Warstwa biznesowa

(wg. D.Alur, J.Crupi, D. Malks, Core J2EE. Wzorce projektowe.)

1. Definicja warstwy biznesowej
2. Podstawowe przypadki - analiza podstawowych przypadków

Pięciowarstwowy model logicznego rozdzielania zadań (wg. D.Alur, J.Crupi, D. Malks, Core J2EE. Wzorce projektowe.)

Warstwa klienta
Klienci aplikacji, aplety, aplikacje i inne elementy z graficznym interfejsem użytkownika

Interakcja z użytkownikiem, urządzenia i prezentacja interfejsu użytkownika

Warstwa prezentacji
Strony JSP, serwlety i inne elementy interfejsu użytkownika

Logowanie, zarządzanie sesją, tworzenie zawartości, formatowania i dostarczanie

Warstwa biznesowa
Komponenty EJB i inne obiekty biznesowe

Logika biznesowa, transakcje, dane i usługi

Warstwa integracji
JMS, JDBC, konektory i połączenia z systemami zewnetrznymi

Adaptery zasobów, systemy zewnętrzne, mechanizmy zasobów, przepływ sterowania

Warstwa zasobów
Bazy danych, systemy zewnętrzne i pozostałe zasoby

Zasoby, dane i usługi zewnętrzne

Problem 1 – ukrycie przed klientem złożoności zdalnej komunikacji z komponentem usług biznesowych

Uwagi:

1. Kiedy klienci bezpośrednio komunikują się z komponentami usług biznesowych, muszą zmieniać swój kod wraz ze zmianami kodu usług biznesowych. Zwiększa się nakład pracy nad systemem.
2. Kiedy klienci bezpośrednio komunikują się z komponentami usług biznesowych, jedno działanie klienta może wymagać wielu odwołań do warstwy biznesowej przez sieć.
3. Kiedy klienci bezpośrednio komunikują się z komponentami usług biznesowych, istnieje konieczność połączenia kodu klienta z kodem związanym z infrastrukturą aplikacji, czyli takich zagadnień zdalnej komunikacji jak: usługa JNDI (*Java Naming and Directory Interface – poszukiwanie obiektów związanych z poszukiwanymi aplikacjami za pomocą nazw tych obiektów*), obsługa błędów sieciowych oraz logika ponawiania wywołań usług biznesowych

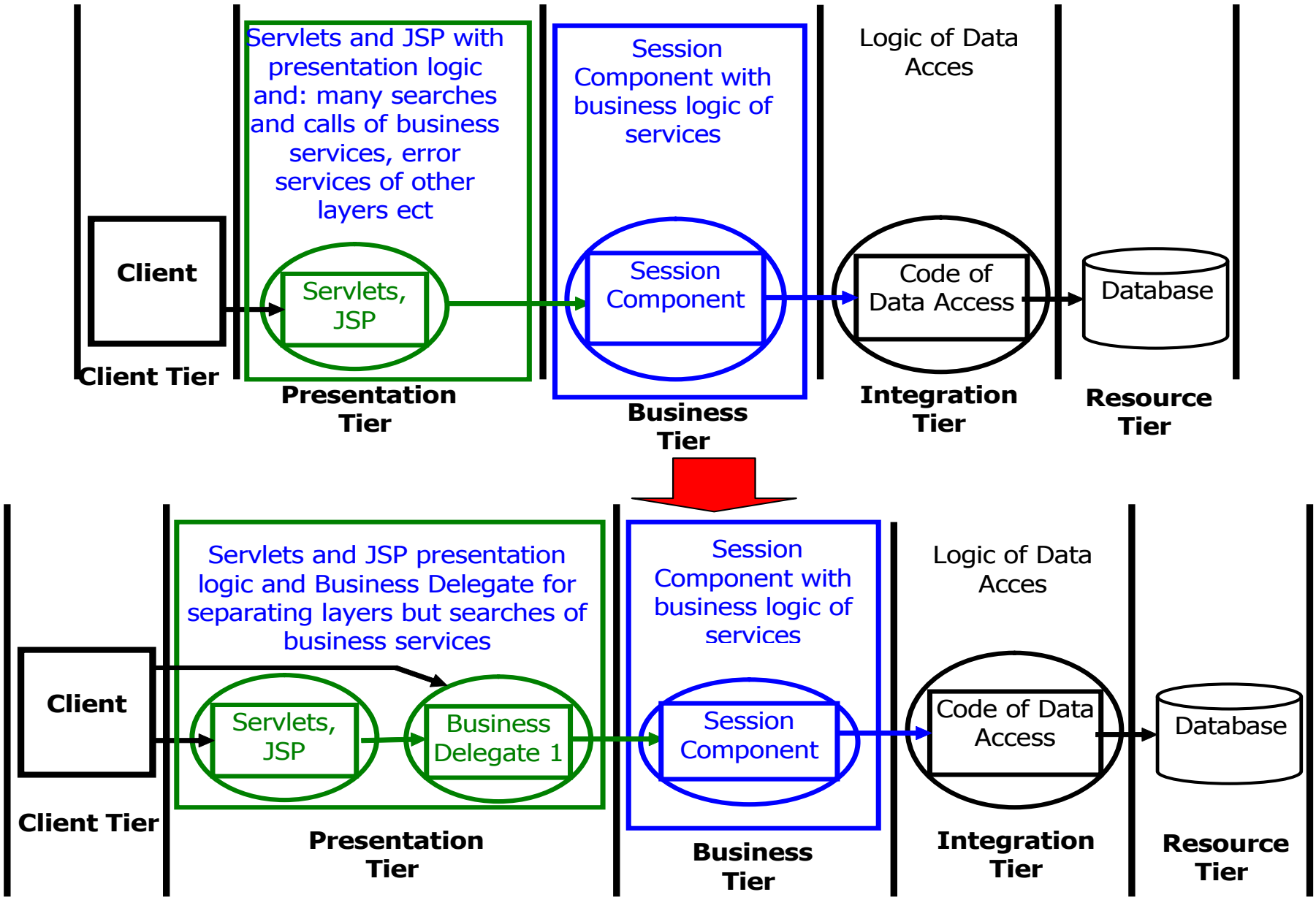
Wymagania:

1. Należy zapewnić dostęp do komponentów warstwy biznesowej z poziomu komponentów prezentacji i klientów, którymi mogą być dowolne urządzenia, usługi internetowe i rozbudowane aplikacje klienckie
2. Należy zminimalizować powiązanie klientów i usług biznesowych ukrywając szczegóły implementacji usług, na przykład ich wyszukiwanie i wywoływanie.
3. Należy uniknąć niepotrzebnych wywołań usług biznesowych
4. Należy zamieniać wyjątki sieci i usług na wyjątki aplikacji lub użytkownika
5. Należy ukryć szczegóły wykorzystania usług, ich konfiguracji i ponawiania prób wywoływania przed klientami

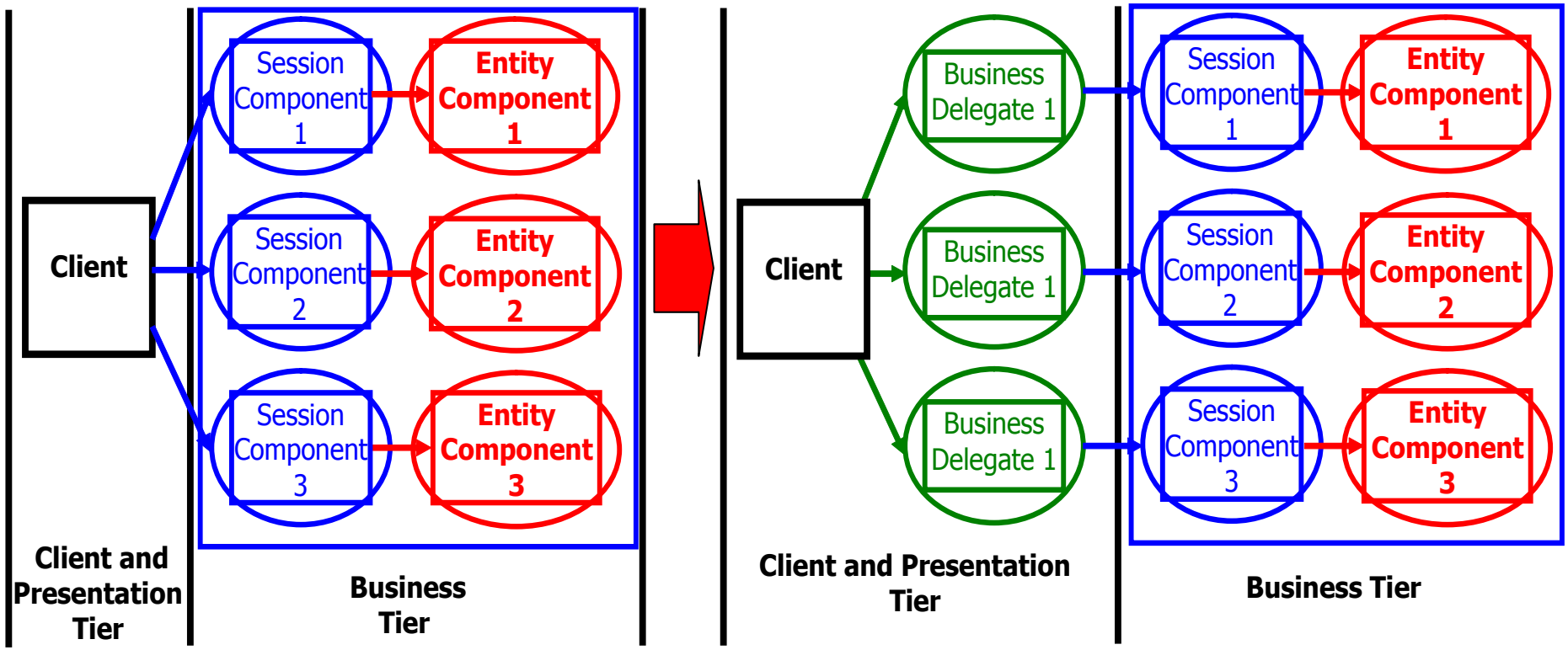
Wzorzec BusinessDelegate: abstrakcyjna usługa biznesowa po stronie klienta pozwalająca ukrywać szczegóły implementacji usług biznesowych – **typ"Control"**

1. Ukrycie szczegółów usługi biznesowej np. klient nie musi znać nazwy usługi i wyszukiwania usługi **(1)**
2. Obsługa wyjątków obiektów z warstwy biznesowej i przekazanie wyjątków, które są łatwiejsze do obsłużenia przez warstwę prezentacji
3. W razie przejściowych nieprawidłowości wywołania usług biznesowych mogą zachodzić wielokrotnie, bez udziału pozostałych części warstwy prezentacji. Dopiero, gdy nastąpi trwała awaria systemu, klient zostanie powiadomiony o awarii.
4. Możliwość buforowania wyników i referencji do zdalnych usług, co ogranicza ruch w sieci i zwiększa wydajność aplikacji **(2)**

Wzorzec Business Delegate



Wzorzec Business Delegate



Diagramy klas wzorca **Business Delegate**

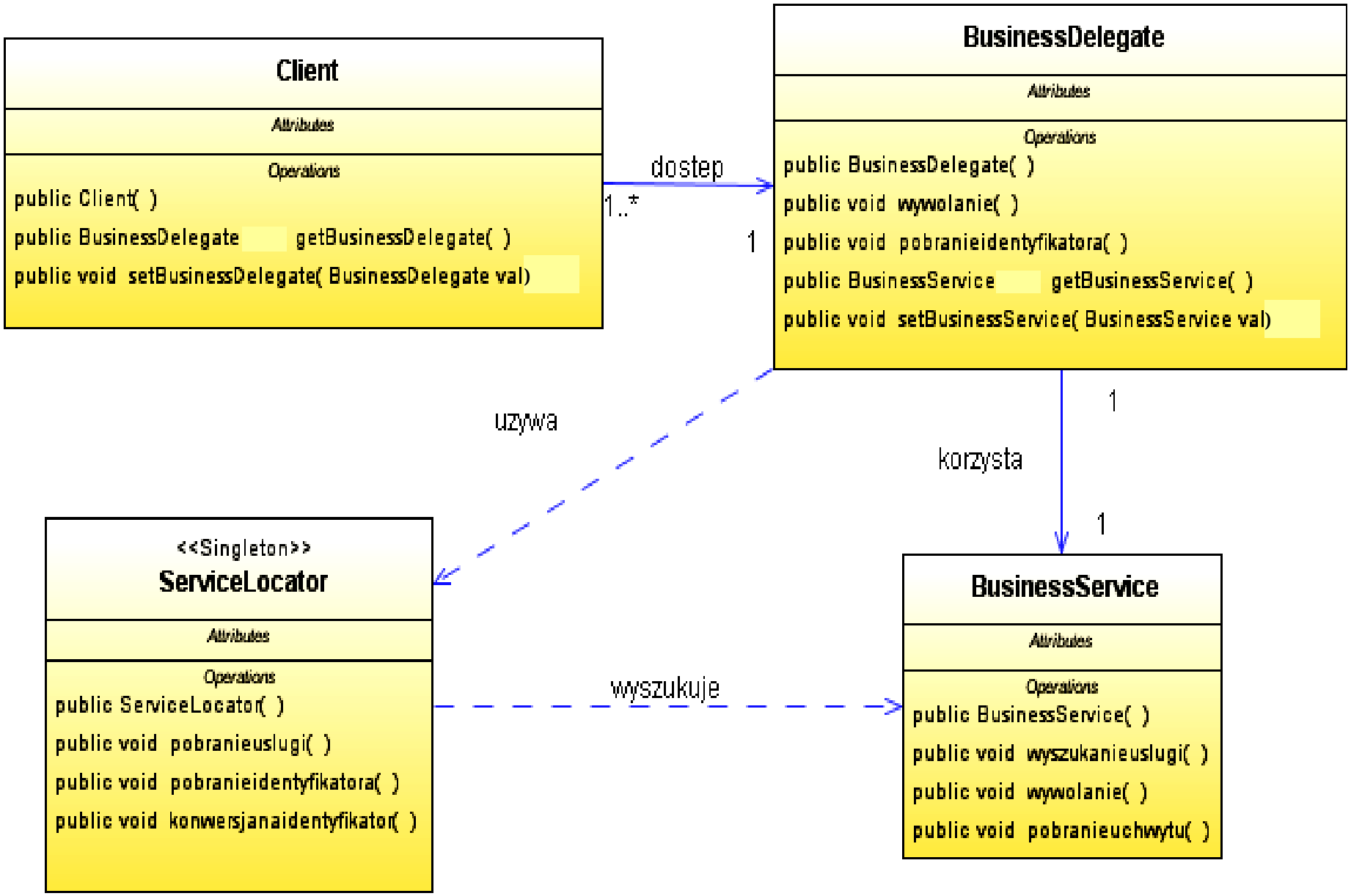


Diagram sekwencji wzorca BusinessDelegate (1) – pobranie i wywołanie usługi biznesowej

