Diagramy klas i sekwencji
Wykład5

Zofia Kruczkiewicz

Diagramy klas, diagramy sekwencji

1. **Wprowadzenie**
2. **Syntaktyka diagramów klas**

   https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html
3. **Identyfikacja elementów diagramów klas**

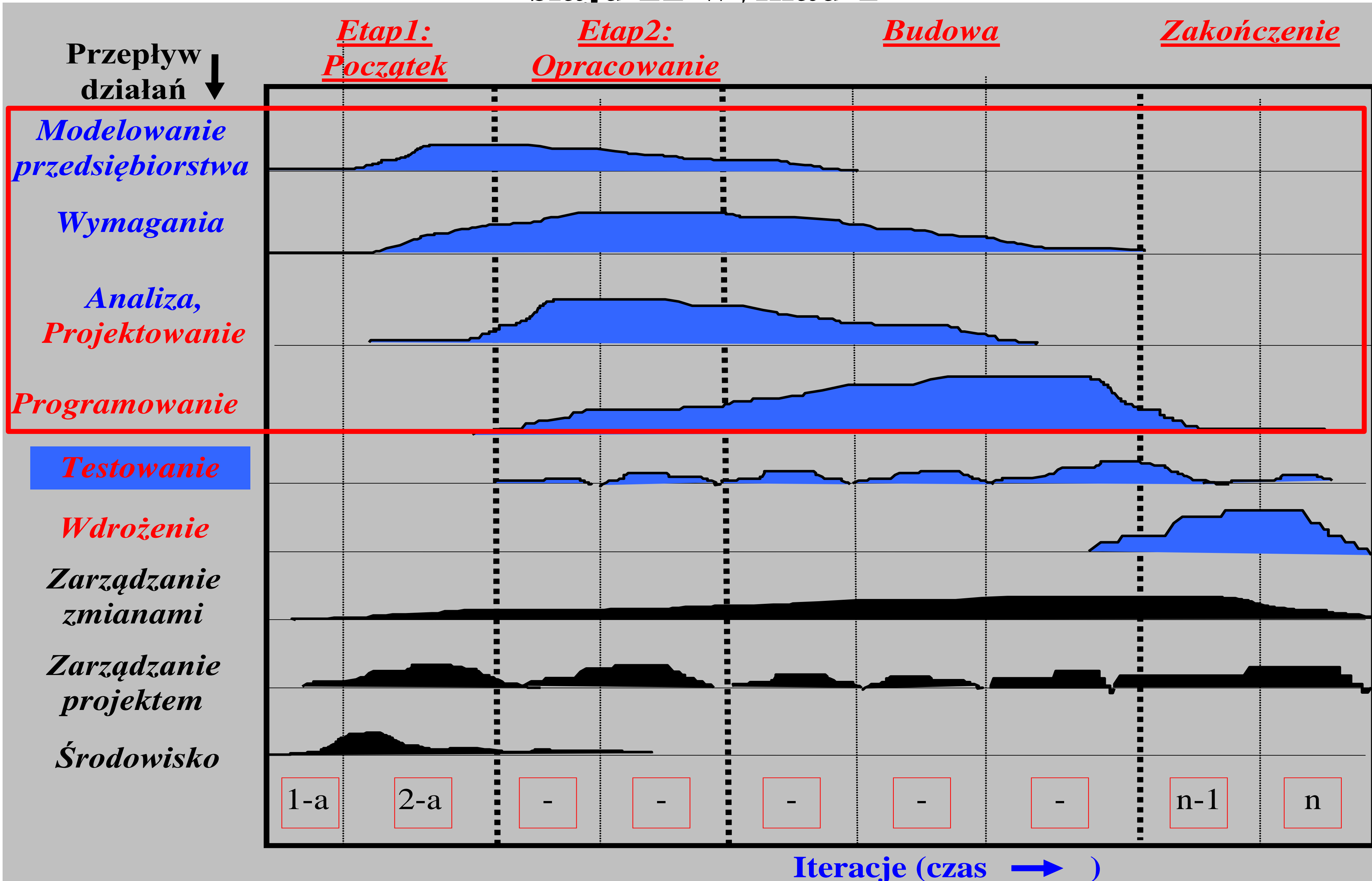
   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]
4. **Diagramy sekwencji UML**

