponenty widoku nadal pozostają ściśle związane. Przetwarzanie samego widoku składa się z dwóch faz: (1) przygotowania widoku oraz (2) tworzenia widoku. Przygotowanie widoku to odwzorowanie żądania na właściwą akcję powodującą wybór widoku i przekazanie żądania do tego widoku. Tworzenie widoku wymaga odczytania danych z modelu i wygenerowanie odpowiedzi udzielanej klientowi. Aby uniknąć powiązania modelu i widoku wynikowego wprowadza się wzorzec View Helper. Umożliwia on transparenty dostęp do modelu i umieszczenie modelu w miejscu swobodnie dostępnym dla komponentów widoku. Wykorzystanie tego wzorca ułatwia korzystanie z technik szablonów w widokach, wyczyszczenie widoków poprzez usunięcie praktycznie całej logiki aplikacji z widoków, wreszcie pozwala na podział zespołu projektowego na grupę odpowiedzialną za generowanie widoku oraz grupę odpowiedzialną za dostęp do modelu. Wzorzec View Helper może mieć bardzo wiele strategii implementacji, najczęściej wykorzystuje się komponenty JavaBean, tradycyjne klasy Java, oraz techniki szablonów: JSTL, pliki znaczników, czy XSLT. Przykładowo, implementacją wzorca View Helper może być klasa Java, która łączy się z bazą danych, odczytuje zawartość tabeli, a następnie transformuje tę tabelę do języka HTML. Konsekwencją wykorzystania wzorca jest, jak wcześniej zaznaczono, pełna separacja ról projektowych oraz znaczna poprawa modularności i łatwości pielęgnacji kodu.

View Helper



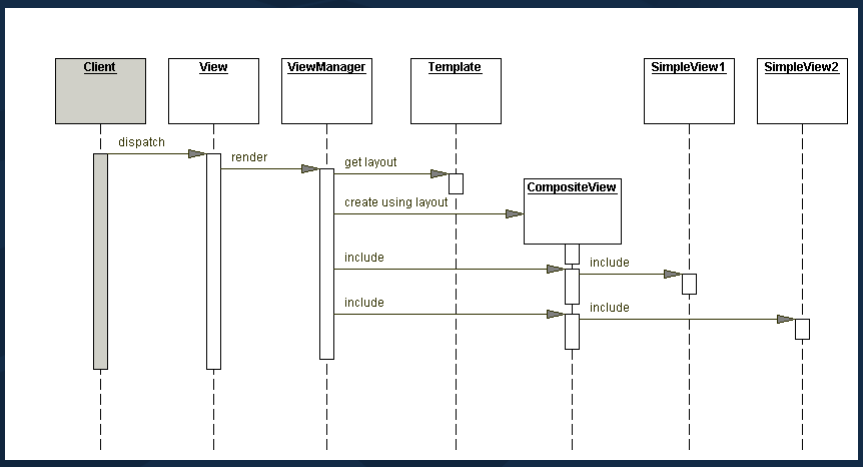
źródło: java.sun.com

Slajd przedstawia diagram interakcji wzorca View Helper. Klient przesyła żądanie, które zostaje dostarczone do komponentu widoku. Komponent widoku wywołuje obiekt Helper w celu przygotowania modelu danych, który zostanie wyświetlony. Obiekt Helper łączy się z usługą biznesową (obiekt BusinessService) lub bazą danych i umieszcza znalezione dane w tle (najczęściej wykorzystując wzorzec Transfer Object). W kolejnym wołaniu warstwa widoku pobiera do wyświetlenia dane wcześniej przygotowane przez komponent obiekt Helper.



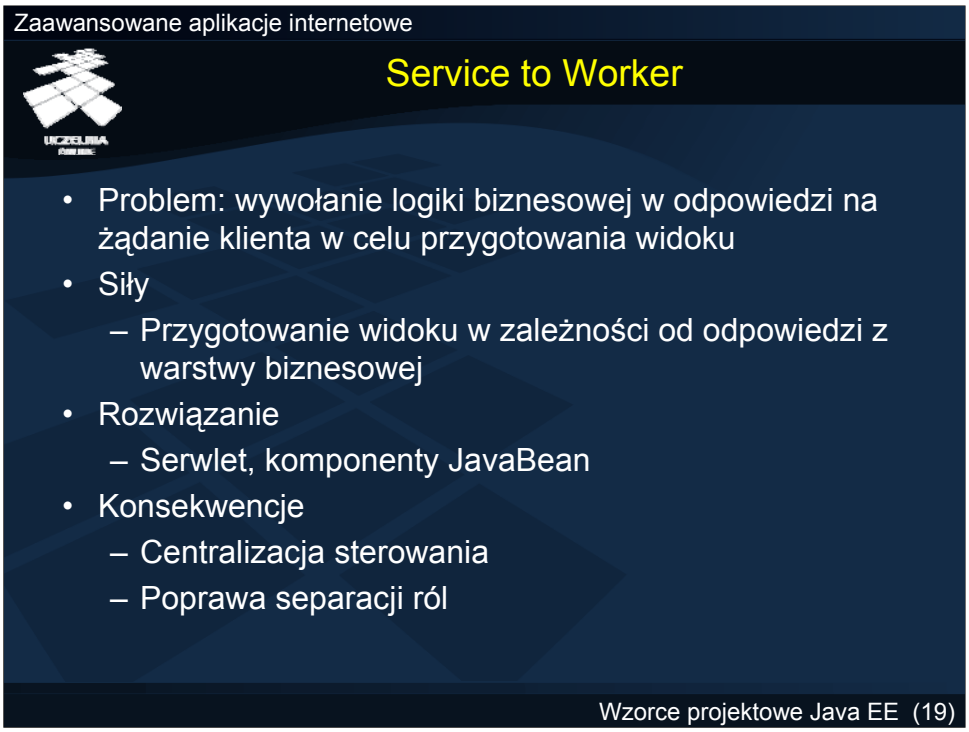
Szata graficzna wielu wynikowych widoków składa się z niezależnych części. Części te mogą odpowiadać podwidokom zawierającym różnego rodzaju informacje (portale), podwidokom odświeżanym z różną częstotliwością, podwidokom powtarzającym się w ramach wielu widoków (np. nagłówki i stopki stron), wreszcie podwidokom, do których dostęp powinien być ograniczony na podstawie ról i uprawnień. Zastosowanie wzorca Composite View pozwala na konstruowanie widoku wynikowego z niezależnie zarządzanych części składowych. Dzięki temu podwidoki powtarzające się w wielu widokach mogą być wielokrotnie wykorzystywane, a dostęp do poszczególnych podwidoków może być łatwo monitorowany. Najczęściej strategią implementacji wzorca Composite View jest technologia szablonów, np. JSP Fragments lub JSF, choć popularnymi rozwiązaniami jest wykorzystanie komponentów JavaBean lub przygotowanie własnych znaczników JSP (szczególnie przy użyciu plików znaczników, JSP 2.0+). Wiele architektur szkieletowych zapewnia własne implementacje wzorca (Struts Tiles) lub umożliwia łatwą integrację technologii szablonów, takich jak Velocity lub FreeMarker, z resztą architektury (Spring Framework). Liczne zalety stosowania wzorca obejmują, między innymi, kontrolę dostępu do podwidoków na podstawie ról i reguł oraz modułowość i łatwość pielęgnacji kodu ze względu na brak duplikacji kodu między widokami. Należy także wspomnieć o tym, że wzorzec Composite View zmniejsza wydajność aplikacji poprzez wprowadzenie dodatkowej warstwy i pewnego narzutu związanego z zarządzaniem wyborem i kontrolą dostępu do podwidoków.

Composite View



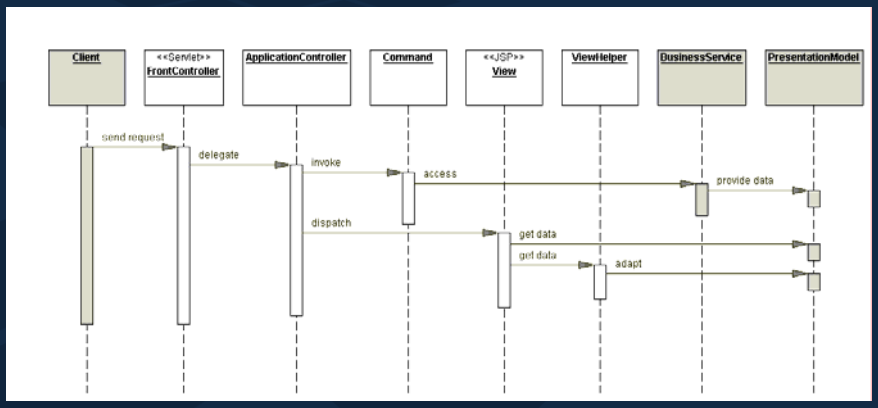
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Composite View. Obiekt View reprezentuje wyświetlany widok. Obiekt ViewManager wykorzystuje obiekt Template do zbudowania szablonu definiującego układ elementów wyświetlanego widoku. Natępnie, na podstawie szablonu tworzy obiekt CompositeView będący kolekcją obiektów SimpleView (segment widoku, podwidok) i CompositeView reprezentujących wyświetlaną zawartość.