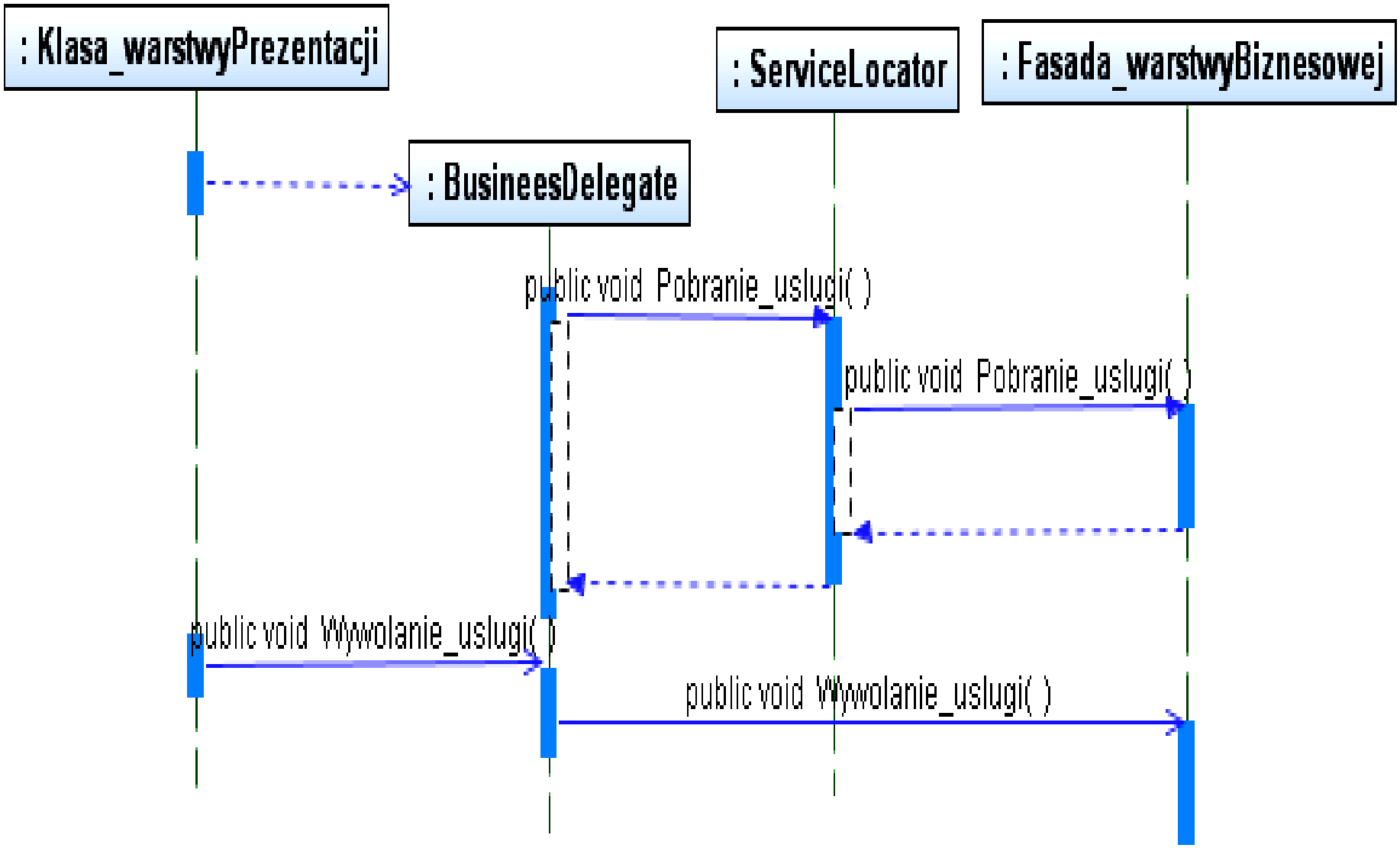
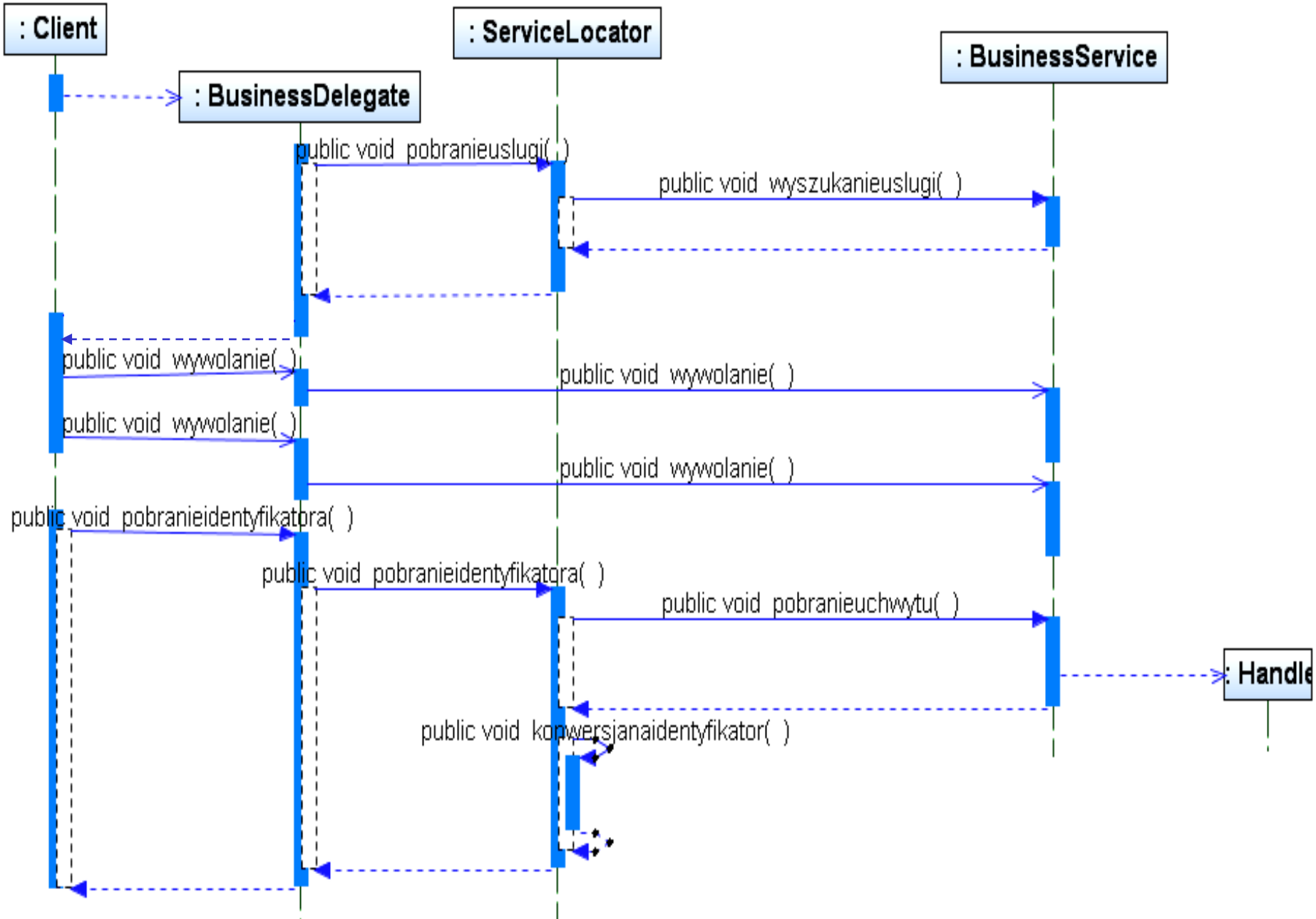


Diagram sekwencji wzorca BusinessDelegate (2) – pobranie i wywołanie usługi biznesowej oraz zapamiętanie jej uchwytu



Problem 2 – przezroczysty i jednolity sposób wyszukiwania usług i komponentów biznesowych

Uwagi:

1. Klienci muszą wyszukiwać i uzyskać dostęp do komponentów i usług warstwy biznesowej.
2. W aplikacjach J2EE komponenty warstwy biznesowej, na przykład komponenty EJB i komponenty warstwy integracji (źródła danych JDBC, komponenty JMS) są rejestrowane w centralnym rejestrze. Klient wykorzystuje interfejs JNDI, aby pobrać obiekt InitialContext, który odwzorowuje nazwy na referencję do obiektów. Kod JNDI nie powinien być powielany w aplikacjach klienckich, ponieważ pogarsza to wydajność systemu
3. Kod JNDI zależy od konkretnej implementacji oprogramowania serwera, dlatego aplikacja klienta powinna być uniezależniona od kodu JNDI.

Wymagania:

1. Należy wykorzystywać interfejs JNDI w celu wyszukiwania komponentów biznesowych (na przykład komponentów EJB lub JMS) lub usług (na przykład źródeł danych) **(1)**
2. Należy zcentralizować i w wielu miejscach aplikacji klienckich wykorzystywać mechanizmy wyszukiwania.
3. Należy przed klientem ukryć szczegóły implementacji rejestrów oraz złożoność korzystania z nich
4. Należy uniknąć straty wydajności związanej z tworzeniem kontekstu i wyszukiwaniem usług
5. Należy pozwolić na ponowne wykorzystanie istniejących instancji komponentów np.. EJB, używając związanych z nimi obiektów Handle

Wzorzec ServiceLocator: pozwala ukryć szczegóły wyszukiwania usług i komponentów oraz szczegóły oprogramowania – **typ „control”**

• **Client** – może być nim obiekt typu **Business Delegate**, który oczekuje dostepu do komponentów **Session Facade**. Podobnie obiekt typu **DataAccess Object** jest klientem podczas pobierania źródła danych **JDBC**.