   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html
5. **Przykłady diagramów sekwencji – kontynuacja przykładu 3 z wykładów: 2, 3, 4**

Diagramy klas, diagramy sekwencji

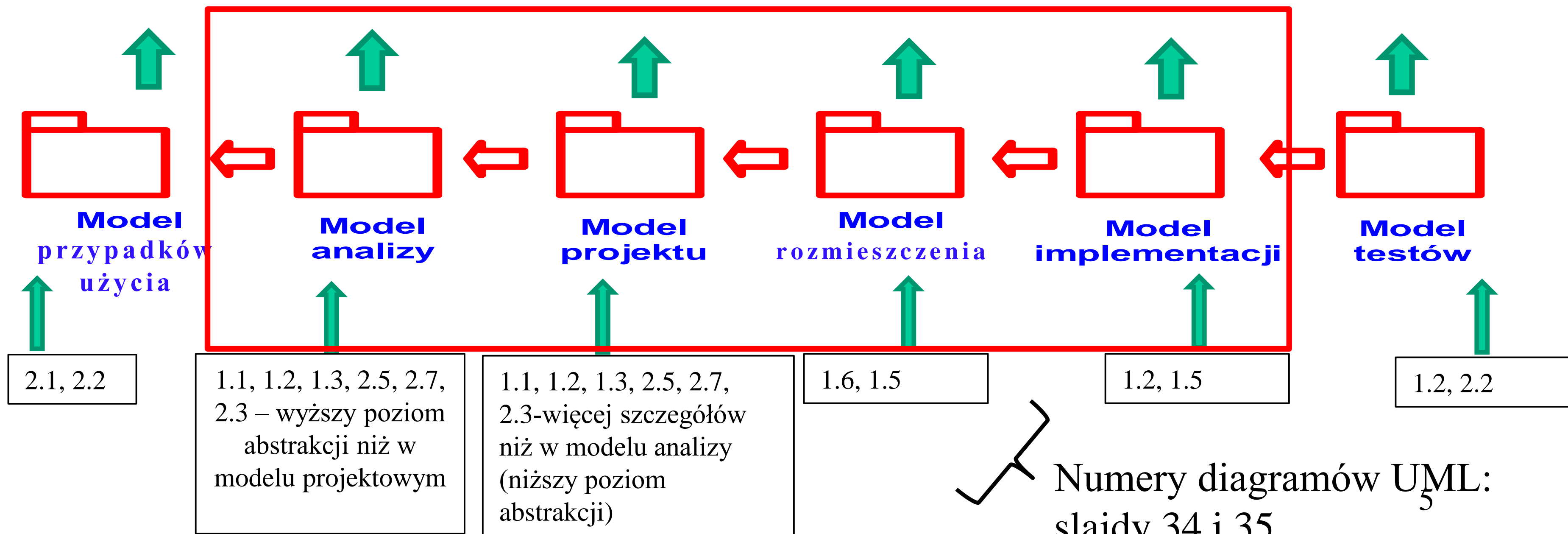
1. **Wprowadzenie**

Proces - zunifikowany iteracyjno- przyrostowy proces tworzenia oprogramowania – kiedy należy wykonać? [3LU] - slajd 22 wykład 1