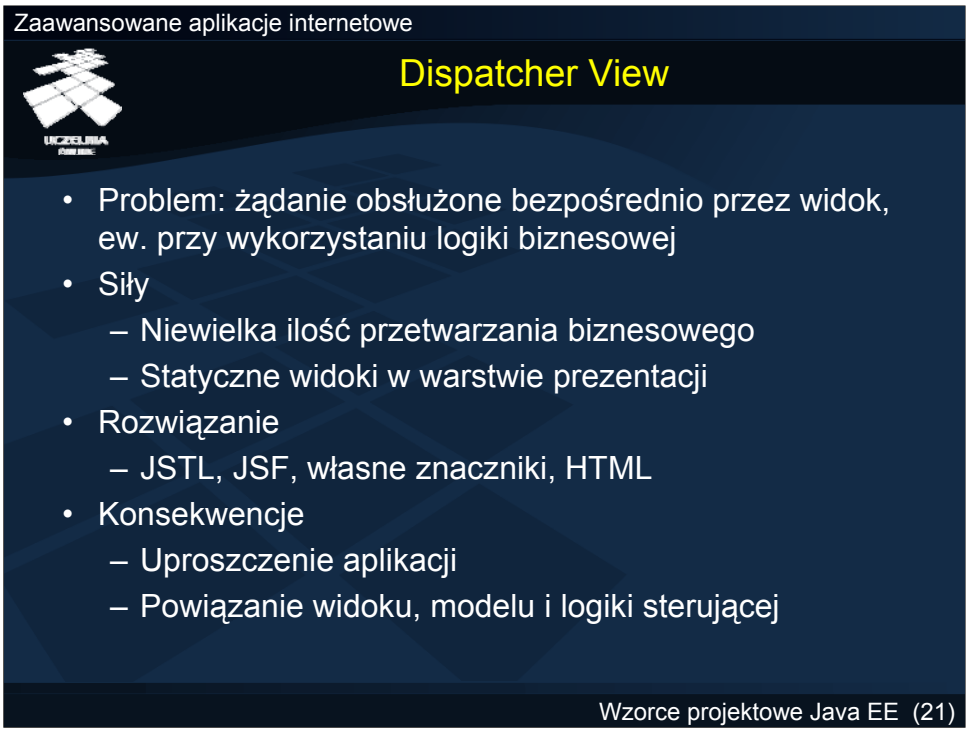
Wynik obsługi wielu żądań zależy od tego, jaka odpowiedź zostanie zwrócona z warstwy biznesowej. Przykładowo, po zalogowaniu się użytkownika do banku internetowego wygląd widoku podstawowego zależy od tego, jakie konto posiada użytkownik i jakie dodatkowe usługi zostały przez użytkownika aktywowane. Zależy nam więc na wywołaniu logiki biznesowej (np. połączeniu z bazą danych i weryfikacji typu konta w banku) przed przekazaniem sterowania do modułu odpowiedzialnego za generowanie widoku. Taką organizację zapewnia wykorzystanie wzorca Service to Worker. Jest to wzorzec złożony, składający się z wzorców Front Controller, Application Controller i View Helper. Umożliwia on wstępne przygotowanie widoku na podstawie danych zwróconych z warstwy biznesowej przed faktycznym rozpoczęciem generowania widoku. Zdecydowanie najczęstszą strategią implementacji wzorca jest serwlet pełniący rolę kontrolera. W praktyce wzorzec ten jest implementowany jako najważniejszy element architektury MVC w wielu popularnych architekturach szkieletowych, takich jak Apache Struts, Spring Framework czy JSF. Możliwa jest też "ręczna" implementacja wzorca, np. za pomocą komponentów JavaBean, choć ze względu na stopień skomplikowania wzorca nie jest to zalecana strategia. Podstawowe zalety wzorca to centralizacja sterowania, poprawa modularności, łatwość pielęgnacji, oraz poprawa separacji ról członków zespołu projektowego.

Service to Worker



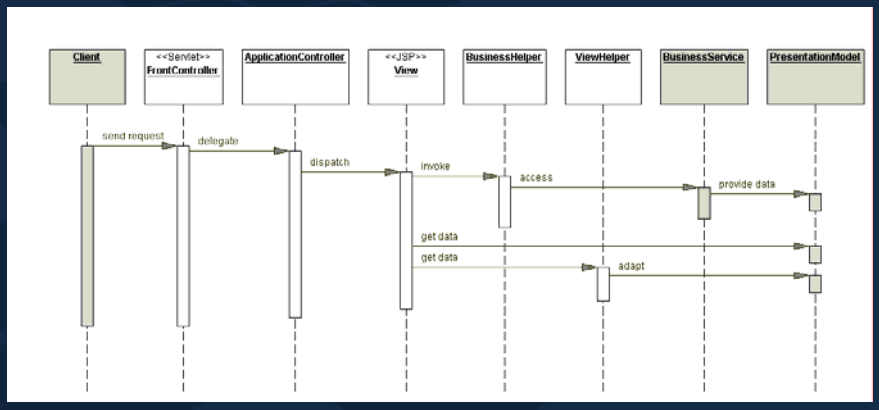
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca złożonego Service to Worker. W pierwszym etapie Front Controller przejmuje i wstępnie obsługuje żądania klienta, a następnie przekazuje sterowanie do modułu Application Controller. Poprzez obiekt Command (stanowiący hermetyzację wywołania usługi biznesowej) następuje wywołanie usługi biznesowej oraz dostarczenie danych do obiektu Presentation Model. Następnie moduł Application Controller wybiera widok wynikowy reprezentowany przez obiekt View. Widok pobiera potrzebne dane z przygotowanego wcześniej obiektu Presentation Model. Jeśli widok jest złożony, może wykorzystać dodatkowy obiekt View Helper do dostosowania widoku zgodnie z pobranymi danymi. Istotną cechą wzorca Service to Worker jest fakt, że dostęp do usług biznesowych odbywa się przed przekazaniem sterowania do modułu generowania widoku.



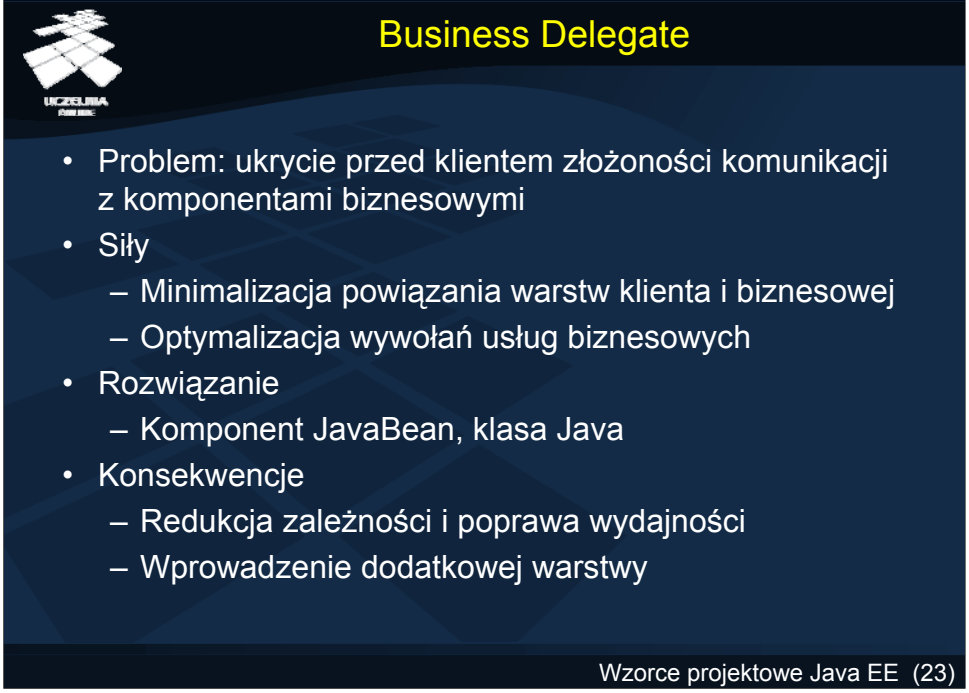
W wielu przypadkach przetwarzanie żądań klienta jest na tyle proste, że stosowanie wzorca Service to Worker staje się nadużyciem. Jeśli żądanie może być bezpośrednio obsłużone przez widok, a wykorzystywana logika biznesowa jest bardzo prosta, podobnie jak logika biznesowa, to właściwym rozwiązaniem projektowym jest wzorzec Dispatcher View. Wzorzec ten umożliwia umieszczanie niewielkiej ilości logiki sterującej i biznesowej bezpośrednio w widokach. Oznacza to, że widoki w warstwie prezentacji są w większości statyczne lub mogą być wygenerowane przy niewielkiej ilości przetwarzania biznesowego. Podstawową strategią implementacji wzorca Dispatcher View są widoki JSP, najczęściej generowane przy użyciu biblioteki znaczników (np. standardowej biblioteki JSTL) lub szkieletu aplikacji (np. JSF). Można pokusić się o implementację wzorca przy użyciu własnych znaczników lub bezpośrednio wykorzystując statyczne strony HTML. Wzorzec Dispatcher View jest też bardzo często stosowany na etapie budowania i testowania aplikacji, gdyż umożliwia niezależne testowanie widoków. W odróżnieniu od wzorca Service to Worker, we wzorcu Dispatcher View usługi i logika biznesowa są wywoływane już z poziomu warstwy prezentacji, a nie z komponentu sterującego. Wzorzec upraszcza aplikację przez eliminację nadmiarowych modułów, natomiast nadaje się tylko do prostego przetwarzania, gdyż bardzo silnie wiąże widok, model i logikę sterującą.

Dispatcher View



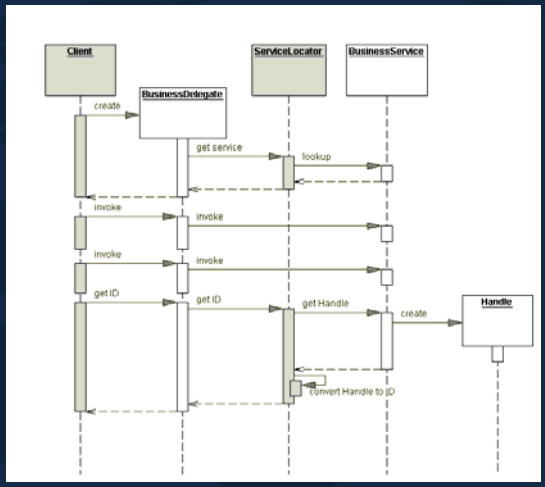
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Dispatcher View. FrontController przejmuje żądanie i dokonuje wstępnego przetworzenia żądania, a następnie przekazuje sterowanie do obiektu View. Obiekt ten wywołuje usługę biznesową (reprezentowaną przez obiekt BusinessService) poprzez interfejs dostarczony przez obiekt BusinessHelper. Dane dostarczone przez usługę biznesową są umieszczane w obiekcie PresentationModel. Po przygotowaniu tego obiektu obiekt View pobiera dane z obiektu PresentationModel i generuje widok wynikowy. Jeśli mamy do czynienia z widokiem złożonym, to obiekt View może wykorzystać pomocniczy obiekt ViewHelper do pobrania danych z obiektu PresentationModel, przygotowaniu szkieletu widoku i dostosowaniu widoku wynikowego do pobranych danych.