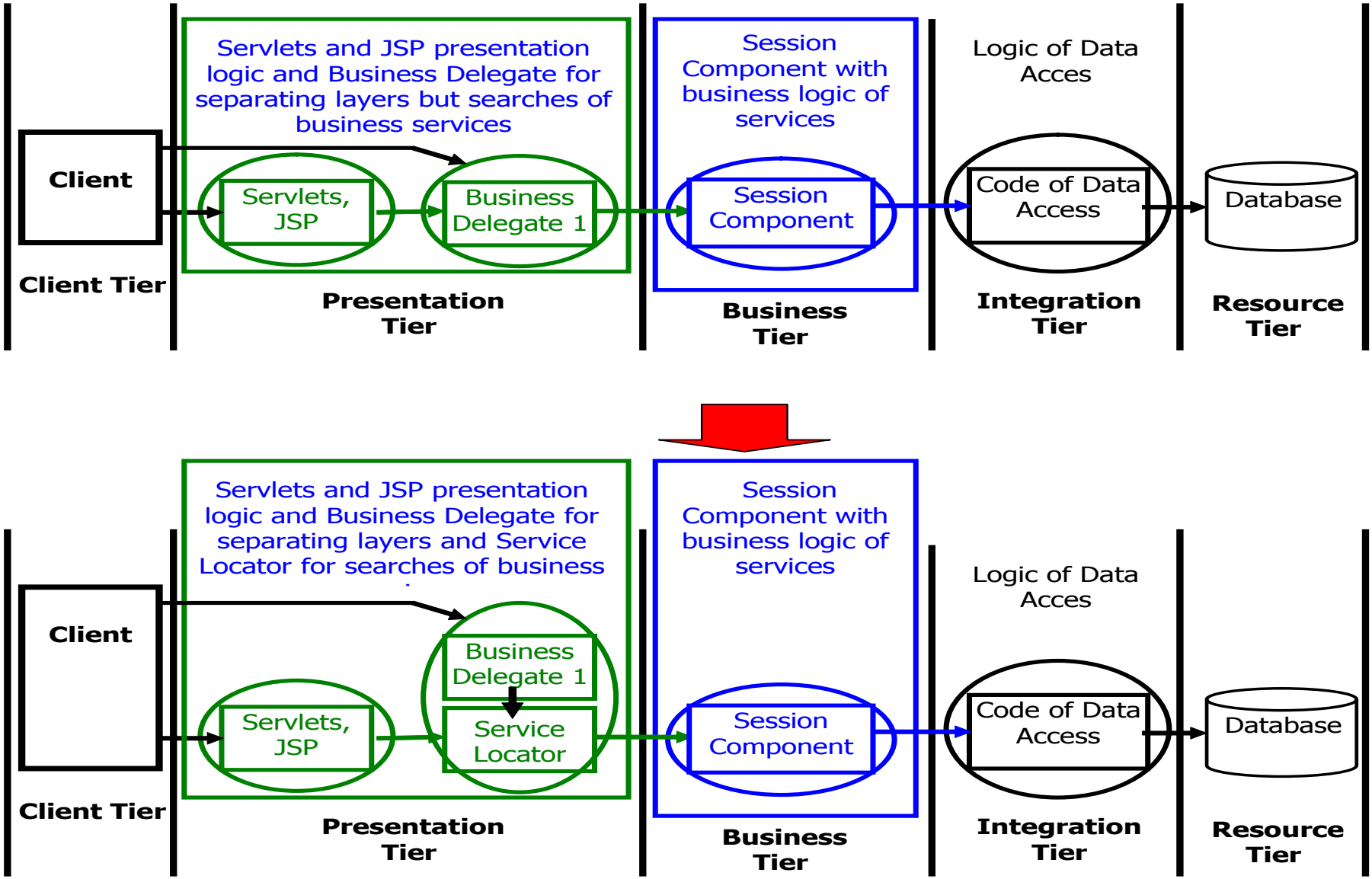
• **Cache** – buforowanie referencji do odszukanych wcześniej usług w celu ograniczenia niepotrzebnych operacji, czyli poprawy wydajności

• **Initial Context** – punkt startowy procesu wyszukiwania i tworzenia obiektów. Dostawcy usług udostępniają obiekt kontekstu zależny od wyszukiwanej usługi (obiekt Target). Każdy dostawca specjalizuje się w typach usług (EJB, JMS)

• **Target** – reprezentuje docelową usługę lub komponent z warstwy biznesowej lub integracji. Mogą to być obiekty typu **EJBHome** dla komponentów **EJB, DataSource** dla źródła danych **JDBC**, obiekt **ConnectionFactory** (**TopicConnectionFactory** dla modelu komunikatów publikacja-subskrypcja lub **QuenneConnectionFactory** dla komunikatów typu punkt-punkt) dla **JMS** (**Java Message Service** – pozwala komponentom Java EE tworzyć, wysyłać i czytać wiadomości w rozproszonym środowisku sterując długimi transakcjami)

• **RegistryService** – obiekt ten reprezentuje implementację rejestru przechowującego referencje do usług lub komponentów zarejestrowanych jako dostawcy usług dla obiektów Client)

Service Locator



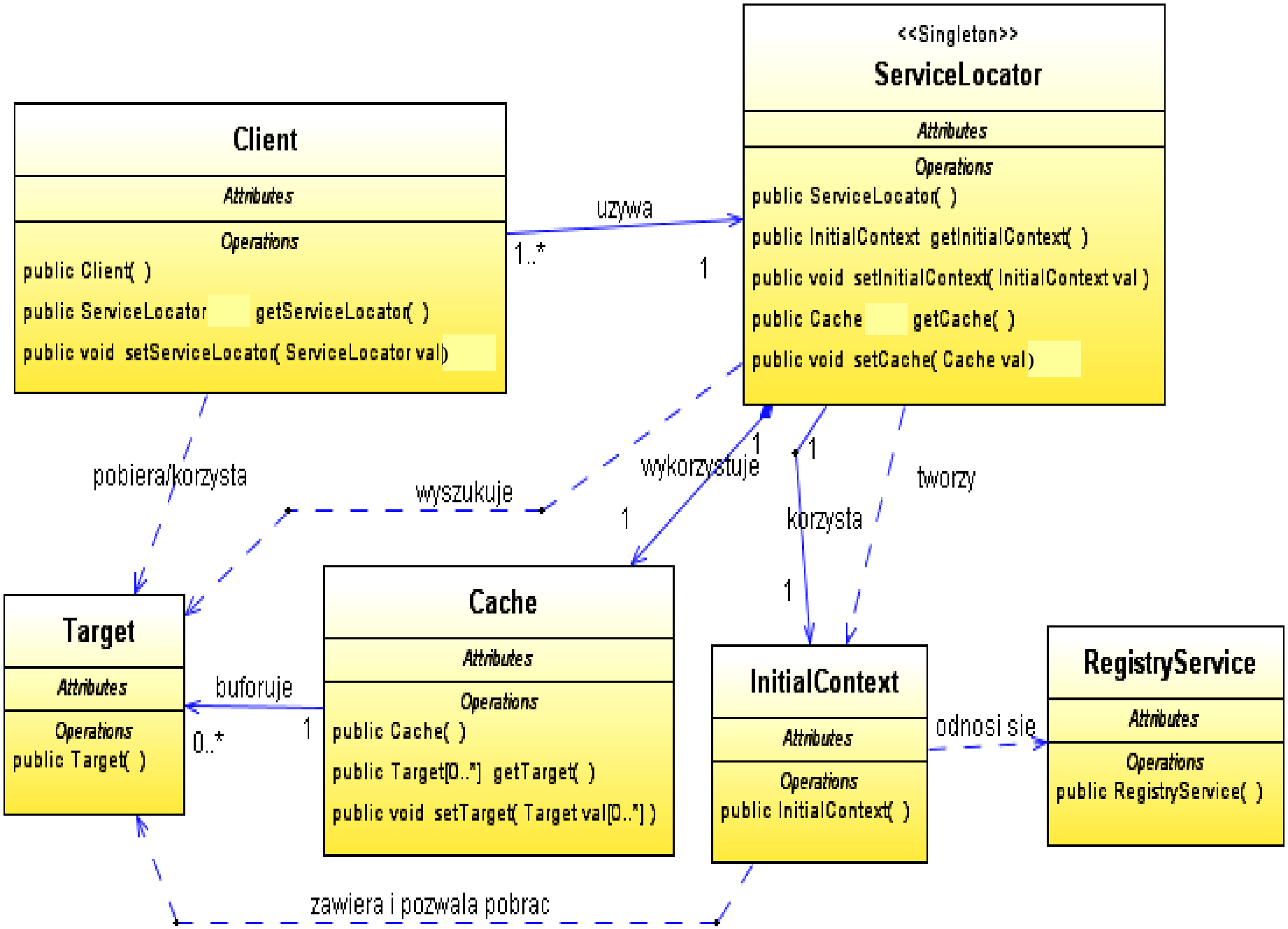
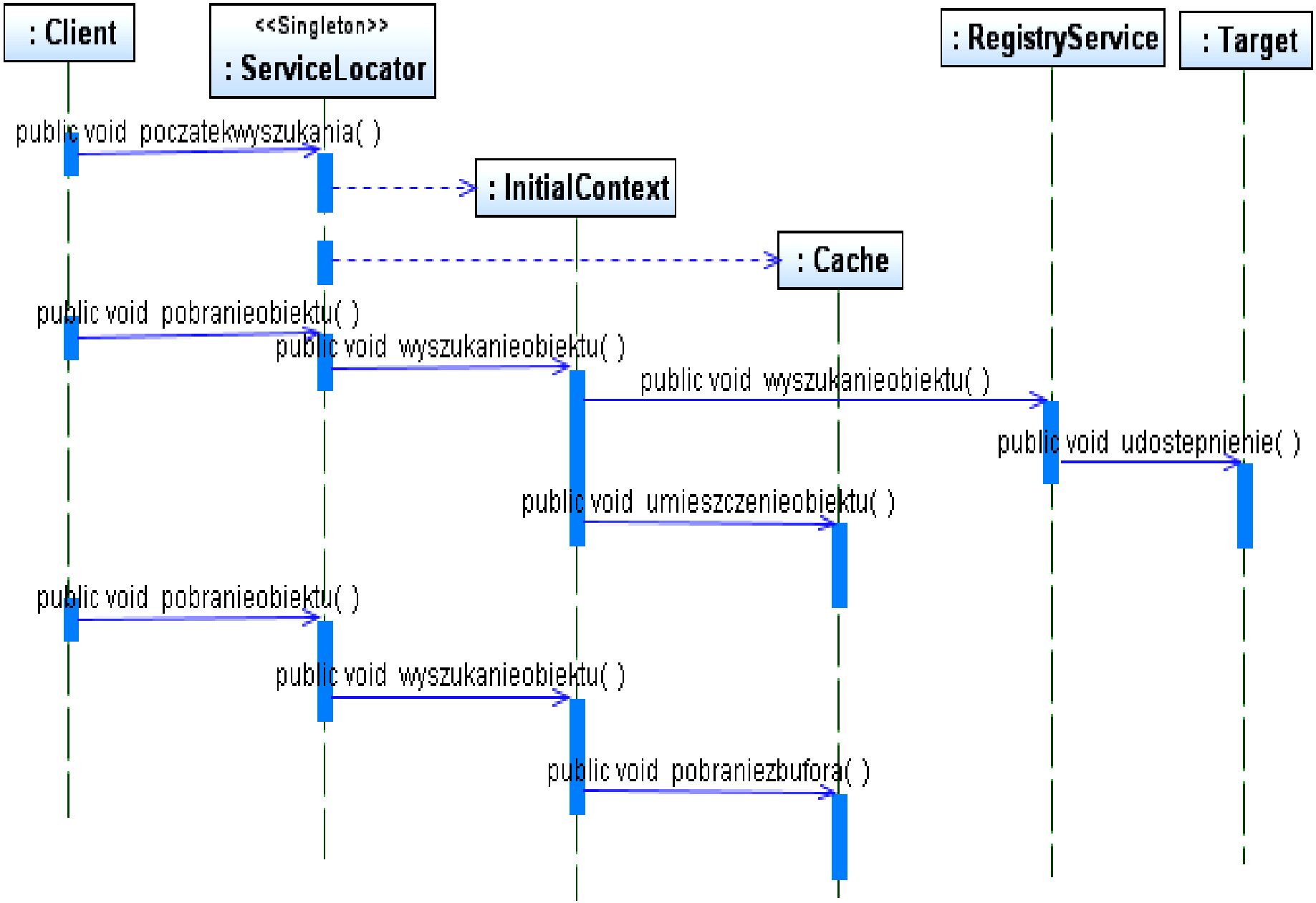

Client
Attributes
Operations
public Client( )
public ServiceLocator getServiceLocator( )
public void setServiceLocator(ServiceLocator val)
<<Singleton>>
ServiceLocator
Attributes
Operations
public ServiceLocator( )
public InitialContext getInitialContext( )
public void setInitialContext(InitialContext val)
public Cache getCache( )
public void setCache(Cache val)
używa
1..*
1
pobiera/korzysta
wyszukuje
wykorzystuje
1
1
1
1
tworzy
korzysta
1
Target
Attributes
Operations
public Target( )
buforuje
0..*
1
Cache
Attributes
Operations
public Cache( )
public Target[0..*] getTarget( )
public void setTarget(Target val[0..*])
InitialContext
Attributes
Operations
public InitialContext( )
odnosi się
RegistryService
Attributes
Operations
public RegistryService( )
zawiera i pozwala pobrać

Diagram sekwencji wzorca ServiceLocator (1) – wyszukiwanie usług biznesowych komponentów



Problem 3 – udostępnianie komponentów i usług biznesowych zdalnym klientom (przejęcie kontroli do obiektów biznesowych i ograniczenie ruchu w sieci, czyli poprawa wydajności)

Uwagi:

1. Model obiektowy warstwy biznesowej jest realizowany po stronie serwera w postaci obiektów *Business Object* **(problem 5)**, zwykłych obiektów Javy (POJO) lub komponentów Entity – wszystkie typu „Entity”.
   - Kiedy klienci bezpośrednio komunikują się z komponentami usług biznesowych, zmiana interfejsu komponentu biznesowego ma bezpośredni wpływ na kod klienta.
   - Bezpośredni dostęp wymaga, by klient zawierał złożoną logikę zajmującą się koordynacją i interakcją pomiędzy wieloma komponentami biznesowymi powiązanymi złożonymi relacjami. Wtedy kod klienta zawiera złożone operacje wyszukiwania, zarządzania transakcjami, bezpieczeństwem i sam przeprowadzać przetwarzanie biznesowe.
   - Kiedy istniej wiele typów klientów, bezpośredni dostęp do komponentów biznesowych prowadzi do nierównomiernego korzystania z usług i powielania kodu u klientów. Utrudnia to pielęgnację kodu klienta i zmniejsza elastyczność jego zastosowania.
2. Rozdrobnienie komponentów na jedną usługę biznesową prowadzi do wielu komunikacji w sieci z powodu zdalnych wywołań do rozdrobnionych komponentów

Wymagania:

1. Należy uniknąć bezpośredniego udostępnienia klientom komponentów warstwy biznesowej, aby nie dopuścić do powstania zbyt wielu zależności między klientami a warstwą biznesową **(1)**
2. Należy zapewnić zdalną warstwę dostępu do obiektów **Business Object (problem 5)** lub innych obiektów warstwy biznesowej
3. Należy zgrupować i udostępnić zdalnym klientom usługi aplikacji (implementować wzorzec **Application Service**) oraz dowolne inne usługi
4. Należy scentralizować i połączyć całą logikę biznesową udostępnianą zdalnym klientom
5. Należy ukryć złożone interakcje i wzajemne zależności między komponentami i usługami biznesowymi w celu ułatwienia zarządzania, centralizacji logiki, zwiększenia elastyczności i ułatwienia wprowadzania zmian.