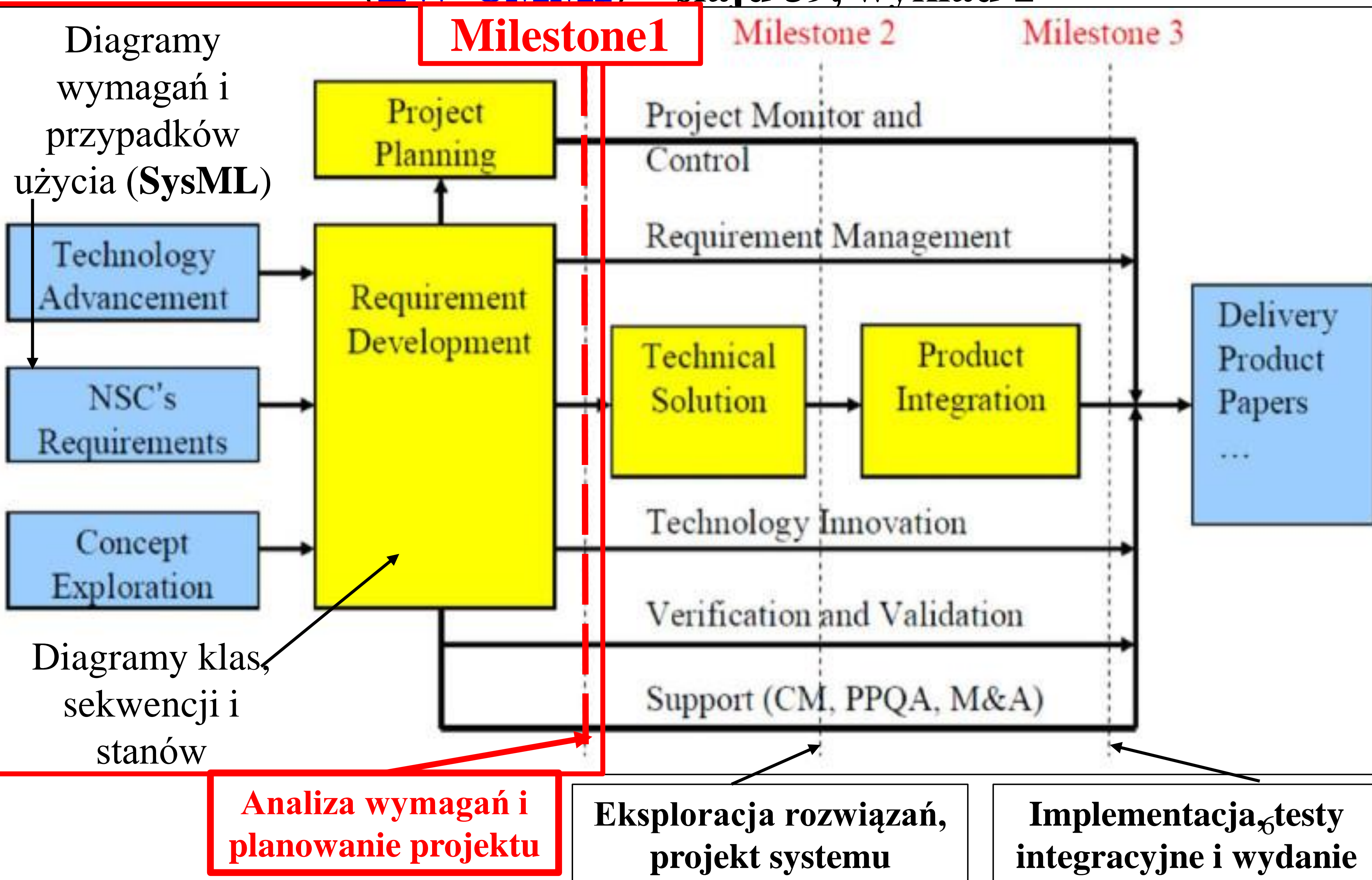


Produkt - diagramy UML – modele, proces - slajd 45, wykład 1

Modelowanie struktury i dynamiki systemu	Implementacja struktury i dynamiki systemu, generowanie kodu	
Perspektywa koncepcji *co należy wykonać?*	Perspektywa specyfikacji *jak należy używać?*	Perspektywa implementacji *jak należy wykonać?*
• model problemu np. przedsiębiorstwa • wymagania • analiza (model konceptualny: diagram przypadków użycia, diagram klas, diagramy sekwencji,) • testy modelu	• projektowanie (model projektowy: architektura sprzętu i oprogramowania; dostęp użytkownika; przechowywanie danych) • testy projektu	• programowanie, wdrażanie (specyfikacja programu : deklaracje, definicje; dodatkowe struktury danych: struktury „pojemnikowe", pliki, bazy danych) • testy oprogramowania • wdrażanie • testy wdrażania