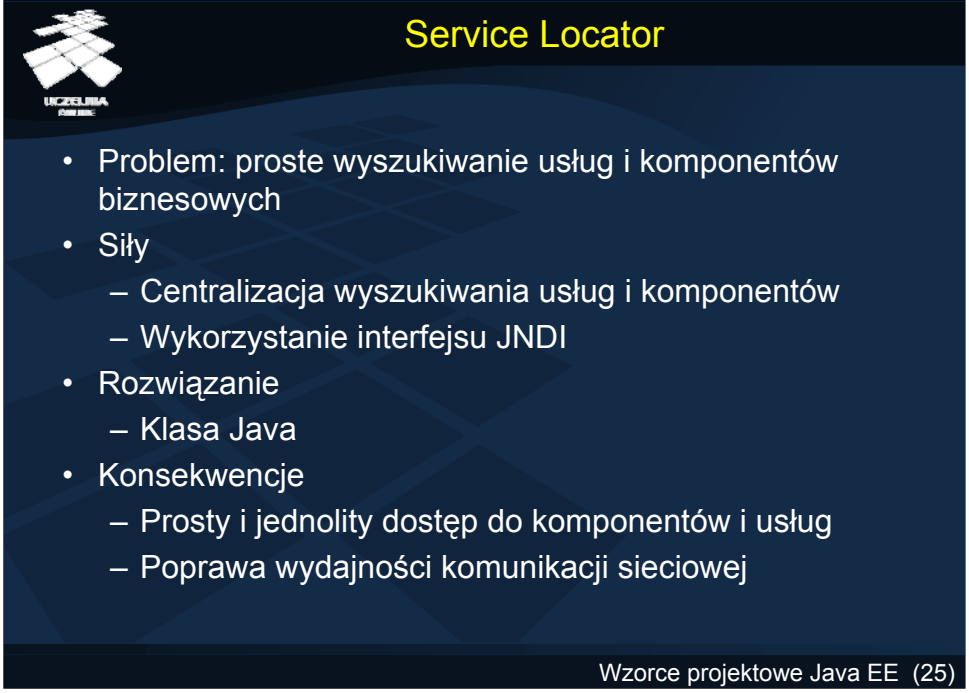
Usługi biznesowe nie powinny być wywoływane bezpośrednio z warstwy klienta lub warstwy prezentacji, gdyż niesie to ze sobą określone problemy: (1) każda zmiana interfejsu usługi biznesowej musi być przeniesiona do klienta lub warstwy prezentacji, (2) osadzenie logiki biznesowej po stronie klienta uniemożliwia łączenie i buforowanie wywołań usług biznesowych i obciąża sieć, (3) wyjątki związane z usługami biznesowymi lub komunikacją sieciową muszą być przechwytywane i obsługiwane w warstwie klienta lub prezentacji. Wzorzec Business Delegate pełni rolę abstrakcyjnej usługi biznesowej po stronie klienta i ukrywa przed klientem złożoność komunikacji i interakcji z usługami biznesowymi. Dzięki temu minimalizuje się powiązanie między warstwami oraz optymalizuje wywołania usług biznesowych. Implementacją wzorca Business Delegate jest najczęściej klasa Java, może to być np. komponent JavaBean. Po stronie zysków zaliczyć należy przede wszystkim redukcję zależności między warstwami, poprawę wydajności komunikacji i zwiększenie modularności kodu. Wadą wprowadzenia wzorca jest obecność dodatkowej warstwy komplikującej wewnętrzną architekturę aplikacji.

Business Delegate



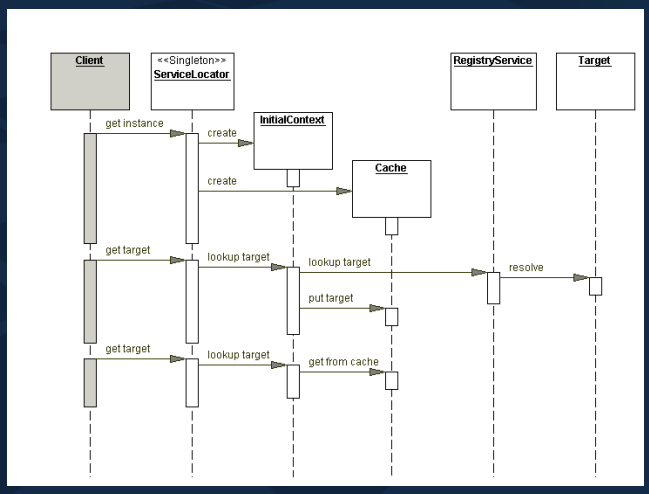
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Business Delegate. Obiekt BusinessDelegate wykorzystuje wzorzec Service Locator do znalezienia właściwej usługi biznesowej i po jej znalezieniu przekazuje sterowanie do klienta. Gdy klient chce wywołać usługę biznesową, wywołuje tę usługę poprzez obiekt BusinessDelegate. Klient ma również możliwość pobrania identyfikatora usługi biznesowej (np. uchwyt do komponentu EJB). W takim wypadku obiekt BusinessDelegate wykorzystuje wzorzec Service Locator do znalezienia usługi, następnie tworzy uchwyt do usługi (reprezentowany przez obiekt Handle) i zwraca ten uchwyt do klienta. Pozwala to, w późniejszych wywołaniach, na pominięcie kosztownego i czasochłonnego etapu wyszukiwania usługi.



Komponenty warstwy klienta i prezentacji często muszą uzyskać dostęp do komponentów i usług warstwy biznesowej. Podstawową metodą wyszukiwania obiektów w Java EE jest wykorzystanie interfejsu JNDI (Java Naming and Directory Interface) odwzorowującego nazwy obiektów i usług na referencje do konkretnych obiektów. Korzystanie z JNDI z poziomu klienta nastręcza wiele trudności, związanych choćby z operacjami wyszukiwania, rzutowania klas, obsługą niskopoziomowych wyjątków, obsługą upływu limitu czasu, itp. Wzorzec Service Locator ukrywa szczegóły implementacji i złożoność mechanizmu wyszukiwania przed klientami. Stanowi on centralny punkt obsługi wyszukiwania usług i komponentów warstwy biznesowej za pomocą interfejsu JNDI. Wzorzec Service Locator jest implementowany jako klasa Java hermetyzująca dostęp do JNDI i obsługująca wszystkie wyjątki niskiego poziomu. Zastosowanie wzorca Service Locator wprowadza jednolity dostęp do komponentów i usług, ułatwia dodawanie nowych komponentów w warstwie biznesowej (komponenty EJB, źródła danych, kolejki komunikatów), oraz poprawia wydajność komunikacji sieciowej poprzez buforowanie.

Service Locator



źródło: www.corej2eepatterns.com

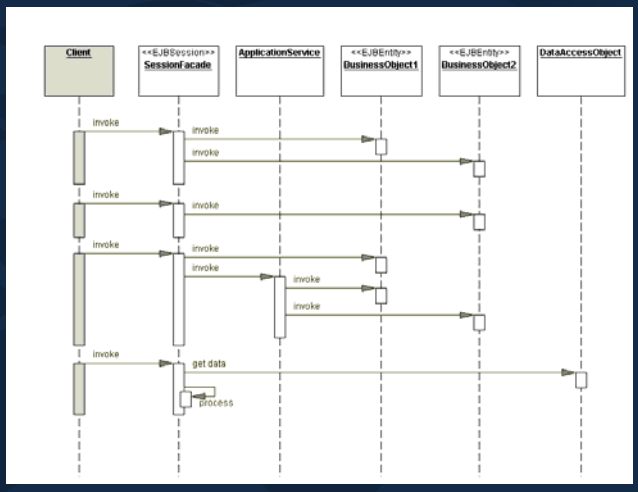
Slajd przedstawia diagram interakcji wzorca Service Locator. Obiekt ServiceLocator wykorzystuje obiekt InitialContext będący punktem startowym wyszukiwania komponentów. Konkretna implementacja tego obiektu zależy od rodzaju poszukiwanego komponentu (komponent EJB, źródło danych, JMS). Obiekt Cache reprezentuje opcjonalny bufor do przechowywania wcześniej znalezionych obiektów. Obiekt InitialContext pobiera z obiektu RegistryService informacje o wyszukiwanym komponencie. Wyszukiwany komponent jest reprezentowany przez obiekt Target. Obiekt ten zostaje umieszczony w podręcznym buforze i zwrócony do klienta, który zainicjował proces wyszukiwania. Kolejne żądania zlokalizowania tego samego komponentu nie będą już wymagały dostępu do obiektu InitialContext, lecz zostaną zrealizowane na podstawie zawartości bufora.