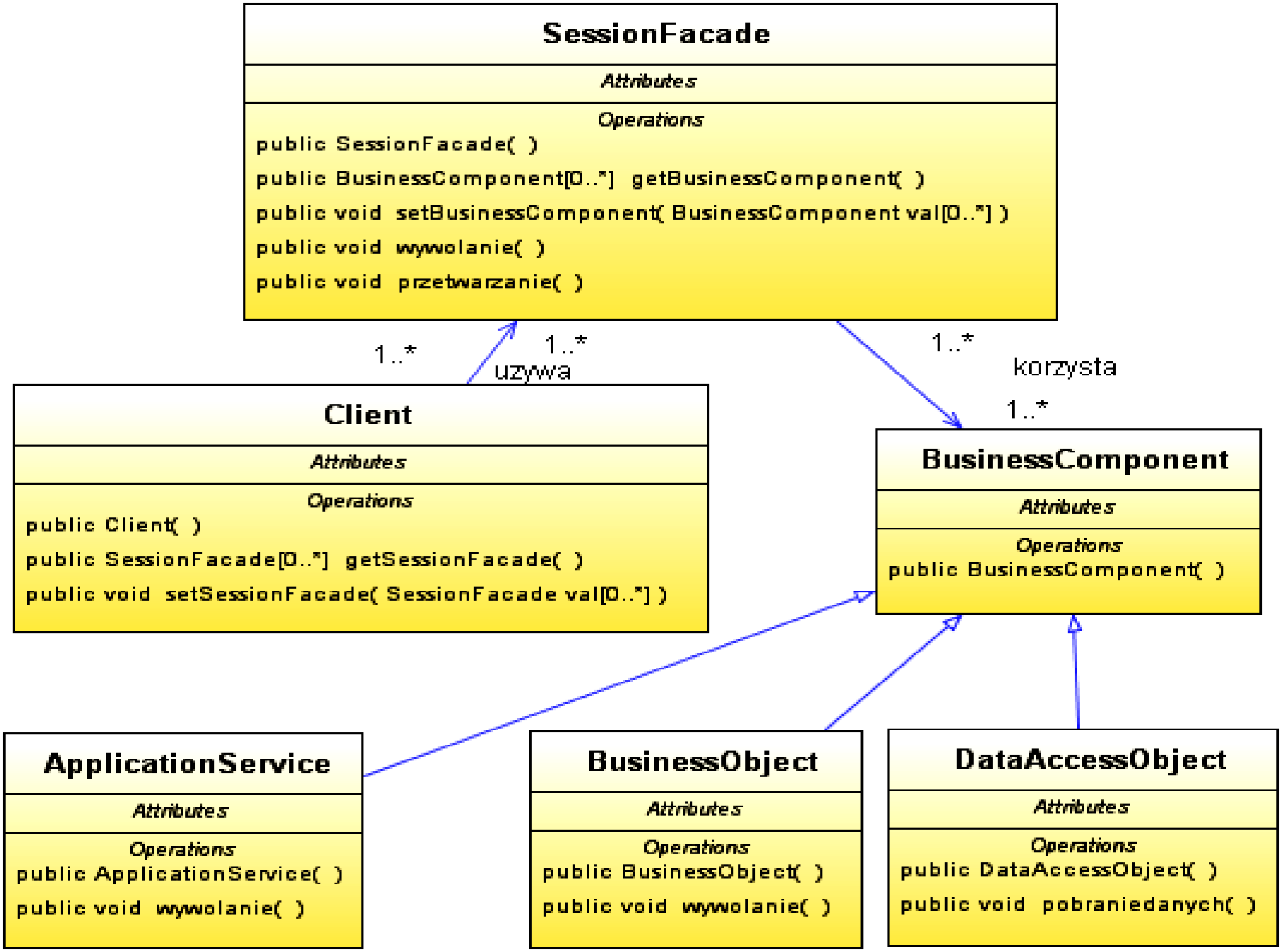
Wzorzec Session Facade: udostępnia klientom ogólne usługi biznesowe realizujące powiązane przypadki użycia, na przykład służące do obsługi rachunku bankowego **- typ „control”**

• **Client** – klient komponentu **Session Facade**, który chce uzyskać dostęp do usług biznesowych. Jest zazwyczaj wzorzec **Business Delegate.**

• **BusinessComponent** – uczestniczy w realizacji żądań klienta. Może nim być **Business Object (problem 5)** jako model obiektowy danych i zachowanie biznesowe lub jako **ApplicationService**

• **ApplicationService** – Obiekt ten korzysta z obiektów biznesowych i implementuje logikę biznesową. Komponent **Session Facade** może korzystać z wielu takich obiektów

• **DataAccessObject** – pośredniczy w dostępie do bazy danych w prostych aplikacjach, w których nie ma warstwy obiektów biznesowych tworzących model obiektowy danych


SessionFacade
Attributes
Operations
public SessionFacade( )
public BusinessComponent[0..*] getBusinessComponent( )
public void setBusinessComponent(BusinessComponent val[0..*])
public void wywolanie( )
public void przetwarzanie( )
1..*
1..*
uzywa
1..*
korzysta
1..*
Client
Attributes
Operations
public Client( )
public SessionFacade[0..*] getSessionFacade( )
public void setSessionFacade(SessionFacade val[0..*])
BusinessComponent
Attributes
Operations
public BusinessComponent( )
ApplicationService
Attributes
Operations
public ApplicationService( )
public void wywolanie( )
BusinessObject
Attributes
Operations
public BusinessObject( )
public void wywolanie( )
DataAccessObject
Attributes
Operations
public DataAccessObject( )
public void pobraniedanych( )

Session Facade

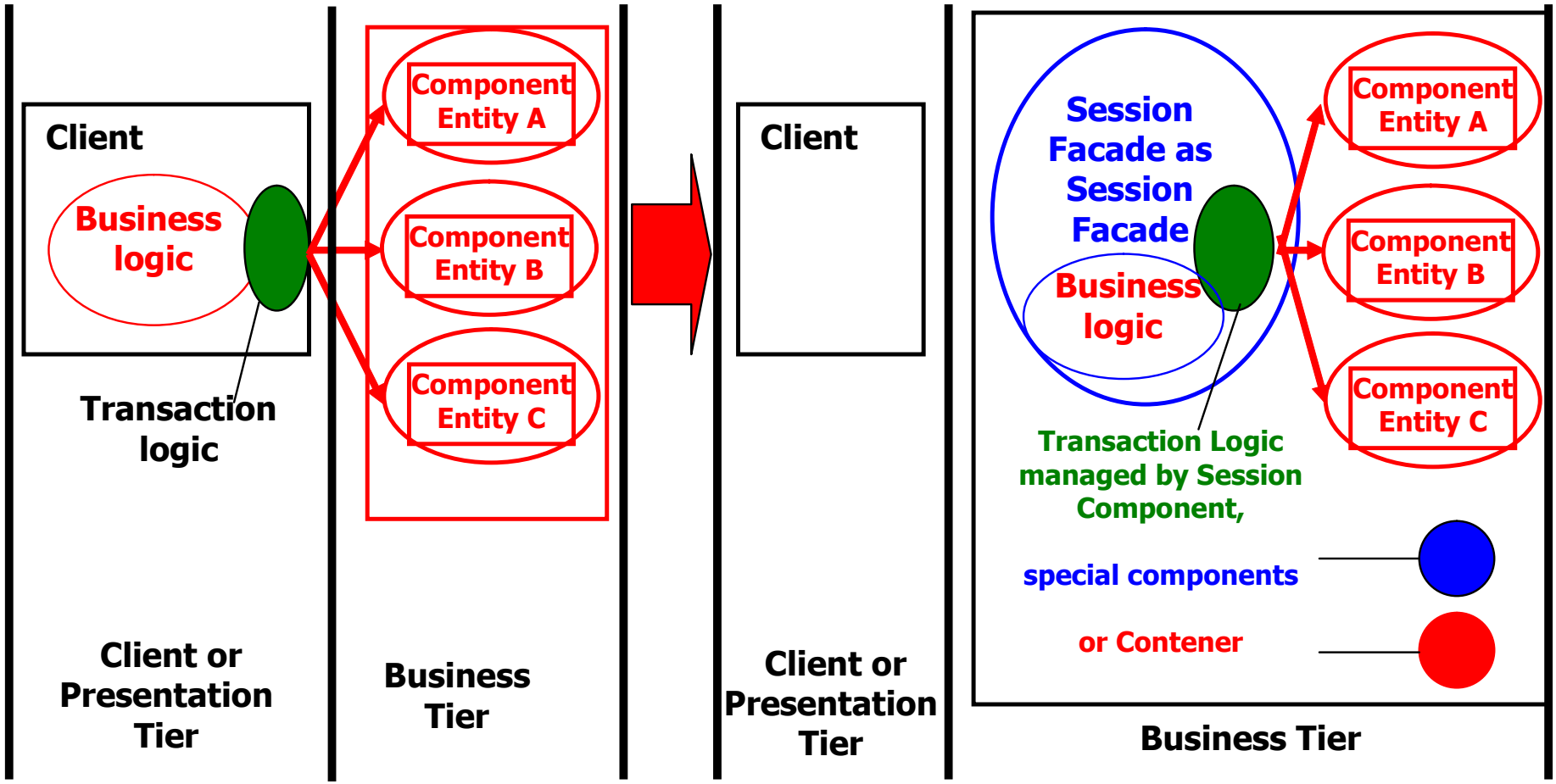
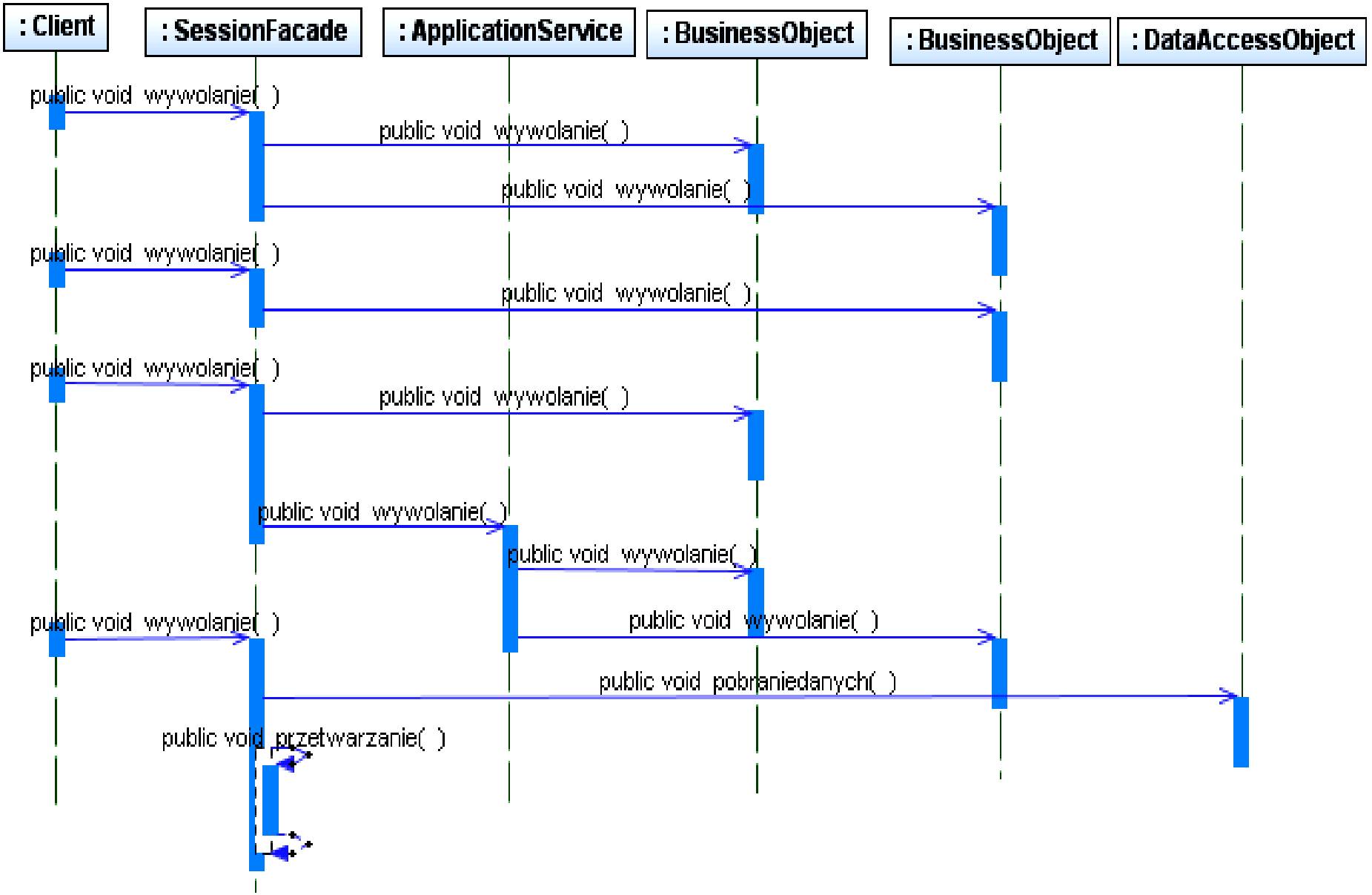