Cykl życia tworzenia oprogramowania w dziedzinie medycyny nuklearnej: Light-Weight Capability Maturity Model Integration (LW-CMMI) – slajd 59, wykład 1



Produkt – oprogramowanie na platformie Java EE

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

Warstwa	Zadania
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

Diagramy klas, diagramy sekwencji

1. Wprowadzenie

2. Syntaktyka diagramów klas

https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html

Dwa rodzaje diagramów UML 2

Diagramy UML modelowania strukturalnego

- Diagramy pakietów
- *Diagramy klas*
- Diagramy obiektów
- Diagramy mieszane
- Diagramy komponentów
- Diagramy wdrożenia

Diagramy UML modelowania zachowania

- *Diagramy przypadków użycia*
- *Diagramy czynności*
- Diagramy stanów
- Diagramy komunikacji
- Diagramy sekwencji
- Diagramy czasu
- Diagramy interakcji

Diagramy klas (Class Diagrams)

- **Diagram klas** reprezentuje statyczny model świata rzeczywistego: jego atrybuty i właściwości, odpowiedzialności oraz powiązania
- **Klasa** reprezentuje model rzeczy konceptualnej i fizycznej i jest powielana w postaci **obiektów**, czyli wystąpień klasy.
- **Atrybuty***:* składowe klasy do przechowywania danych, które posiadają nazwę, typ, zakres wartości oraz określony dostęp.
- **Operacje:** składowe klasy do wykonania operacji na atrybutach, zadeklarowane jako funkcje publiczne, chronione lub prywatne posiadające nazwę oraz zdefiniowany sposób wykonania.

Notatacje

Atrybuty: length, width, center. Atrybut **center** posiada wartość początkową.

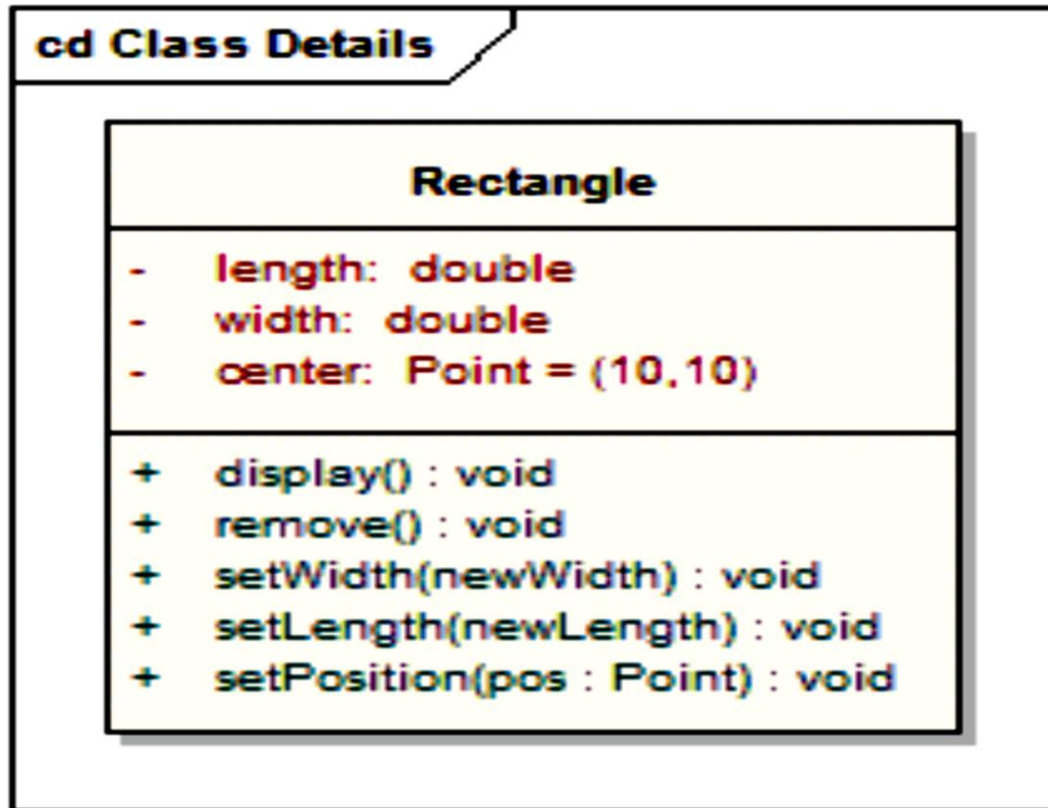
Operacje: setWidth, setLength, setPosition