Session Façade

- Problem: udostępnienie komponentów i usług biznesowych zdalnym klientom
- Siły
  - Brak zależności między klientem i warstwą biznesową
  - Centralizacja logiki biznesowej
  - Ukrycie zależności w warstwie biznesowej
- Rozwiązanie
  - Sesyjny komponent EJB
- Konsekwencje
  - Poprawa wszystkich aspektów aplikacji

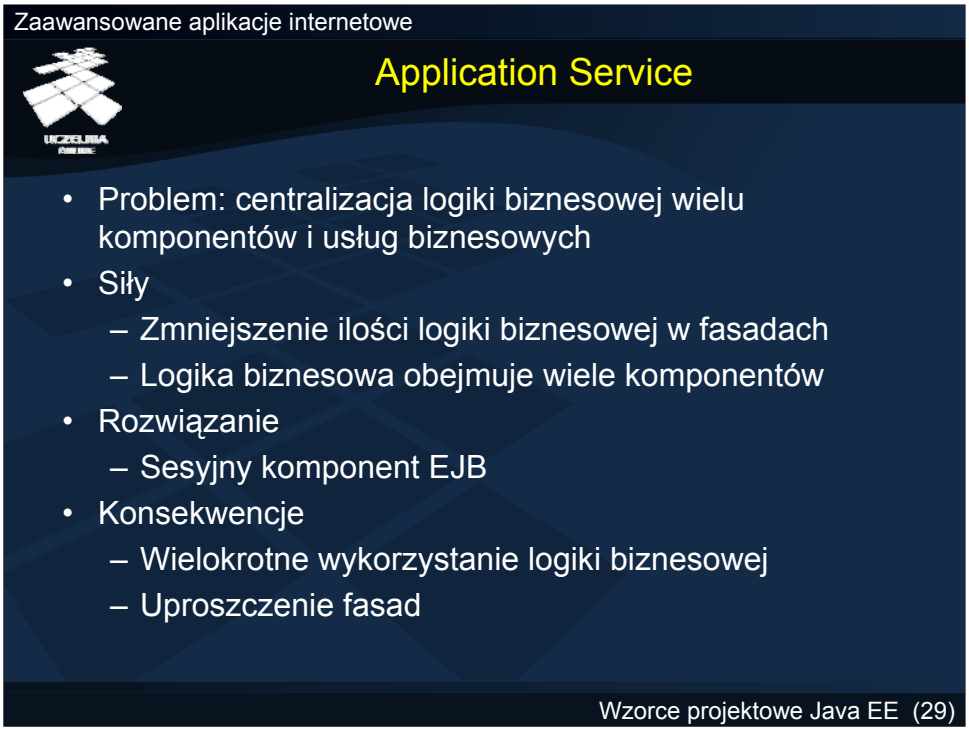
Bezpośrednie udostępnienie usług warstwy biznesowej klientom może mieć niekorzystny wpływ na architekturę aplikacji. W takim przypadku klienci stają się mocno związani z interfejsem warstwy biznesowej, co zmniejsza elastyczność i znacznie podnosi koszt wprowadzania ewentualnych zmian w warstwie biznesowej. Bezpośrednie operowanie na komponentach biznesowych wymaga od klientów implementacji złożonej logiki biznesowej i oprogramowywania złożonych interakcji między komponentami biznesowymi. Klienci muszą być świadomi szczegółów przetwarzania transakcyjnego, zarządzania bezpieczeństwem i kontrolą dostępu, itp. Wzorzec Session Facade umożliwia ukrycie złożonych zależności i związków między komponentami biznesowymi za fasadą komponentów sesyjnych i udostępnienie klientom interfejsu warstwy biznesowej w spójny i prosty sposób. Uzyskujemy dzięki temu wzorcowi niezależność między warstwą klienta i warstwą biznesową, logika biznesowa zostaje scentralizowana w warstwie prezentacji, zwiększa się modułowość i elastyczność aplikacji. Dodatkowo, warstwa komponentów tworzących wzorzec Session Facade może publikować interfejs warstwy biznesowej zgodny z logiką biznesową (procesami i usługami), a nie zgodny z komponentami biznesowymi reprezentującymi poszczególne encje. Wzorzec Session Facade jest implementowany najczęściej za pomocą stanowych komponentów sesyjnych EJB, choć zdarzają się implementacje wykorzystujące bezstanowe komponenty sesyjne EJB. Wzorzec Session Facade jest zalecany w każdej dużej aplikacji webowej, a jego zastosowanie ma pozytywny wpływ na praktycznie każdy aspekt aplikacji: ujednolicenie interfejsu, zmniejszenie zależności między warstwami, zwiększenie elastyczności, zmniejszenie złożoności, centralizacja zarządzania transakcjami i bezpieczeństwem, poprawa wydajności.

Session Façade



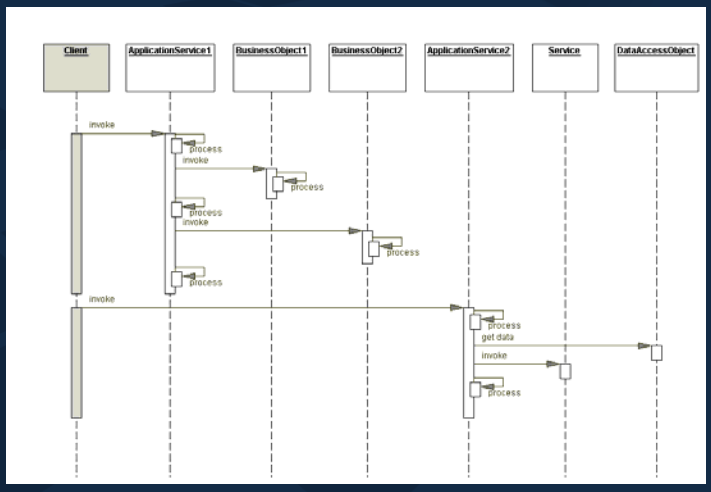
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Session Facade. Żądania klienta dostępu do usług zdalnych są kierowane bezpośrednio do obiektu SessionFacade będącego stanowym lub bezstanowym komponentem sesyjnym. Obiekt SessionFacade przenosi wywołanie usługi do komponentu biznesowego, którym może być obiekt BusinessObject (np. encyjny komponent EJB), obiekt ApplicationService (klasa Java implementująca złożoną logikę biznesową), lub DataAccessObject (obiekt implementujący wzorzec Data Access Object i umożliwiający pobieranie danych z zewnętrznych źródeł danych).



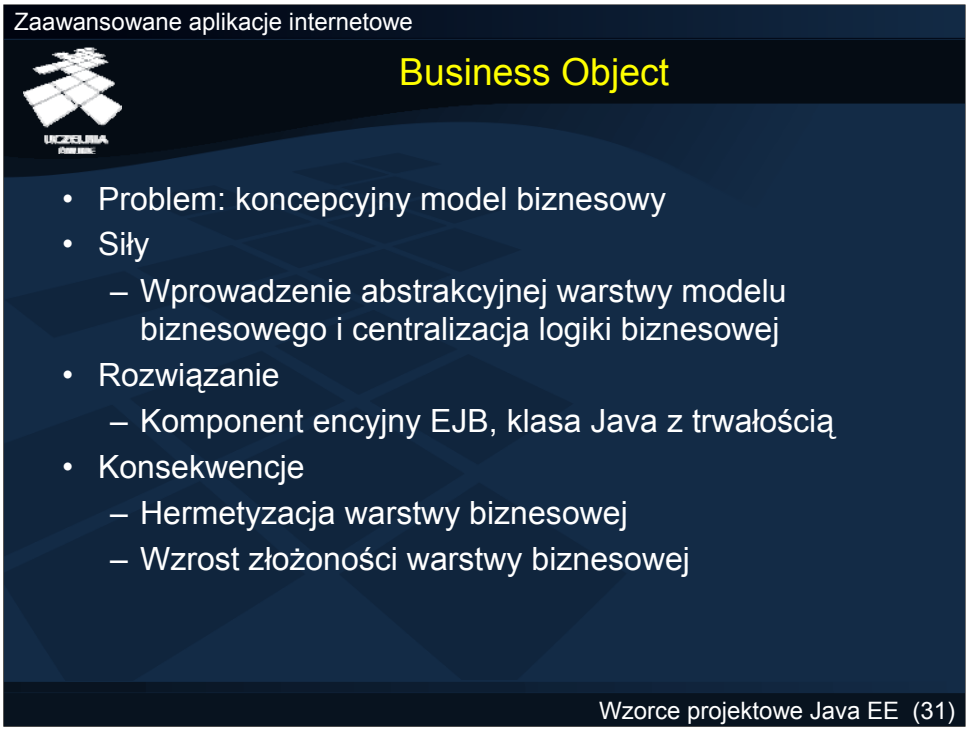
W przypadku, gdy logika biznesowa obejmująca interakcje między wieloma komponentami biznesowymi staje się zbyt złożona, umieszczanie tej logiki w fasadach warstwy prezentacji zaczyna powodować niespodziewane problemy, w szczególności wzrasta stopień zależności między warstwami. Wzorzec Application Service wprowadza dodatkową warstwę obiektów w warstwie biznesowej. Obiekty te, implementowane jako sesyjne komponenty EJB, stanowią hermetyzację logiki biznesowej i zarządzania współdziałaniem między komponentami biznesowymi. Dzięki temu znacznemu uproszczeniu ulegają fasady warstwy prezentacji a interfejs warstwy biznesowej staje się czytelniejszy i spójniejszy. Dodatkowo, ta sama logika biznesowa może być łatwo wykorzystywana przez różne komponenty warstw klienta i prezentacji. Wzorzec Application Service powinien być stosowany w praktycznie wszystkich aplikacjach webowych.

Application Service



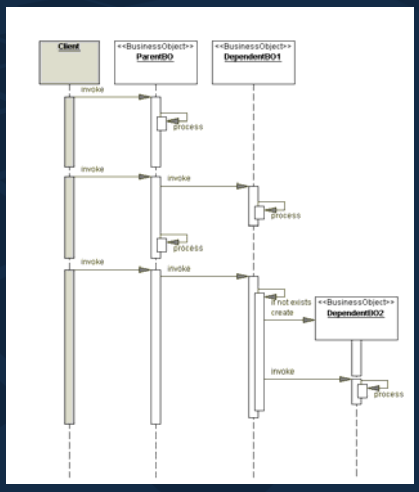
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Application Service. Klient wywołuje usługę biznesową przesyłając żądanie do obiektu ApplicationService. Obiekt ten przetwarza żądanie wykonując zaimplementowaną w sobie logikę biznesową i zwracając się, jeśli to konieczne, do konkretnych komponentów biznesowych (reprezentowanych przez obiekty BusinessObject). Istotna różnica między wzorcami Application Service i Business Object polega na tym, że Business Object reprezentuje wykonanie logiki biznesowej na pojedynczym obiekcie, a Application Service reprezentuje logikę biznesową obejmująca wiele obiektów. Obiekty ApplicationService zarządzają nie tylko dostępem do komponentów biznesowych, ale także usług (reprezentowanych przez obiekt Service) i źródeł danych (obiekt DataAccessObject).