Diagram sekwencji wzorca *Session Facade* (1) –udostępnianie usług biznesowych klientom ukrywając dostęp do komponentów biznesowych wykonujących te operacje

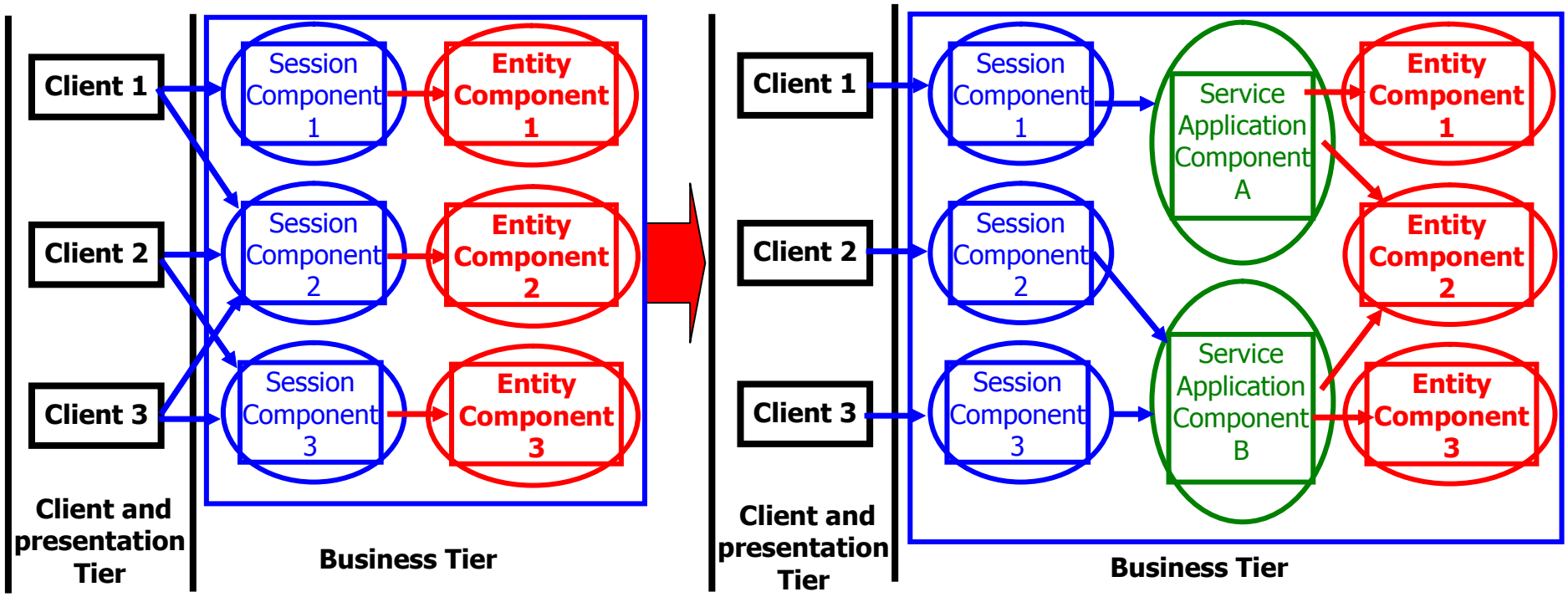


Problem 4 – centralizacja logiki biznesowej kilku komponentów i usług biznesowych

Uwagi:

1. Fasady usług, na przykład **Session Facade** lub fasady zwykłych obiektów zawierają niewiele logiki biznesowej i udostępniają prosty, nierozdrobniony interfejs.
2. Aplikacje implementują przypadki użycia koordynujące współpracę kilku obiektów biznesowych i usług. Nie należy jednak implementować logiki zarządzającej relacjami pomiędzy obiektami biznesowymi w samych obiektach, gdyż zwiększa to zależności pomiędzy nimi i zmniejsza elastyczność. Logiki nie należy umieszczać w fasadzie, gdyż mogłaby ona zostać powielona w wielu komponentach fasady.
3. Aplikacje, które nie stosują komponentów EJB, implementują komponenty warstwy biznesowej jako zwykłe obiekty. Ta logika biznesowa nie powinna być umieszczana ani w fasadzie, ani w kodzie klienta.

Application Service



Wymagania:

1. Należy ograniczyć ilość logiki biznesowej w fasadach usług.
2. Logika biznesowa operuje na kilku obiektach biznesowych lub usługach.
3. Należy utworzyć nierozdrobniony interfejs usług nad istniejącymi komponentami i usługami biznesowymi.
4. Należy umieścić logikę związaną z konkretnymi przypadkami użycia obiektami biznesowymi (głównie dotyczącą wzajemnych zależności pomiędzy obiektami biznesowymi) poza obiektami **Business Object (problem 5)**

Wzorzec **Application Service:** zapewnia jednolitą warstwę usług i stanowi zaplecze fasad - **typ „control”**

- **Client –** fasad typu **Session Facade**, obiekty zwykłych klas Javy, inny obiekt typu **Application Service**
- **Application Service –** pełni główną rolę, hermetyzuje logikę biznesową operującą na kilku obiektach biznesowych lub opartą na konkretnym przypadku użycia. Może on wywoływać metodę biznesową obiektu typu **BusinessObject** lub innego obiektu typu **ApplicationService**
- **BusinessObject (problem 5)–** kilka obiektów tego typu do realizaji żądania **ApplicationService**
- **Service –** komponent udostepniający dowolny rodzaj usługi
- **DataAccessObject** – obiekty dostępu do danych bez pośrednictwa obiektów typu **BusinessObject**

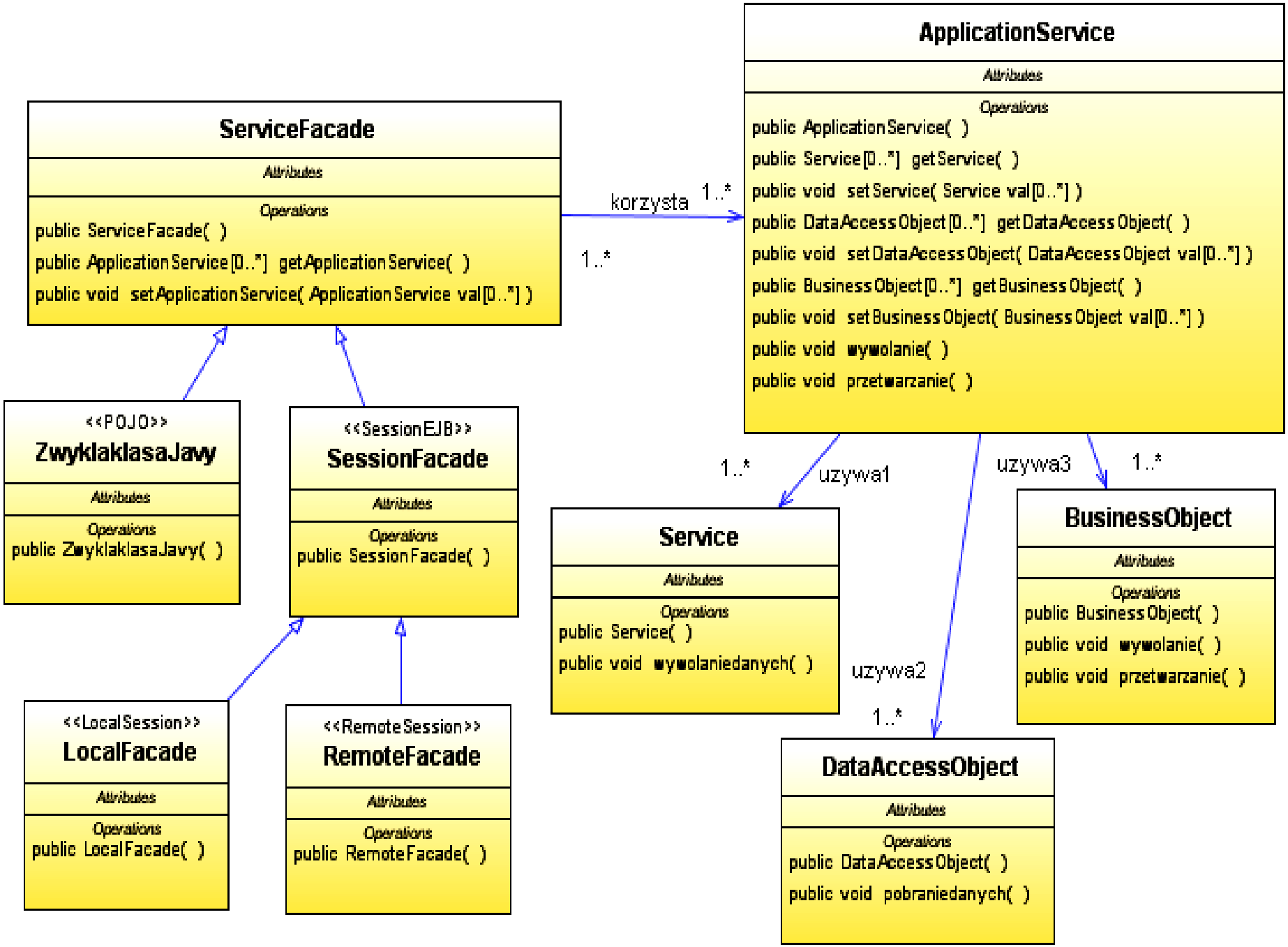
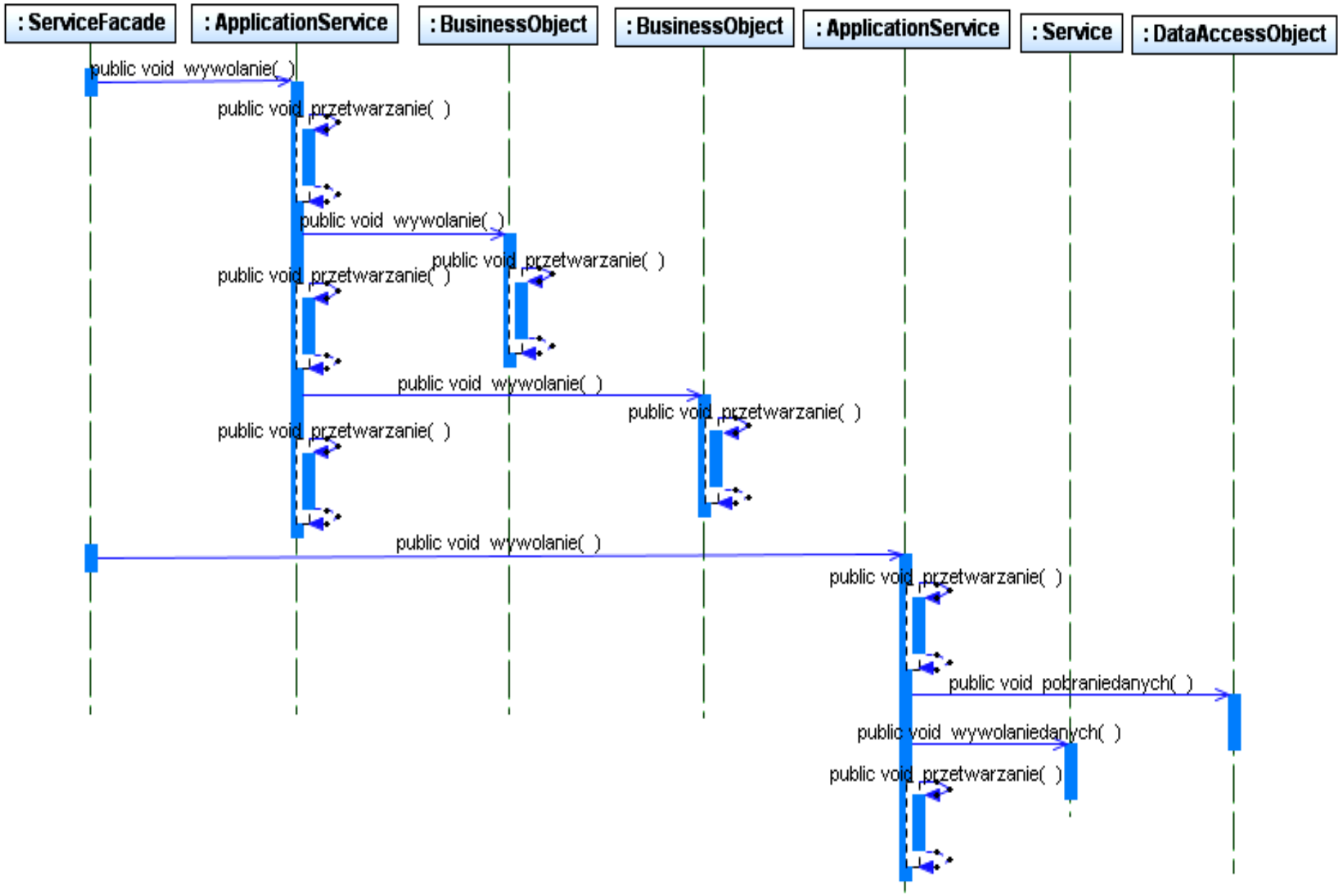