+ składowa publiczna

- składowa prywatna

składowa typu protected

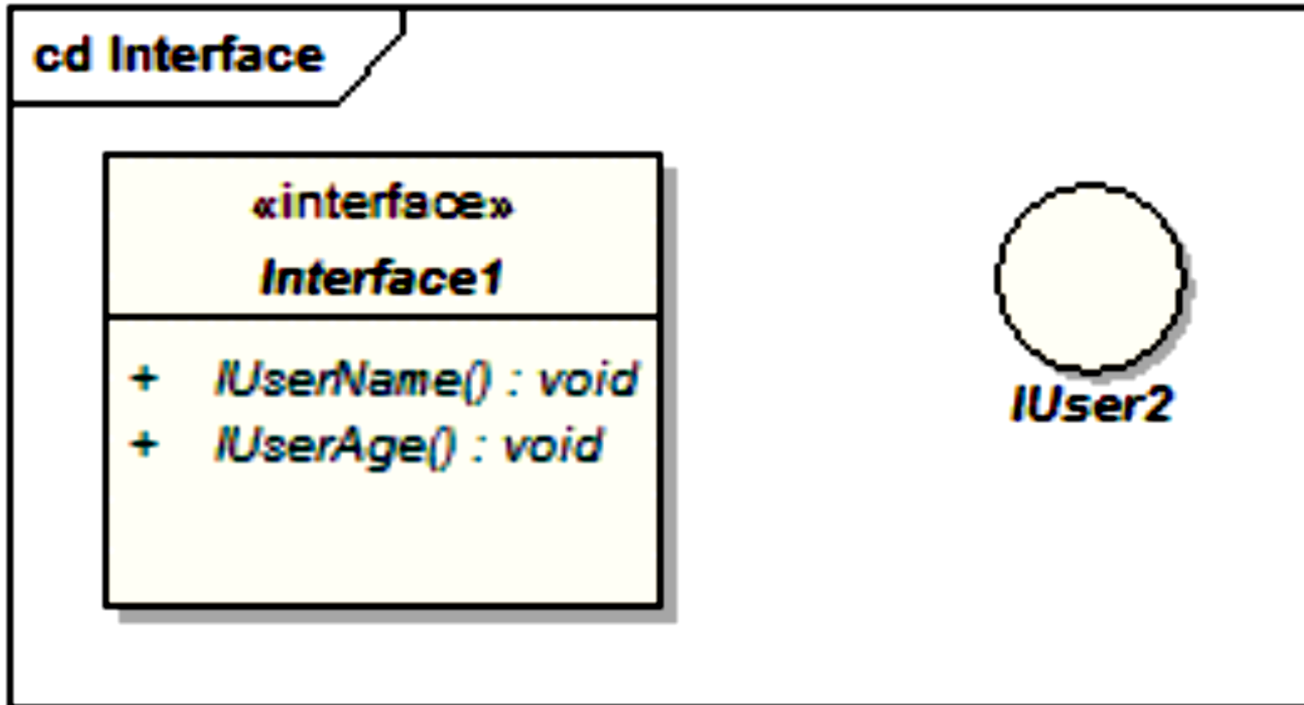