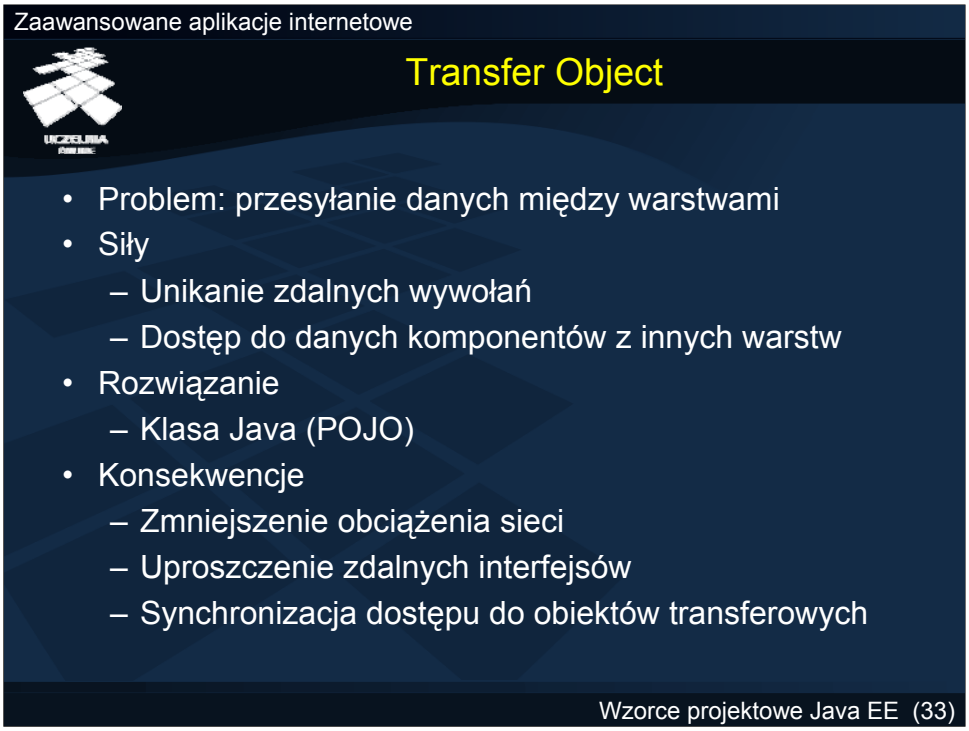
W prostych i niewielkich aplikacjach webowych warstwa prezentacji za zwyczaj umożliwia bezpośredni dostęp klientom do danych biznesowych, np. przez obiekty dostępu do danych. W takich przypadkach warstwa biznesowa nie posiada własnego, obiektowego modelu danych i aplikacja korzysta z dwóch modeli danych: modelu obiektowego wykorzystywanego w warstwie prezentacji oraz modelu relacyjnego wykorzystywanego w warstwie danych. Większe aplikacje webowe wymagają jednak wprowadzenia w warstwie biznesowej własnego modelu danych. Zwiększa to możliwość wielokrotnego wykorzystania kodu implementacji logiki biznesowej, dodatkowo separuje warstwę biznesową od warstw klienta i prezentacji, upraszcza kod proceduralny logiki biznesowej. Dodatkową warstwę obiektowego modelu warstwy biznesowej wprowadza wzorzec Business Object. Implementacją tego wzorca jest najczęściej encyjny komponent EJB, a w przypadku prostszych systemów klasa Java z zarządcą trwałości (Hibernate, JDO). Obiekty Business Object zarządzają danymi, logiką biznesową dotyczącą danych, oraz trwałością danych (ta ostatnia może być zarządzana automatycznie przez kontener). Wprowadzenie dodatkowej warstwy zawierającej obiektowy model biznesowy powoduje prawdziwą hermetyzację warstwy biznesowej, a co za tym idzie, separację warstw, zwiększenie modularności kodu, rozdział ról projektowych, itp. Trzeba jednak pamiętać, że kosztem implementacji wzorca Business Object jest znaczący wzrost złożoności warstwy biznesowej.

Business Object



źródło: www.corej2eepatterns.com

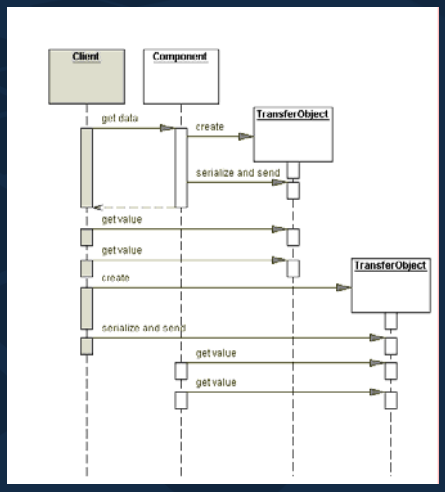
Slajd przedstawia diagram interakcji wzorca Business Object. Klient wywołuje logikę biznesową obiektu BusinessObject. Obiekt ten wczytuje swoje dane ze źródła danych (np. z relacyjnej bazy danych) i wykonuje żądaną logikę. W celu obsłużenia żądania konieczne może się okazać wywołanie logiki biznesowej w innym, zależnym obiekcie BusinessObject, a czasem wręcz utworzenie nowego obiektu BusinessObject i wywołanie w nim właściwej logiki biznesowej.



Wzorzec Transfer Object porządkuje i upraszcza komunikację między warstwami aplikacji. Umożliwia on zamianę zdalnych wywołań metod komponentów (dotyczy to przede wszystkim komponentów sesyjnych i encyjnych) na transfer danych między warstwami. Wzorzec Transfer Object jest implementowany w postaci obiektów transferowych należących do prostych klas Java (POJO – plain old Java object). Dzięki ich zastosowaniu interfejsy zdalne wielu komponentów upraszczają się do dwóch metod: getData() i setData(), a wszystkie pozostałe operacje są wykonywane na kopii danych przyniesionych przez obiekt transferowy. Zastosowanie obiektów transferowych redukuje też obciążenia sieci, ponieważ transfer danych między warstwami jest realizowany za pomocą pojedynczego przesłania. Niestety, stosowanie obiektów transferowych narzuca konieczność synchronizacji zawartości obiektów transferowych ze źródłem danych (obiekt transferowy może zawierać dane, które stały się już nieaktualne) oraz kontroli współbieżnego dostępu do takich obiektów.

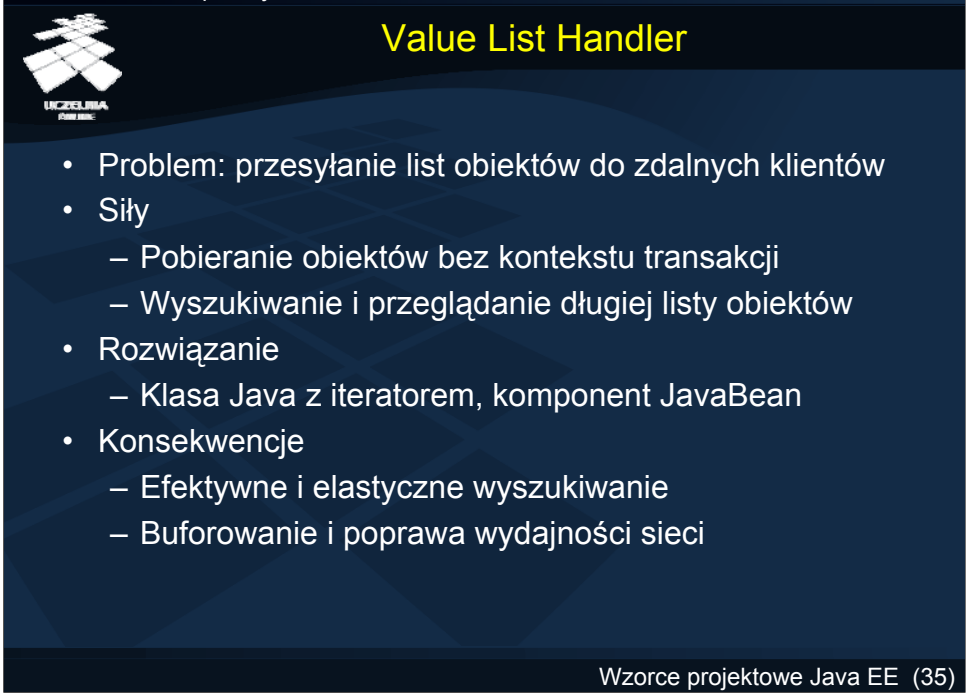
Ze wzorcem Transfer Object ściśle wiąże się wzorzec Transfer Object Assembler. Jest to wzorzec projektowy umożliwiający pobranie modelu aplikacji z warstwy biznesowej w postaci dużego złożonego obiektu TransferObject. Jeśli model aplikacji jest potrzebny w warstwie prezentacji, to w warstwie biznesowej następuje złożenie modelu aplikacji (poprzez połączenie wielu części modelu z obiektów Business Object, Data Access Object, Application Service) i przetransferowanie modelu aplikacji do warstwy prezentacji w postaci jednego obiektu transferowego.