ApplicationService
Attributes
Operations
public ApplicationService( )
public Service[0..*] getService( )
public void setService(Service val[0..*])
public DataAccessObject[0..*] getDataAccessObject( )
public void setDataAccessObject(DataAccessObject val[0..*])
public BusinessObject[0..*] getBusinessObject( )
public void setBusinessObject(BusinessObject val[0..*])
public void wywolanie( )
public void przetwarzanie( )
ServiceFacade
Attributes
Operations
public ServiceFacade( )
public ApplicationService[0..*] getApplicationService( )
public void setApplicationService(ApplicationService val[0..*])
korzysta
1..*
1..*
<<POJO>>
ZwyklaklasaJavy
Attributes
Operations
public ZwyklaklasaJavy( )
<<SessionEJB>>
SessionFacade
Attributes
Operations
public SessionFacade( )
1..*
uzywa1
Service
Attributes
Operations
public Service( )
public void wywolaniedanych( )
uzywa3
1..*
BusinessObject
Attributes
Operations
public BusinessObject( )
public void wywolanie( )
public void przetwarzanie( )
uzywa2
1..*
<<LocalSession>>
LocalFacade
Attributes
Operations
public LocalFacade( )
<<RemoteSession>>
RemoteFacade
Attributes
Operations
public RemoteFacade( )
DataAccessObject
Attributes
Operations
public DataAccessObject( )
public void pobraniedanych( )

Diagram sekwencji wzorca Application Service (1) –zcentralizowanie logiki biznesowej operujących na usługach i obiektach biznesowych



Problem 5 – Istnieje koncepcyjny model domeny z relacjami i logiką biznesową. Model obiektowy jest implementacją modelu koncepcyjnego.

Uwagi:

1. Gdy operacje biznesowe zawierają niewiele logiki biznesowej lub nie zawierają jej wcale, aplikacja zazwyczaj umożliwia bezpośredni dostęp klientom do danych biznesowych. Na przykład strona JSP może bezpośrednio odwoływać się do obiektów dostępu do danych (wzorzec DAO). Oznacza to brak modelu obiektowego w warstwie biznesowej i oznacza zastosowanie implementacji proceduralnej – model danych odpowiada jest blisko związany z koncepcyjnym modelem domeny aplikacji.
2. Jeśli jednak model koncepcyjny aplikacji definiuje szeroki zakres biznesowych relacji i zachowań, wykorzystanie podejścia proceduralnego:

   2.1. zmniejsza możliwość wielokrotnego wykorzystania kodu i powielania logiki biznesowej w wielu miejscach

   2.2. implementacje procedur realizacji usług są długie i złożone

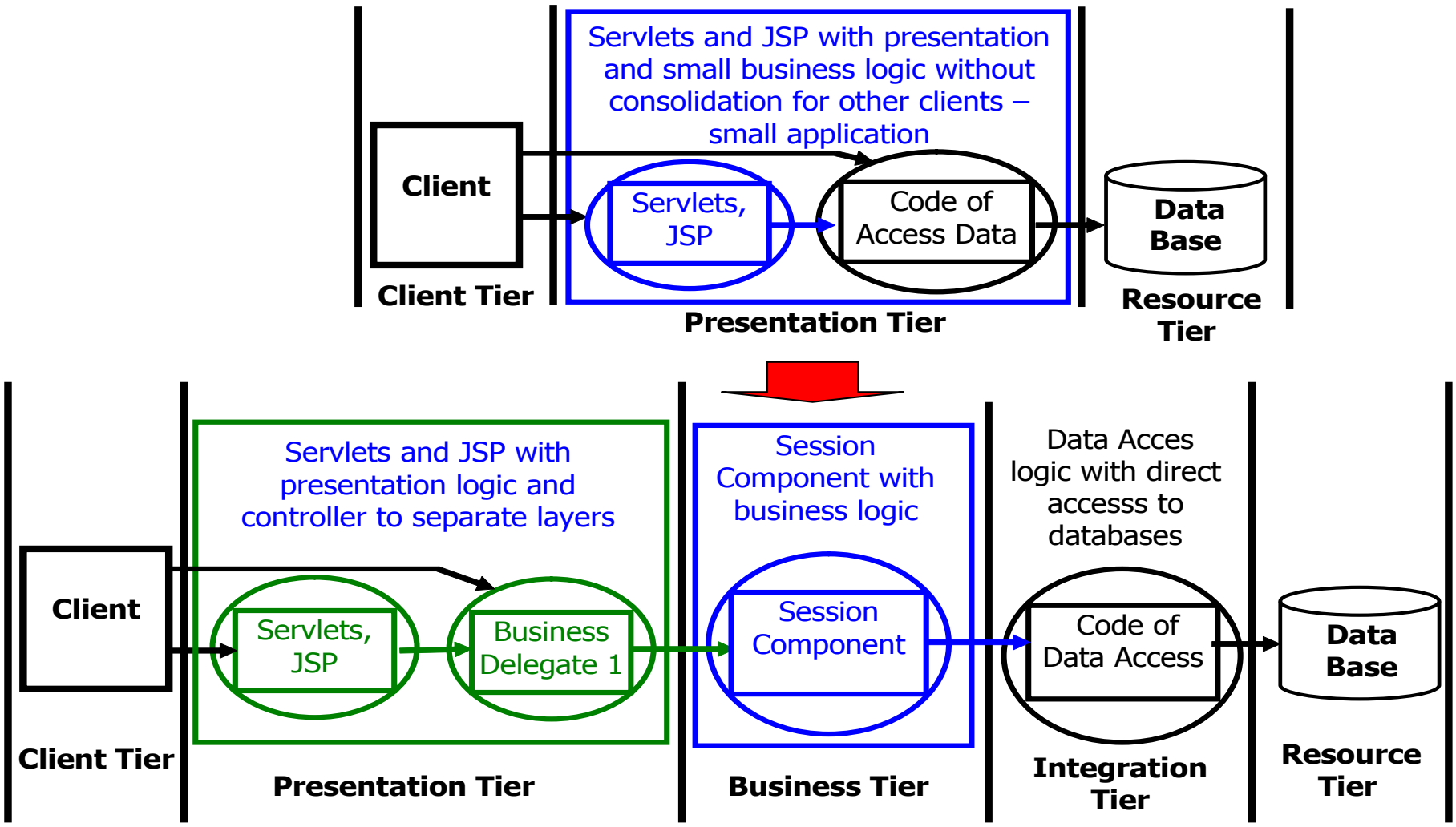
   2.3. z powodu duplikacji kodu i rozmieszczenia logiki biznesowej w różnych modułach pielęgnacja kodu jest znacznie utrudniona

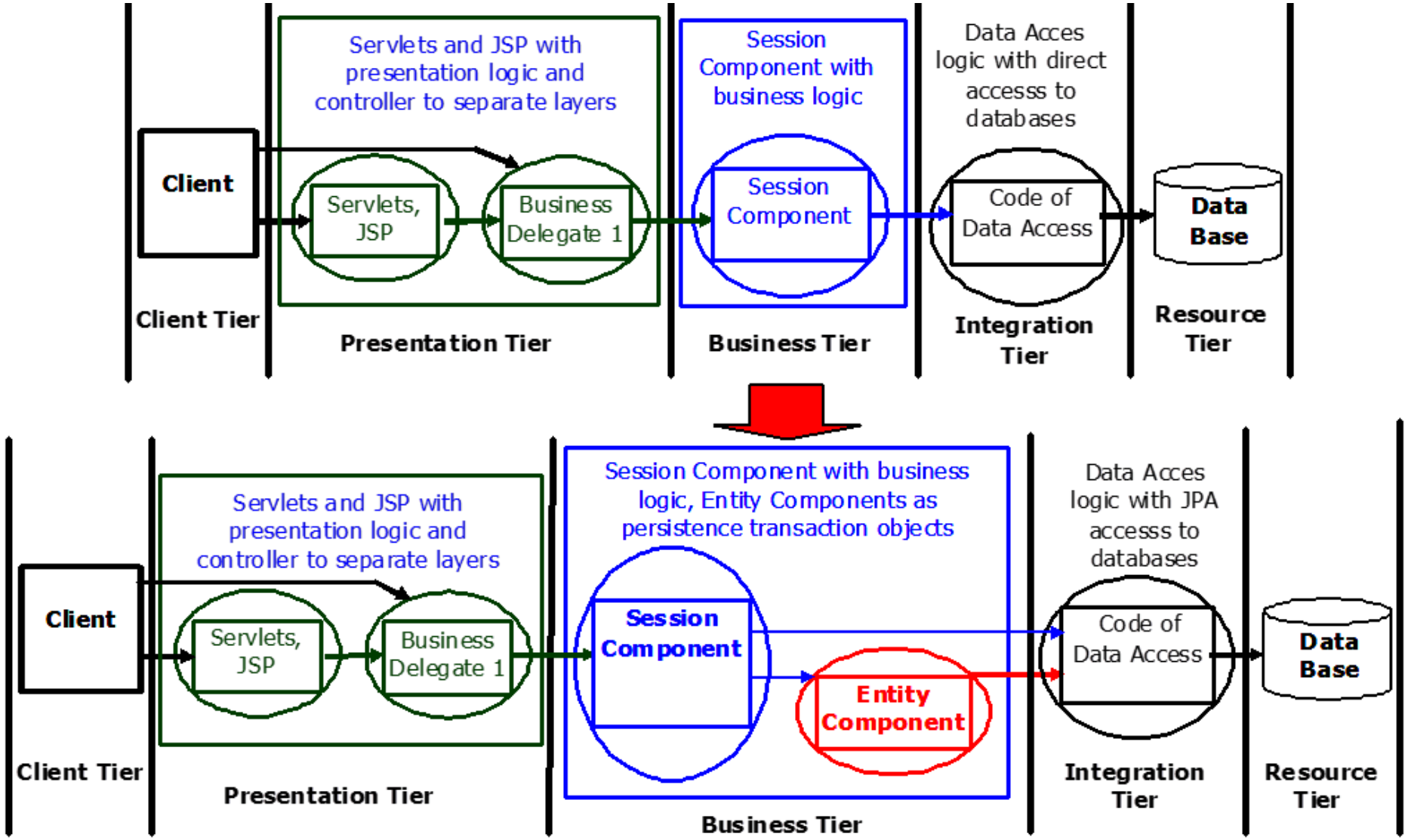
Model biznesowy, model biznesowych przypadków użycia, model obiektów biznesowych, model domenowy, model obiektowy, model danych

(The Unified Software Development Process [Jacobson])

- **Model biznesowy** obejmuje dwa modele: *model biznesowych przypadków użycia* opisujący aktorów i procesy biznesowe oraz *model obiektów biznesowych* opisujący byty wykorzystywane w poszczególnych przypadkach użycia
- **Model domenowy** jest abstrakcyjnym modelem, który opisuje najważniejsze typy obiektów w kontekście systemu. Obiekty domenowe reprezentują zdarzenia oraz „rzeczy" istniejące w środowisku, w którym działa system. **Model domenowy jest traktowany jako model biznesowy.**
- **Model obiektowy** jest implementacją modelu abstrakcyjnego (domenowego)
- **Model danych** jest służy do opisu modelu implementacji danych

Business Object




Client
Servlets and JSP with presentation logic and controller to separate layers
Servlets, JSP
Business Delegate 1
Session Component with business logic
Session Component
Data Acces logic with direct accesss to databases
Code of Data Access
Data Base
Client Tier
Presentation Tier
Business Tier
Integration Tier
Resource Tier
Client
Servlets and JSP with presentation logic and controller to separate layers
Servlets, JSP
Business Delegate 1
Session Component with business logic, Entity Components as persistence transaction objects
Session Component
Entity Component
Data Acces logic with JPA accesss to databases
Code of Data Access
Data Base
Client Tier
Presentation Tier
Business Tier
Integration Tier
Resource Tier

Wymagania:

1. Istnieje model koncepcyjny zawierający strukturalne, wzajemnie powiązane złożone obiekty.
2. Istnieje model koncepcyjny ze szczegółowo zdefiniowaną logiką biznesową ograniczeniami i regułami biznesowymi **(1).**
3. Należy oddzielić stan biznesowy i związane z nim zachowanie od reszty aplikacji, poprawiając spójność i łatwość wielokrotnego wykorzystania komponentów aplikacji.
4. Należy zcentralizować logikę biznesową i stan biznesowy w jednym miejscu aplikacji.
5. Należy zwiększyć możliwość wielokrotnego wykorzystania logiki biznesowej i uniknąć powielenia kodu.