~ składowa publiczna w zasięgu pakietu

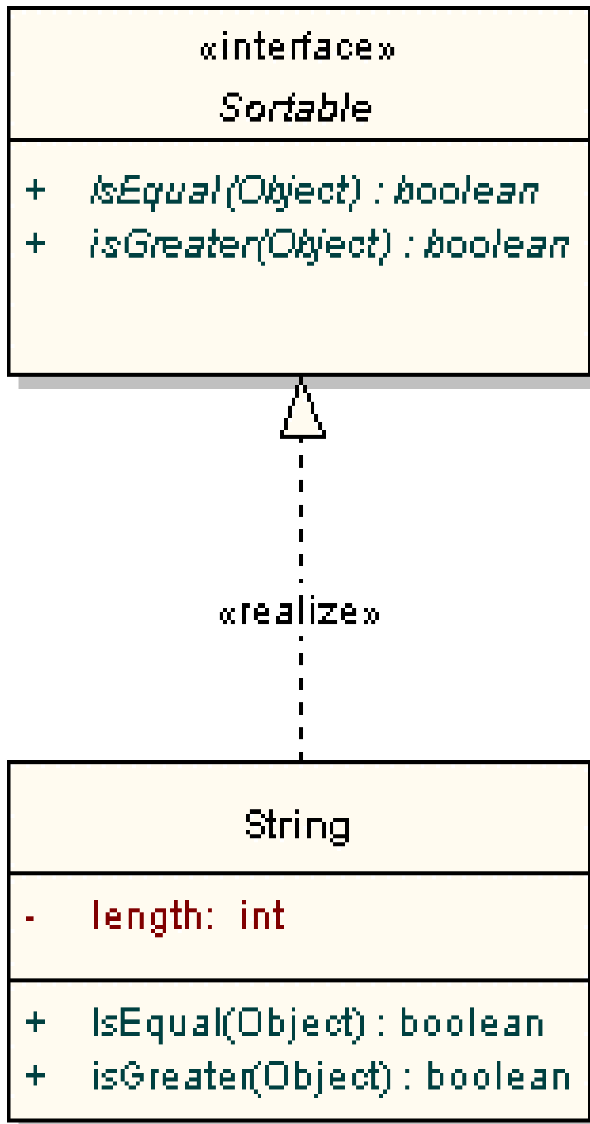


Interfejs<<interface>>

Jest przedstawiany jako:

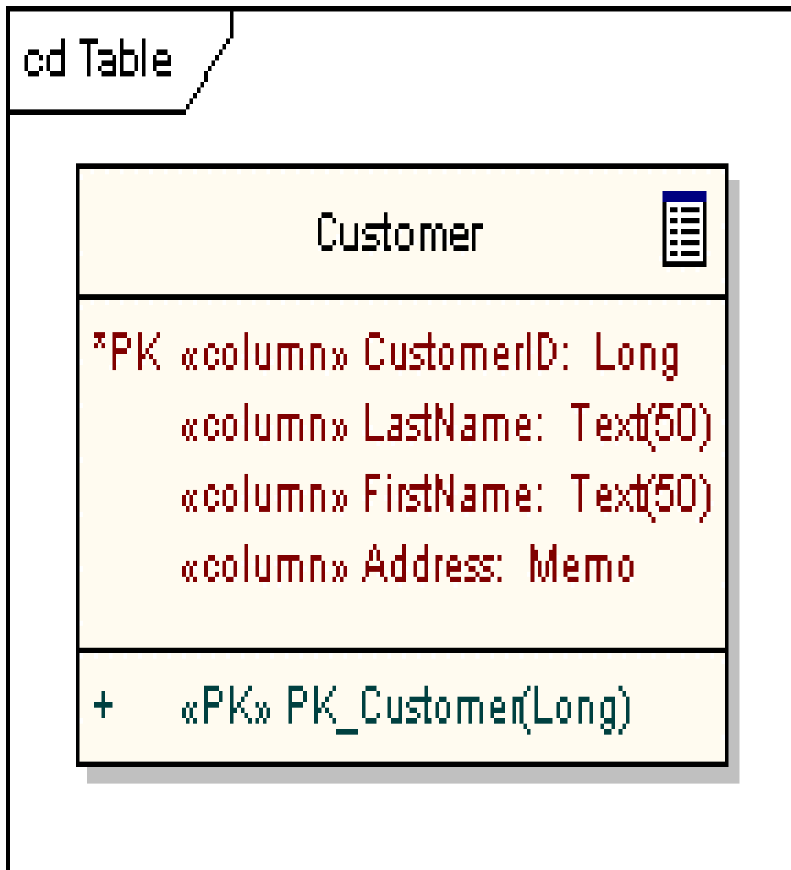
- klasa zawierająca specyfikację właściwości (operacji czyli metod), które musi zdefiniować implementująca go klasa
- **reprezentowany jest jako koło** bez wyspecyfikowanych metod i połączenia z interfejsem przez klasę implementującą nie są oznaczane strzałką





Realizacja (Realization)

- oznaczane są przerywaną strzałką ze stereotypem **<<realize>>**
- strzałka wychodzi z klasy implementującej do klasy implementowanej
- implementacja właściwości klasy typu ***interface***
- klasa implementująca jest rysowana podobnie jak klasa implemetowana



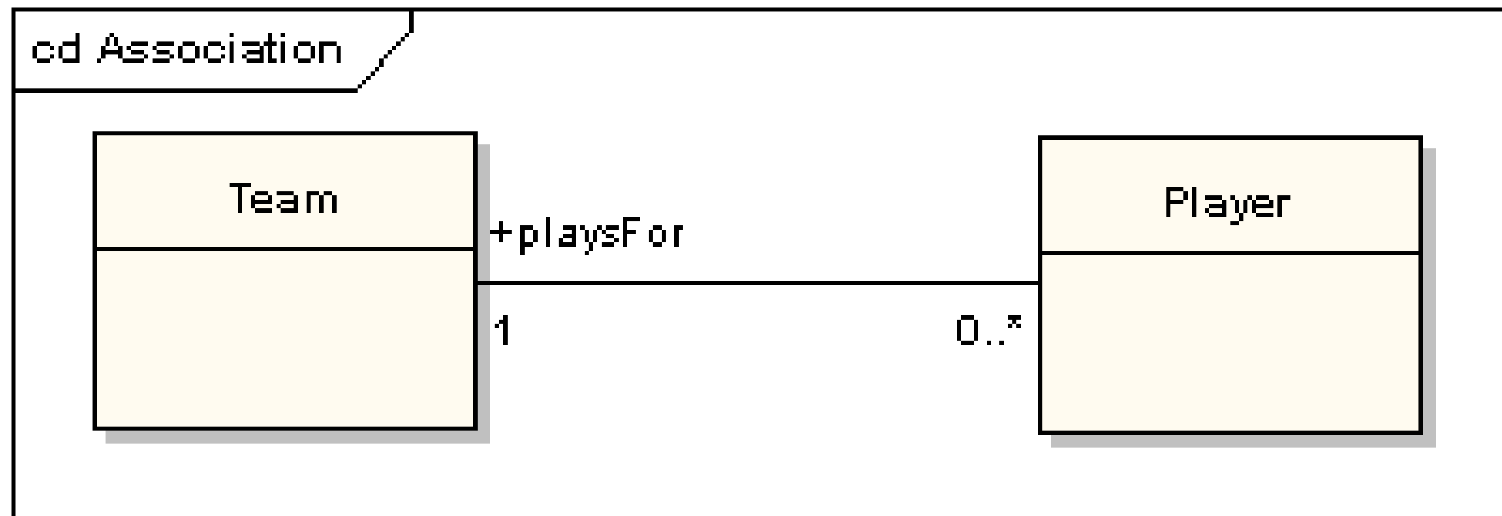
Tabele (table)