Transfer Object

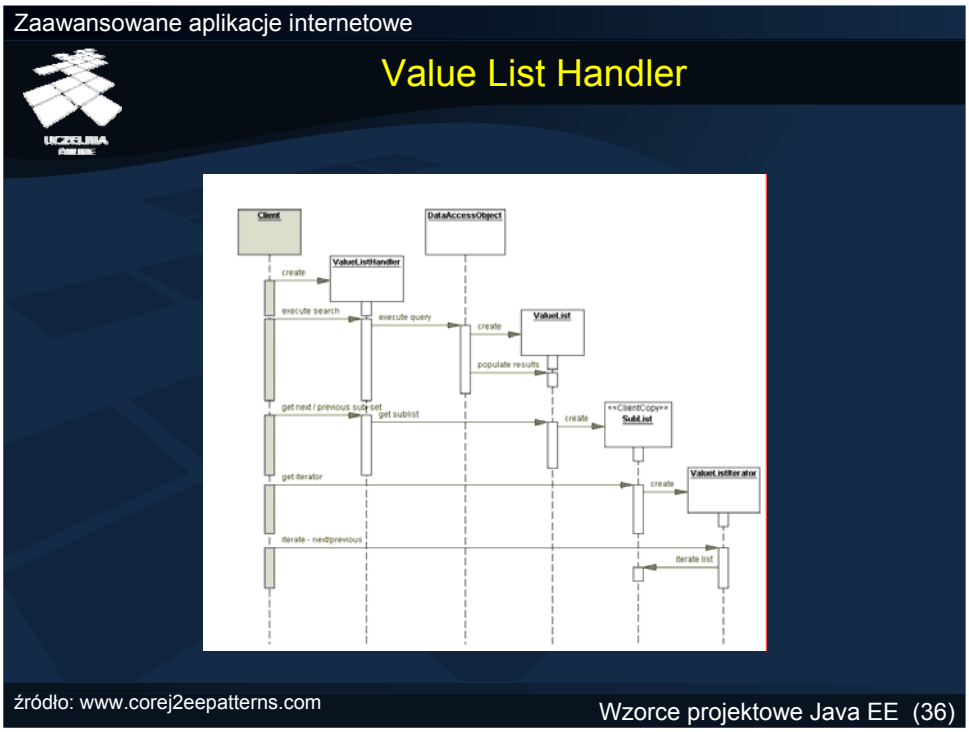


źródło: www.corej2eepatterns.com

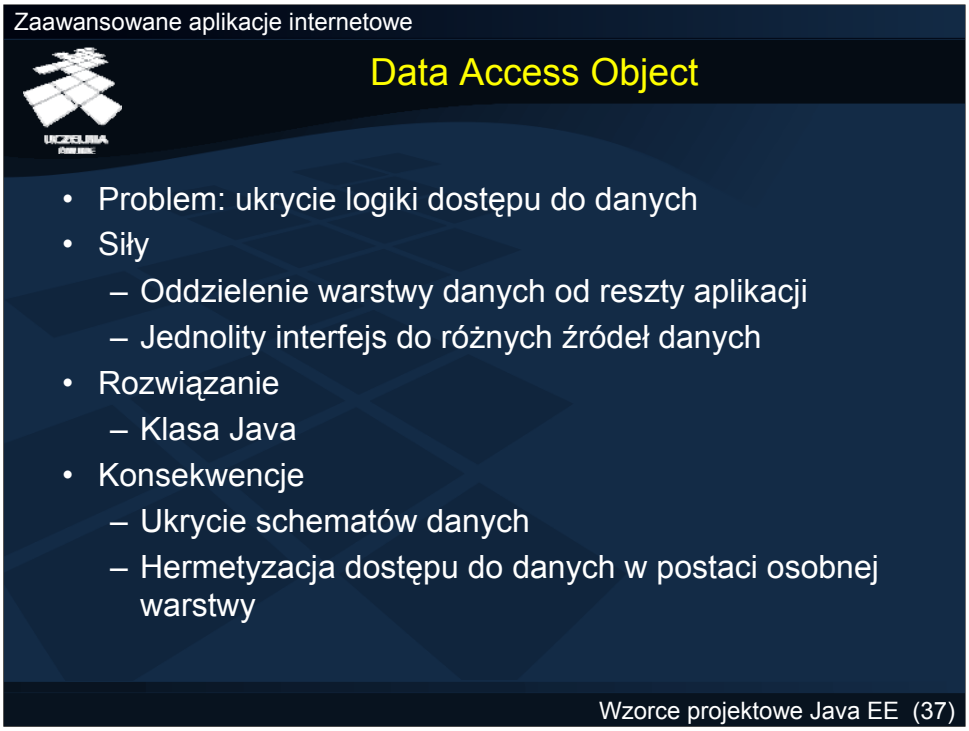
Slajd przedstawia diagram interakcji wzorca Transfer Object. Klient wykorzystuje obiekt Component do pobrania i wysłania danych. W tym przypadku klientem nie jest klient HTTP, lecz najczęściej inny komponent z innej warstwy aplikacji. Obiekt Component może być komponentem warstwy prezentacji, biznesowej lub integracji. Obiekt Component tworzy serializowalny obiekt TransferObject i wypełnia go danymi. Obiekt TransferObject jest następnie zwracany do wywołującego klienta, gdzie następuje pobranie danych. Komunikacja może także nastąpić w drugą stronę, klient może utworzyć nowy obiekt TransferObject, wypełnić go danymi, a następnie serializować i wysłać do komponentu innej warstwy, gdzie nastąpi pobranie przesłanych danych.



Większość aplikacji webowych zawiera element umożliwiający użytkownikowi wyszukiwanie i przeglądanie dużej liczby obiektów. Niebagatelne znaczenie ma efektywne zaimplementowanie mechanizmu wyszukiwania. Wzorzec Value List Handler proponuje rozwiązania umożliwiające pobieranie dużej liczby obiektów biznesowych w trybie "tylko do odczytu", bez konieczności wykorzystywania kontekstu transakcji. Dodatkowo, wzorzec zapewnia efektywność przetwarzania przez buforowanie wyników i tzw. paginację, czyli wyświetlanie wyników wyszukiwania w postaci niewielkich paczek. Wyniki wyszukiwania pozostają po stronie serwera i nie muszą być dzięki temu w całości konsumowane przez klienta. Wydatnie poprawia to wydajność sieci i zmniejsza jej obciążenie. Implementacje wzorca Value List Handler najczęściej korzystają ze zwykłych klas Java wykorzystujących predefiniowane obiekty dostępu do danych, np. java.sql.ResultSet i javax.sql.RowSet z JDBC lub implementacje interfejsu java.util.Collection (np. java.util.List). Zaletą wykorzystania gotowych klas jest fakt, że implementują one interfejs java.util.Iterator umożliwiający nawigację po znalezionych elementach i oprogramowanie paginacji.

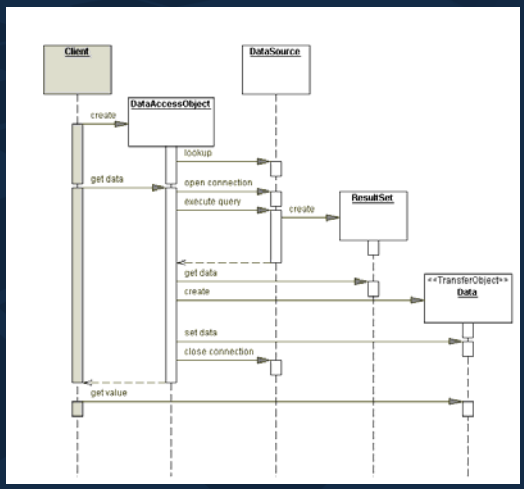


Slajd przedstawia diagram interakcji wzorca Value List Handler. Client to dowolny obiekt wykonujący wyszukiwanie, które zwraca duży zbór wyników. Obiekt Client tworzy obiekt ValueListHandler i przekazuje do niego zapytanie. Obiekt ValueListHandler tworzy nowy obiekt ValueList służący do przechowywania wyników wyszukiwania, a następnie, poprzez obiekt DataAccessObject, wykonuje zapytanie i umieszcza jego wynik w obiekcie ValueList. W trakcie konsumpcji wyników zapytania obiekt Client wysyła do obiektu ValueListHandler żądania pobrania kolejnego podzbioru wyników, które są realizowane w oparciu o zawartość obiektu ValueList. Możliwe jest też zawężenie kryteriów wyszukiwania i utworzenie przez obiekt ValueListHandler dodatkowej listy wyników, reprezentowanej przez obiekt SubList. Wreszcie, klient może zażądać iteracji po wynikach wyszukiwania. W takim przypadku obiekt ValueList zwraca pomocniczy obiekt ValueListIterator, który zapewnia metody pobierania poprzedniego/następnego elementu wyniku wyszukiwania.