Wzorzec **Business Object**: wydzielenie danych i logiki biznesowej przy użyciu modelu obiektowego – **typ „Entity”**

Istnieją dwa sposoby implementacji, związane z zagadnieniami bezpieczeństwa, zarządzaniem transakcjami, pulami zasobów, buforowaniem i współbieżnością:

1. **Użycie obiektów zwykłych klas Javy (POJO) oraz dowolnego mechanizmu trwałości** spełniającego konkretne wymagania np.: obiekty *DAO*, własna implementacja mechanizmu trwałości wykorzystująca wzorzec *Domain Store* lub implementacja zgodna z JDO.
2. Zastosowanie obiektów **Entity** zgodnie z zaleceniami wzorca **Composite Entity**. W tej strategii należy zdecydować, czy korzystać z trwałości BMP (realizowanej przez komponent) czy CMP (realizowanej przez kontener) .

Proste aplikacje działające na tej samym komputerze mogą te obiekty biznesowe udostępnić klientom, w przypadku wywołań zdalnych należy stosować wzorce: **Session Facade, Application Service**, do przesyłania danych **Transfer Object**.

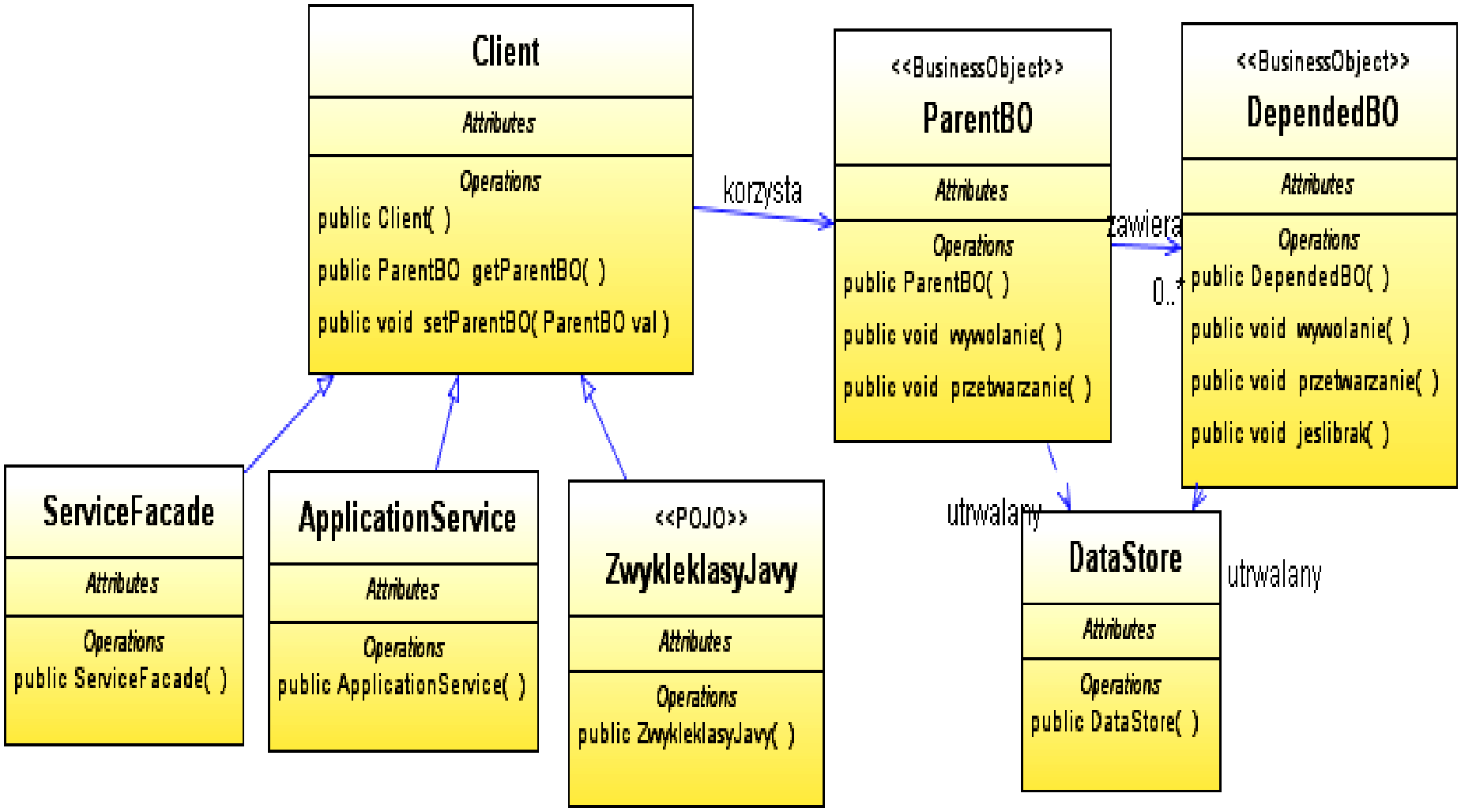
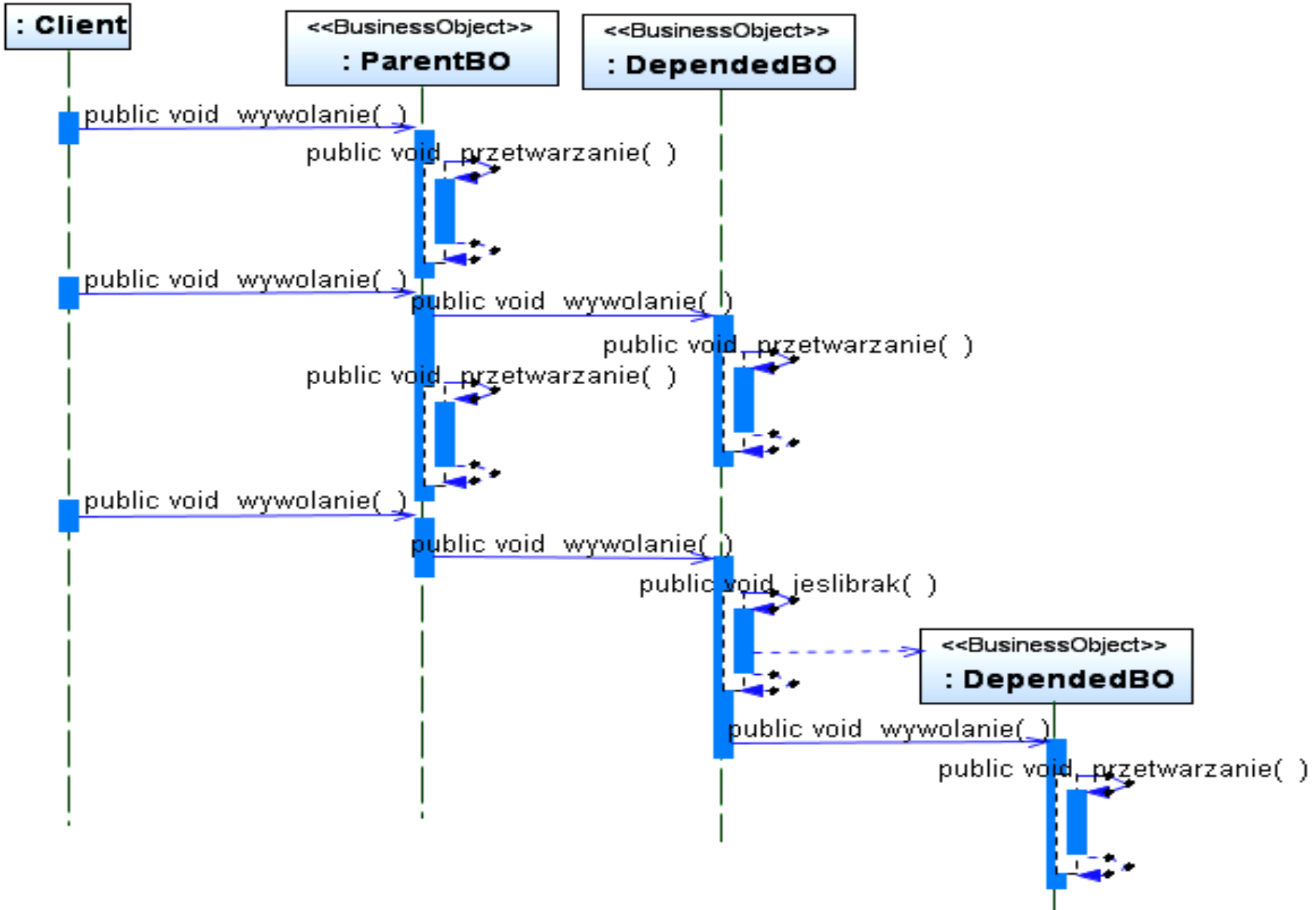

Client
Attributes
Operations
public Client( )
public ParentBO getParentBO( )
public void setParentBO(ParentBO val)
korzysta
<<BusinessObject>>
ParentBO
Attributes
Operations
public ParentBO( )
public void wywolanie( )
public void przetwarzanie( )
zawiera
0..*
<<BusinessObject>>
DependedBO
Attributes
Operations
public DependedBO( )
public void wywolanie( )
public void przetwarzanie( )
public void jeslibrak( )
ServiceFacade
Attributes
Operations
public ServiceFacade( )
ApplicationService
Attributes
Operations
public ApplicationService( )
<<POJO>>
ZwykleklasyJavy
Attributes
Operations
public ZwykleklasyJavy( )
utrwalany
DataStore
Attributes
Operations
public DataStore( )
utrwalany

Diagram sekwencji wzorca BusinessObject (1) – działanie logiki biznesowej na danych pobranych z bazy danych; w przypadku zmiany danych w wyniku przetwarzani zapisuje się je w bazie danych



Problem 6 – Użycie komponentów Entity w celu implementacji koncepcyjnego modelu domeny (JavaEE 2).

Uwagi:

Należy dokonać wyboru pomiędzy lokalnymi i zdalnymi komponentami **Entity.** Lokalne komponenty są bardziej wydajne niż zdalne, jednak mniej wydajne od zwykłych obiektów biznesowych, implementujących wzorzec **Business Object.**

Wymagania:

1. Należy uniknąć wad zdalnych komponentów **Entity**, na przykład narzutu sieci i zdalnych relacji między komponentami
2. Należy wykorzystać trwałość zarządzaną przez komponent (BMP), stosując własne lub nietypowe implementacje mechanizmów trwałości.
3. Należy w optymalny sposób zaimplementować relacje rodzic-potomek, wykorzystując obiekty biznesowe zaimplementowane w postaci komponentów **Entity**.
4. Należy wykorzystać i połączyć istniejące obiekty biznesowe zaimplementowane jako zwykłe obiekty z komponentami **Entity**.
5. Należy wykorzystać udostępniane przez kontener EJB mechanizmy zarządzania transakcjami i bezpieczeństwem.
6. Należy ukryć fizyczny projekt bazy danych przed klientami aplikacji.

Wzorzec **Composite Entity**: implementacja połączonych (za pomocą agregacji) trwałych obiektów biznesowych w postaci lokalnych komponentów Entity i zwykłych obiektów Javy – **typ „Entity”**

Istnieją dwa sposoby implementacji, związane z zagadnieniami bezpieczeństwa, zarządzaniem transakcjami, pulami zasobów, buforowaniem i współbieżnością:

1. Użycie obiektów zwykłych klas Javy (POJO) oraz dowolnego mechanizmu trwałości spełniającego konkretne wymagania np.: obiekty *DAO*, własna implementacja mechanizmu trwałości wykorzystująca wzorzec *Domain Store* lub implementacja zgodna z JDO.
2. Zastosowanie obiektów *Entity* zgodnie z zaleceniami wzorca *Composite Entity*. W tej strategii należy zdecydować, czy korzystać z trwałości BMP czy CMP.

Proste aplikacje działające na tej samym komputerze mogą te obiekty biznesowe udostępnić klientom, w przypadku aplikacji złożonych korzystających z wywołań zdalnych należy stosować wzorce: ***Session Facade, Application Service***, do przesyłania danych ***Transfer Object***.