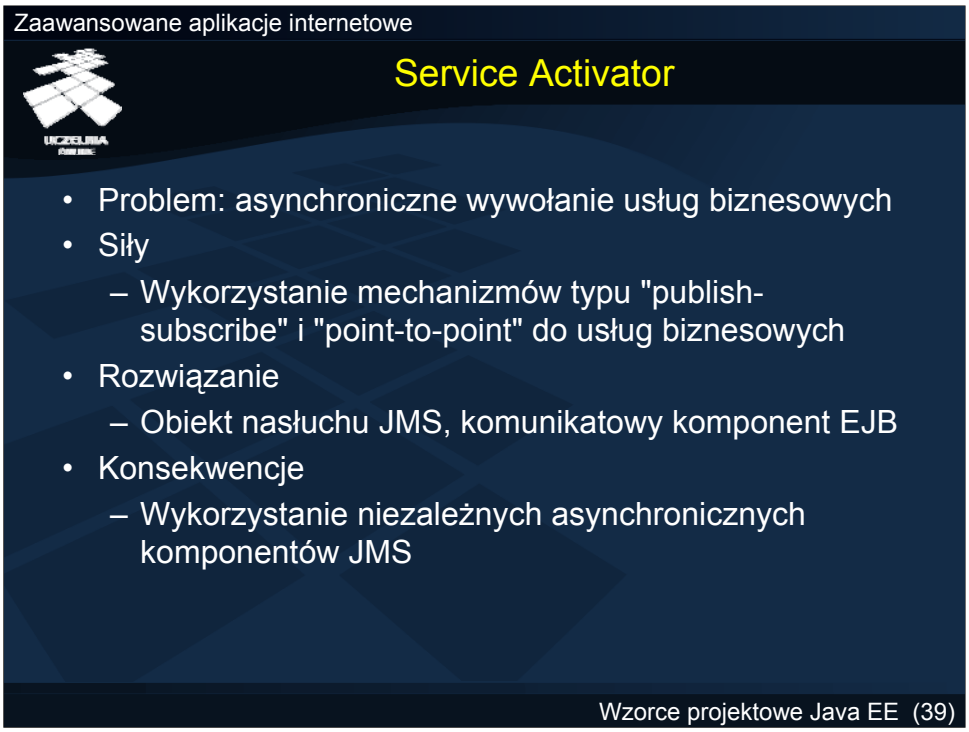
Aplikacje webowe czerpią swoje dane z wielu różnych repozytoriów. Najczęściej są to rzecz jasna relacyjne bazy danych, ale mogą to być również pliki tekstowe, repozytoria XML, obiektowe bazy danych, repozytoria LDAP, systemy integracyjne B2B, usługi autoryzacji kart kredytowych, i wiele innych. Każdy typ źródła danych posiada własny interfejs i specyfikę. Umieszczanie kodu dostępu do danych razem z logiką aplikacji wprowadza duże zamieszanie, wymusza obsługę wyjątków specyficznych dla dostępu do danych w kodzie aplikacji oraz silnie wiąże warstwy aplikacji. Zaleca się więc wprowadzanie dodatkowej warstwy hermetyzującej wszystkie aspekty dostępu do danych. Uzyskujemy dzięki temu wyraźną separację warstwy danych od reszty aplikacji oraz jednolity interfejs dostępu do danych przechowywanych w wielu różnych źródłach danych. Takie rozwiązanie promuje wzorzec projektowy Data Access Object. Najczęstszą implementacją wzorca Data Access Object jest pisana ręcznie klasa Java zawierająca kod dostępu do danych, choć niektóre architektury szkieletowe dostarczają gotowych implementacji wzorca Data Access Object, dobrym przykładem jest klasa JdbcTemplate z architektury Spring Framework. Dodatkową zaletą zastosowania wzorca Data Access Object jest ukrycie przed aplikacją schematów danych, co umożliwia łatwą i elastyczną pielęgnację i ewolucję schematów danych.

Data Access Object



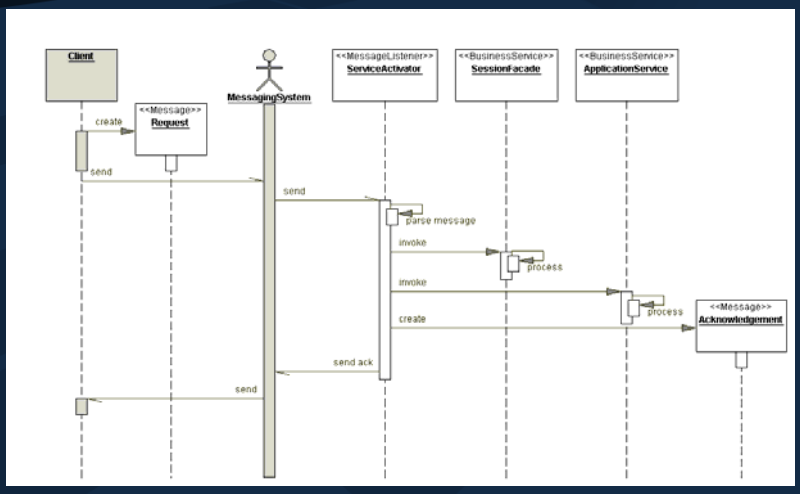
źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Data Access Object. Klient, którym może być dowolny komponent wymagający dostępu do danych, tworzy obiekt DataAccessObject związany z konkretnym źródłem danych, reprezentowanym przez obiekt DataSource. Żądanie pobrania danych pochodzące od klienta jest realizowane przez obiekt DataAccessObject w kilku etapach. W pierwszym kroku następuje nawiązanie połączenia ze źródłem danych, następnie do źródła danych zostaje przesłane zapytanie a jego wynik zostaje zapisany w postaci obiektu ResultSet. Przesłanie danych między obiektem DataAccessObject i źródłem danych jest realizowane zgodnie ze wzorcem Transfer Object. Obiekt DataAccessObject tworzy nowy obiekt transferowy TransferObject i wypełnia go wynikami zapytania pobranymi z obiektu ResultSet. DataAccessObject zamyka połączenie ze źródłem danych i przekazuje obiekt transferowy z powrotem do klienta.



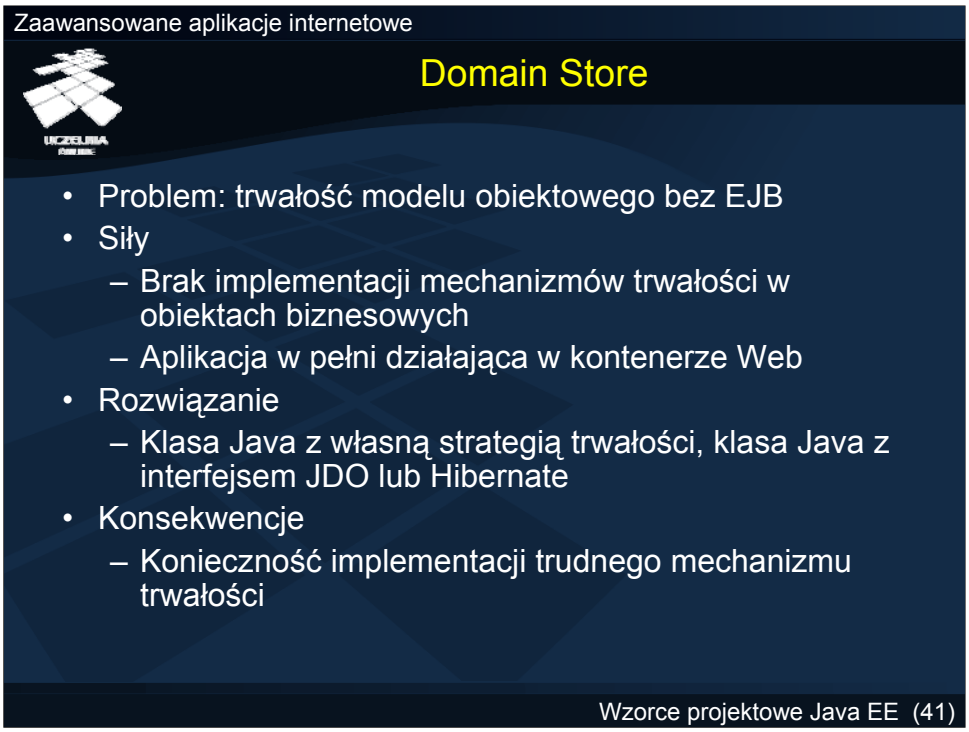
W aplikacjach webowych większość wywołań usług i logiki biznesowej ma charakter synchroniczny: komponent wywołuje usługę i czeka na zwrócenie wyniku działania usługi. Istnieją jednak usługi biznesowe, które w naturalny sposób muszą być wywoływane w sposób asynchroniczny. Dotyczy to przede wszystkim usług zależnych od komponentów zewnętrznych względem całej aplikacji. Dwa popularne mechanizmy działania asynchronicznego obejmują model "publish-subscribe" (odbiorcy i nadawcy asynchronicznych komunikatów współdzielą listy tematów, do których mogą być publikowane komunikaty, np. wywołania usług biznesowych) oraz "point-to-point" (nadawca przekazuje komunikat tylko jednemu odbiorcy). Wzorzec Service Activator ułatwia włączenie komunikacji asynchronicznej do ogólnej architektury aplikacji. Na platformie Java EE istnieją dwie popularne implementacje komunikacji asynchronicznej: interfejs JMS (Java Messaging Service) oraz komunikatowe komponenty EJB. Obie implementacje oferują podobną funkcjonalność, przy czym przygotowanie własnej klasy zdolnej do prowadzenia nasłuchu JMS jest preferowane w przypadku mniejszych i prostszych systemów. Główną konsekwencją zastosowania wzorca Service Activator jest integracja JMS z aplikacją i umożliwienie komunikacji asynchronicznej.

Service Activator

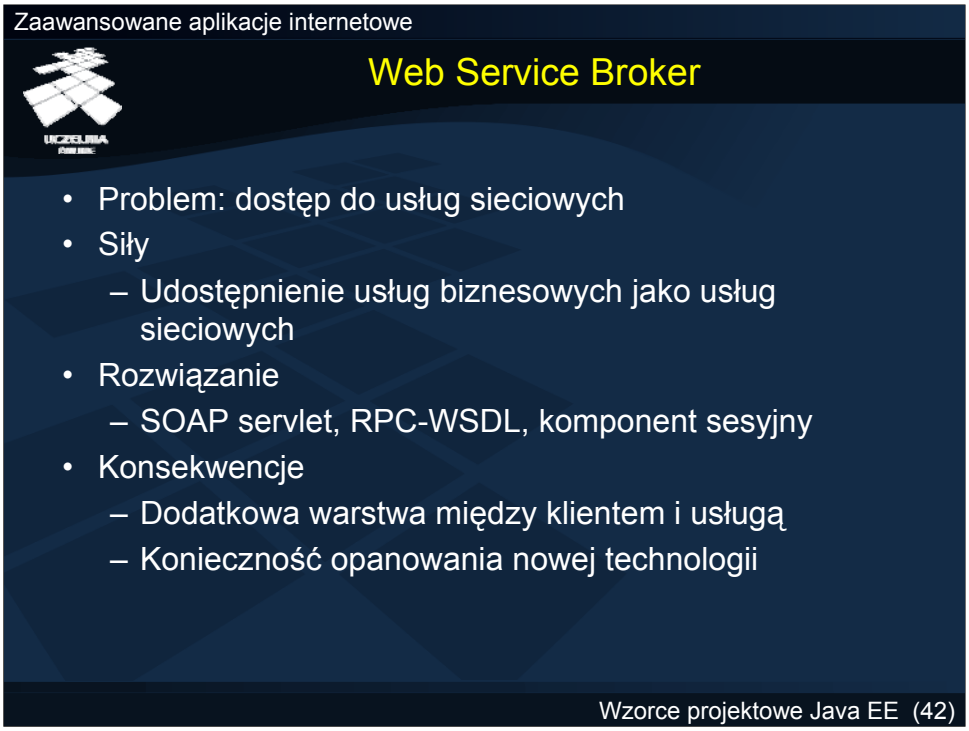


źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Service Activator. Obiekt Client reprezentuje dowolny komponent aplikacji, który musi wywołać usługę biznesową w sposób asynchroniczny. Żądanie wykonania usługi biznesowej jest przesyłane do systemu przesyłania komunikatów (Messaging Service) i dalej wysyłane do obiektu ServiceActivator, który nasłuchuje nadejścia komunikatów przeznaczonych dla siebie. Obiekt ServiceActivator analizuje uzyskane żądanie i wywołuje właściwą usługę biznesową albo za pomocą fasady (obiekt SessionFacade), albo poprzez obiekt ApplicationService stanowiący hermetyzację i obiektową reprezentację usługi. Możliwe jest, że obiekt ServiceActivator musi przesłać klientowi potwierdzenie wykonania usługi biznesowej (potwierdzenie jest reprezentowane przez obiekt Acknowledgement). Takie potwierdzenie jest przesyłane zwrotnie do klienta także w sposób asynchroniczny.

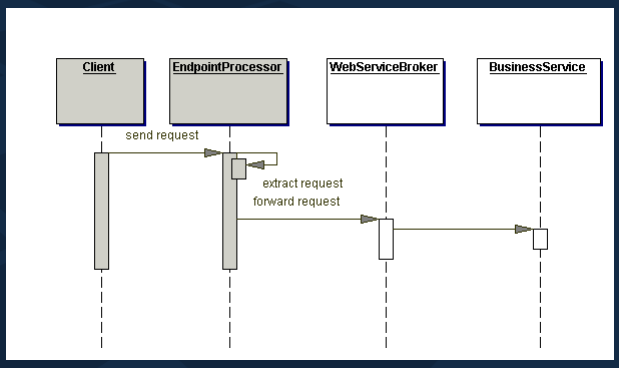


Technologia encyjnych komponentów EJB spotkała się z bardzo dużą krytyką środowiska programistycznego, przede wszystkim ze względu na nieefektywność przetwarzania, wysoki stopień skomplikowania, trudność w administracji i strojeniu. Wiele projektów aplikacji webowych rezygnuje z wykorzystywania encyjnych komponentów EJB w warstwie obiektowego modelu biznesowego na rzecz prostych klas Java. Pojawia się jednak wówczas problem zapewniania trwałości obiektom biznesowym. W optymalnym scenariuszu zapewnianie trwałości obiektom biznesowym powinno być transparentne i oddzielone od logiki biznesowej. Celem wzorca Domain Store jest zapewnienie trwałości obiektom biznesowym bez korzystania z kontenera EJB. Implementacja wzorca Domain Store może być zrealizowana na dwa sposoby. Pierwszy sposób polega na przygotowaniu prostych klas Java z własną strategią trwałości. Takie rozwiązanie nadaje się tylko dla bardzo prostych i niewielkich modeli obiektowych, gdyż w ogólności przygotowanie efektywnego, wydajnego i pozbawionego błędów mechanizmu trwałości jest zadaniem bardzo trudnym i pracochłonnym. Zdecydowanie preferowane jest rozwiązanie polegające na wykorzystaniu prostych klas Java wspomaganych przez gotowe systemy trwałości, np. Hibernate lub JDO (Java Data Objects).



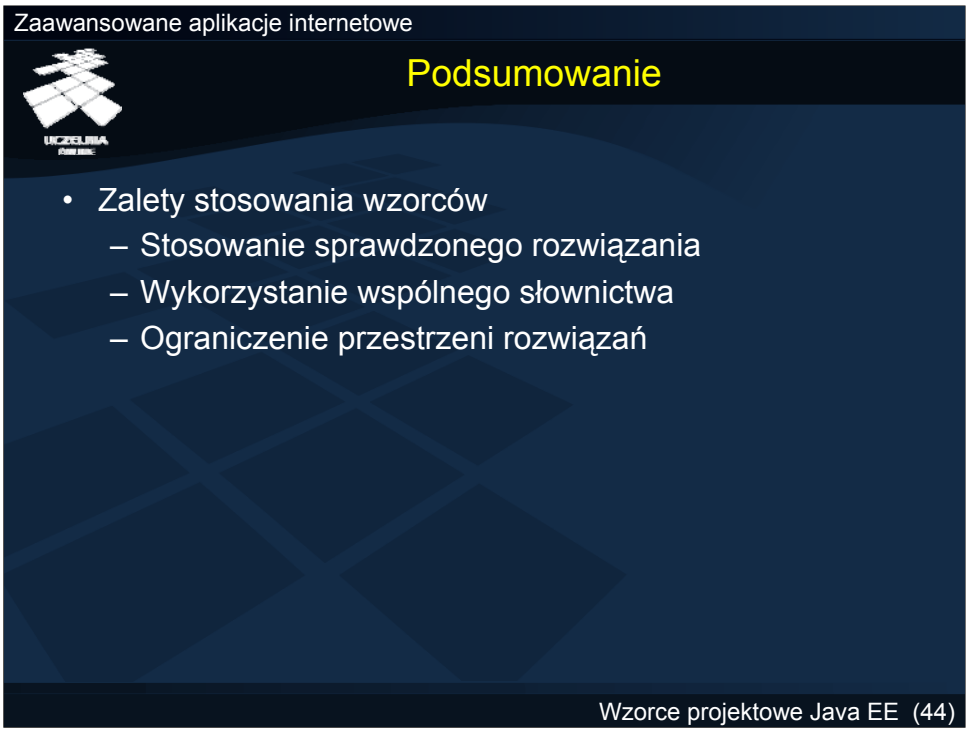
Technologia usług sieciowych (ang. Web services) cieszy się ostatnio bardzo dużym powodzeniem. Umożliwia ona udostępnianie usług biznesowych klientom zewnętrznym przy wykorzystaniu środowiska heterogenicznego i otwartych protokołów (XML, HTTP, SOAP). Wzorzec Web Service Broker umożliwia włączenie technologii usług sieciowych do aplikacji i udostępnienie wybranych usług biznesowych na zewnątrz, w postaci usług sieciowych. Umożliwia dzięki temu integrację heterogenicznych systemów i tworzenie aplikacji, w których klienci są powiązani z warstwą biznesową w sposób bardzo luźny. Implementacja wzorca Web Service Broker jest złożona i zależy przede wszystkim od rodzaju wykorzystywanej usługi sieciowej. Jako przykłady implementacji można podać serwlet SOAP, strategię zdalnego wołania metod RPC opartą na rejestrach UDDI i deskryptorach usług sieciowych WSDL, wreszcie implementacją wzorca może być sesyjny komponent EJB funkcjonujący jako końcówka (endpoint) usługi sieciowej. Szczegółowy opis każdej z możliwości wykracza zdecydowanie poza ramy niniejszego wykładu. Wzorzec Web Service Broker znajduje zastosowanie w dużych, złożonych systemach, w przypadku niewielkich aplikacji narzut związany z dodaniem nowej warstwy między klientem i usługą oraz konieczność opanowania nowej, złożonej technologii zdecydowanie przeważa ewentualne korzyści płynące z implementacji wzorca.

Web Service Broker



źródło: www.corej2eepatterns.com

Slajd przedstawia diagram interakcji wzorca Web Service Broker w najprostszej strategii implementacji. Zewnętrzny heterogeniczny klient przesyła żądanie do końcówki, którą jest najczęściej bezstanowy komponent sesyjny. Komponent analizuje żądanie, dokonuje transformacji i ekstrakcji parametrów żądania (zostają przysłane w postaci koperty SOAP, pliku XML zawierającego żądanie wykonania usługi) i przekazuje parametry do obiektu WebServiceBroker. Obiekt ten lokalizuje właściwą usługę biznesową i wywołuje tę usługę, a wynik wykonania przekazuje z powrotem do klienta, uprzednio konstruując odpowiedź w postaci koperty SOAP (tym zadaniem znów zajmuje się obiekt EndpointProcessor).



Korzystanie z katalogu wzorców projektowych niesie ze sobą wiele zalet. Po pierwsze, osoba projektująca aplikację ma pewność, że wykorzystuje sprawdzone i optymalne rozwiązanie. Zapewnienie najlepszego dopasowania wzorca do konkretnego problemu daje wybór właściwej strategii implementacji wzorca, natomiast sam wybór wzorca wskazuje na zbiór poprawnych rozwiązań problemu. W większych zespołach programistycznych wykorzystywanie katalogu wzorców ułatwia komunikację między członkami zespołu poprzez uspójnienie wykorzystywanej terminologii. Wpływa to pozytywnie na wydajność zespołu, jakość komunikacji i współpracy, pozwala na uniknięcie wielu nieporozumień. Wreszcie wzorce definiują jasno przestrzeń możliwych rozwiązań i wskazują na właściwe kierunki implementacji. Pozwala to zaoszczędzić wiele osobogodzin pracy programistycznej, która mogłaby prowadzić donikąd.

Materiały dodatkowe

- "Core J2EE Wzorce projektowe", D.Alur, J.Crupi, D.Malks, ISBN 83-7361-344-7, Helion 2004
- Sun Developer Network, http://sun.java.com/blueprints/patterns
- Core J2EE Patterns, http://www.corej2eepatterns.com