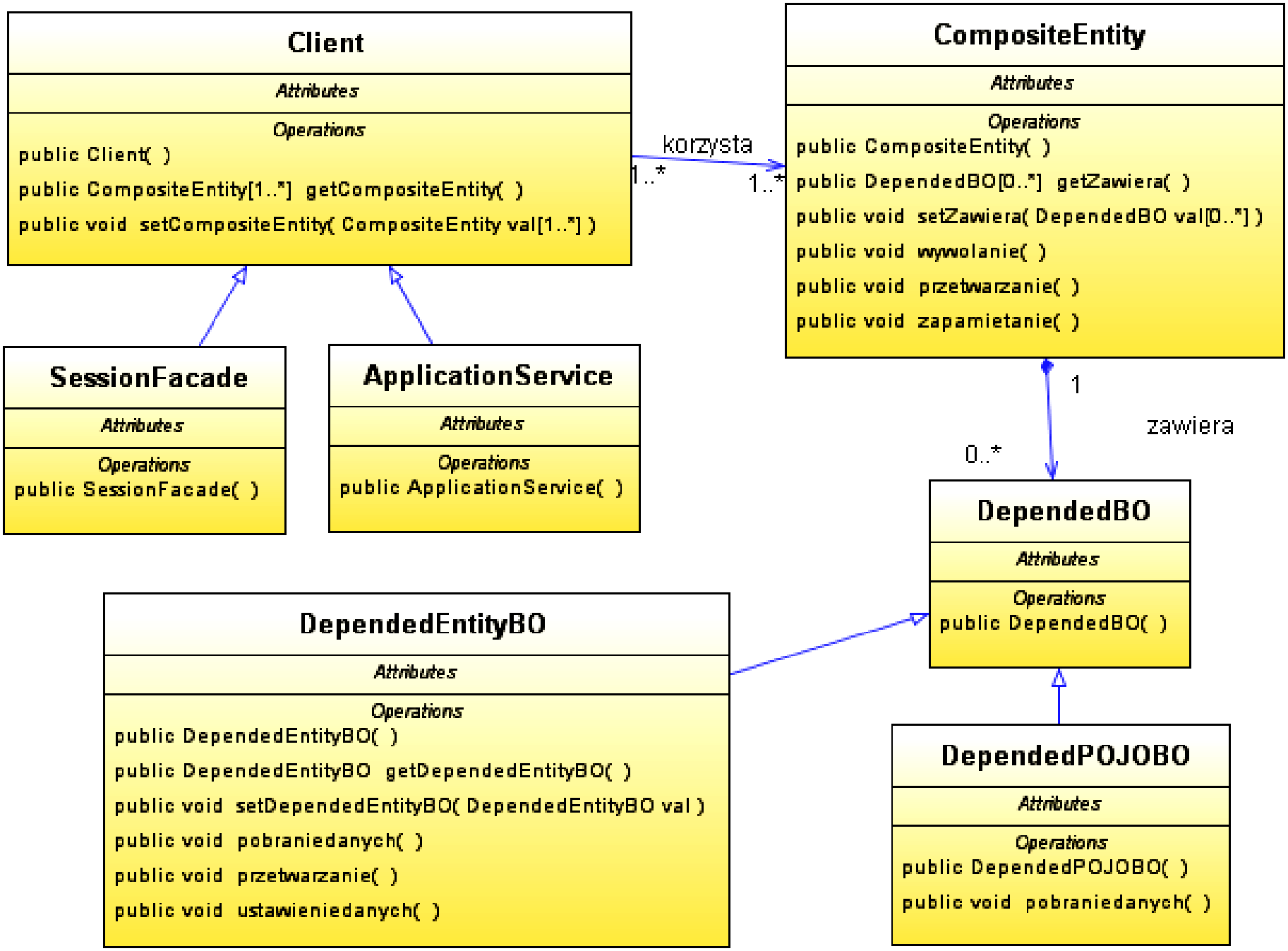
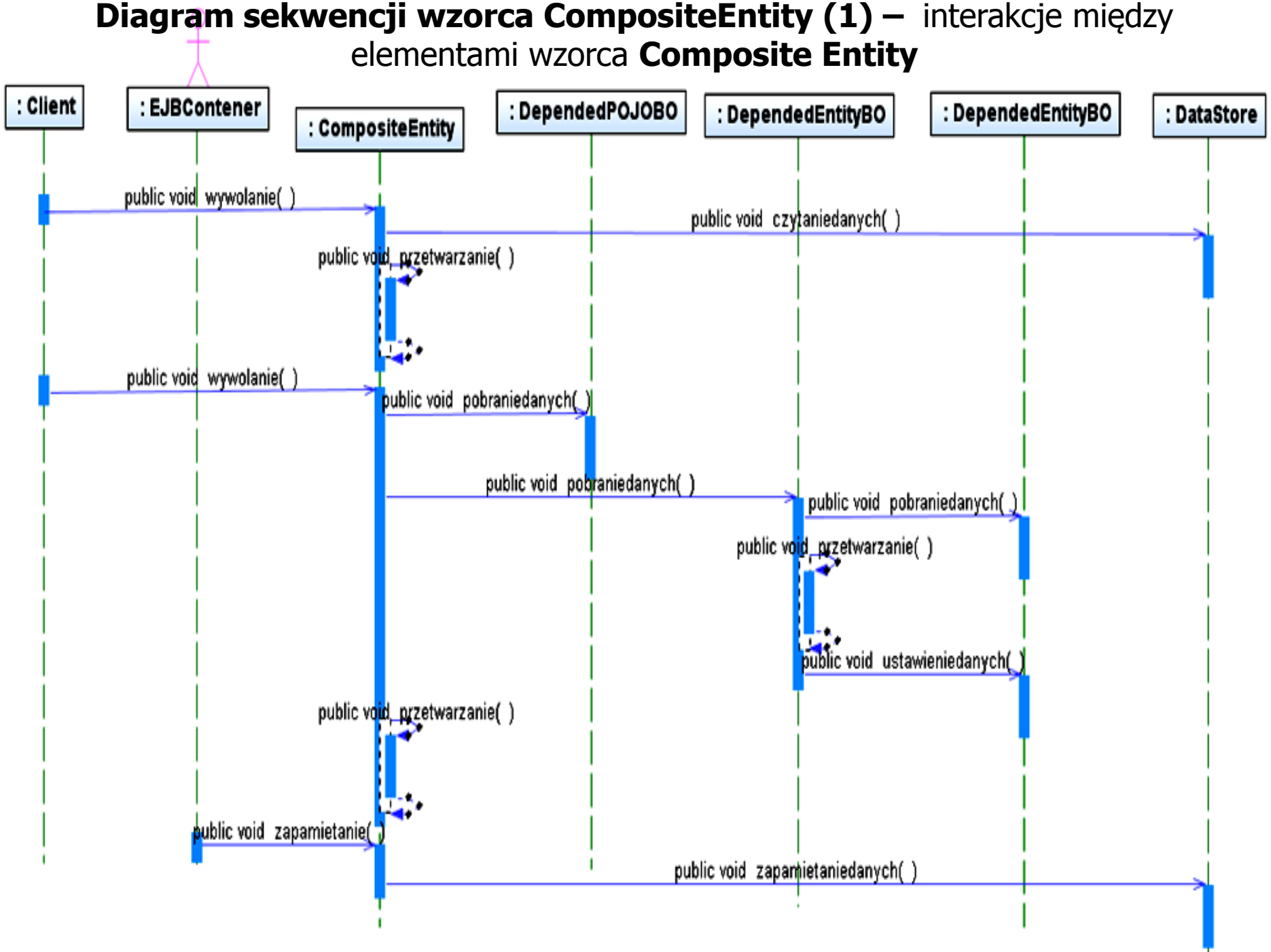

Client
Attributes
Operations
public Client( )
public CompositeEntity[1..*] getCompositeEntity( )
public void setCompositeEntity(CompositeEntity val[1..*])
korzysta
1..*
1..*
CompositeEntity
Attributes
Operations
public CompositeEntity( )
public DependedBO[0..*] getZawiera( )
public void setZawiera(DependedBO val[0..*])
public void wywolanie( )
public void przetwarzanie( )
public void zapamietanie( )
1
zawiera
0..*
SessionFacade
Attributes
Operations
public SessionFacade( )
ApplicationService
Attributes
Operations
public ApplicationService( )
DependedBO
Attributes
Operations
public DependedBO( )
DependedEntityBO
Attributes
Operations
public DependedEntityBO( )
public DependedEntityBO getDependedEntityBO( )
public void setDependedEntityBO(DependedEntityBO val)
public void pobraniedanych( )
public void przetwarzanie( )
public void ustawieniedanych( )
DependedPOJOBO
Attributes
Operations
public DependedPOJOBO( )
public void pobraniedanych( )

Diagram sekwencji wzorca CompositeEntity (1) – interakcje między elementami wzorca **Composite Entity**



Problem 7 – Przesyłanie wielu danych między warstwami (ograniczenie ruchu w sieci przez zmniejszenie liczby zdalnych wywołań, czyli poprawa wydajności).

Uwagi:

1. Przesyłanie danych między warstwami (z obiektów biznesowych z warstwy biznesowej lub obiektów DAO z warstwy integracji) nie powinno generować dużego ruchu w sieci, dlatego warto przesyłać wiele danych w jednym obiekcie złożonym typu **Transfer Object.**
2. Uniezależnienie implementacji warstwy prezentacji od warstwy biznesowej korzystającej obiektów **Transfer Object** oraz warstwy biznesowej od warstwy integracji.

Uwagi:

1. Użycie obiektu **Transfer Object** nie powinno dotyczyć przechowywania danych w obiektach biznesowych, ze względu na:

   1.1. *cel zastosowania*: do przesyłania danych między warstwami, a nie do przechowywania stanu obiektów biznesowych.

   1.2. *przypadek dodania lub usunięcia pola w obiekcie transferowym*: zmiany w obiektach biznesowych

   1.3. *przypadek dodania lub usunięcia pola w obiekcie biznesowym*: efekt kaskadowy zmian w obiektach transferowych

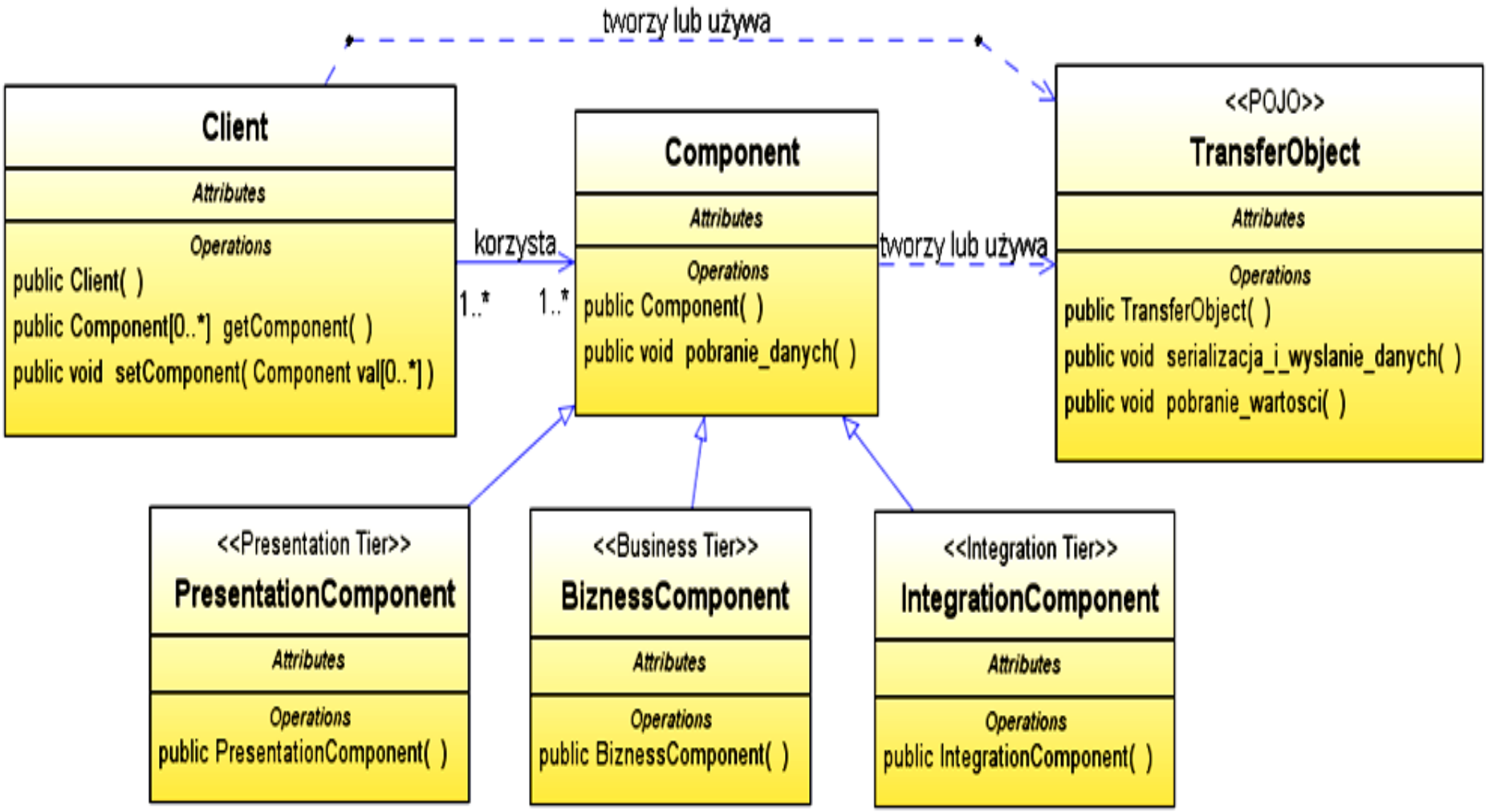
   1.4. *przypadek wysłania innego obiektu transferowego niż znajduje się w obiekcie biznesowym*: rozmnożenie obiektów transferowych

   1.5. *przypadek, gdy obiekt biznesowy musi wysłać obiekt transferowy z polami pochodnymi (zastosowanie dziedziczenia):* rozmnożenie obiektów transferowych

Wymagania:

1. Należy umożliwić klientom dostęp do komponentów z innych warstw i możliwość pobrania oraz modyfikacji ich danych.
2. Należy zmniejszyć liczbę zdalnych wywołań.
3. Należy uniknąć zmniejszenia wydajności spowodowanego przez znaczną ilość zdalnych wywołań.

Wzorzec Transfer Object – typ „Control”



Korzystanie z obiektu typu ***TransferObject*** przez komponenty różnych typów