- **klasa stereotypowa**
- **atrybuty tabeli o stereotypie <<column>>**
- **posiada klucz główny (<<PK>> – primary key)** obejmujący jedną lub wiele kolumn o unikatowym znaczeniu
- **może posiadać jeden lub wiele kluczy obcych (<<FK>>- foreign key)** jako kluczy głównych w powiązanych tabelach po stronie „1" powiązanych tabel.

Powiązanie (Association)

Wiąże dwa elementy modelu w związek strukturalny:

- połączenie może zawierać nazwy ról na każdym końcu, liczność wystąpień instancji tych elementów, kierunek oraz ograniczenia
- dla większej liczby powiązanych elementów jest przedstawiana jako romb

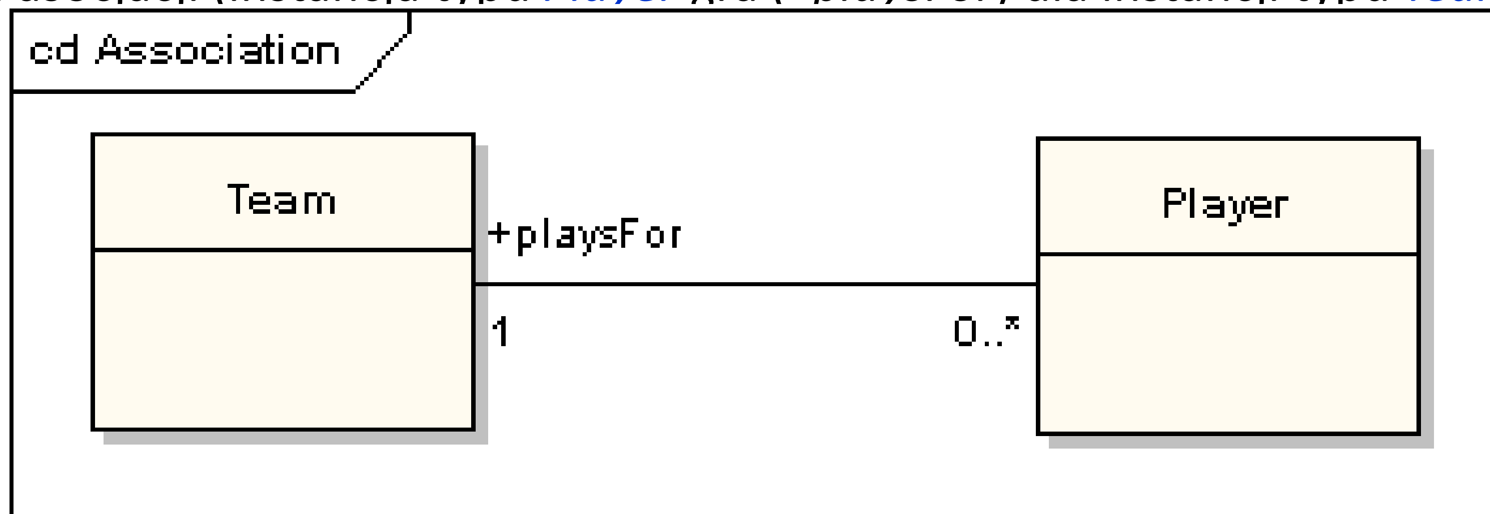


- **jest implementowana następująco:**
  1. **relacje wiele do jeden** lub **jeden do jeden: w obiekcie po stronie wiele lub jeden znajduje się referencja do obiektu z przeciwnej strony relacji (strony jeden)**
  2. **relacje jeden do wiele: kolekcja referencji instancji obiektów po stronie wiele w obiekcie po stronie jeden**

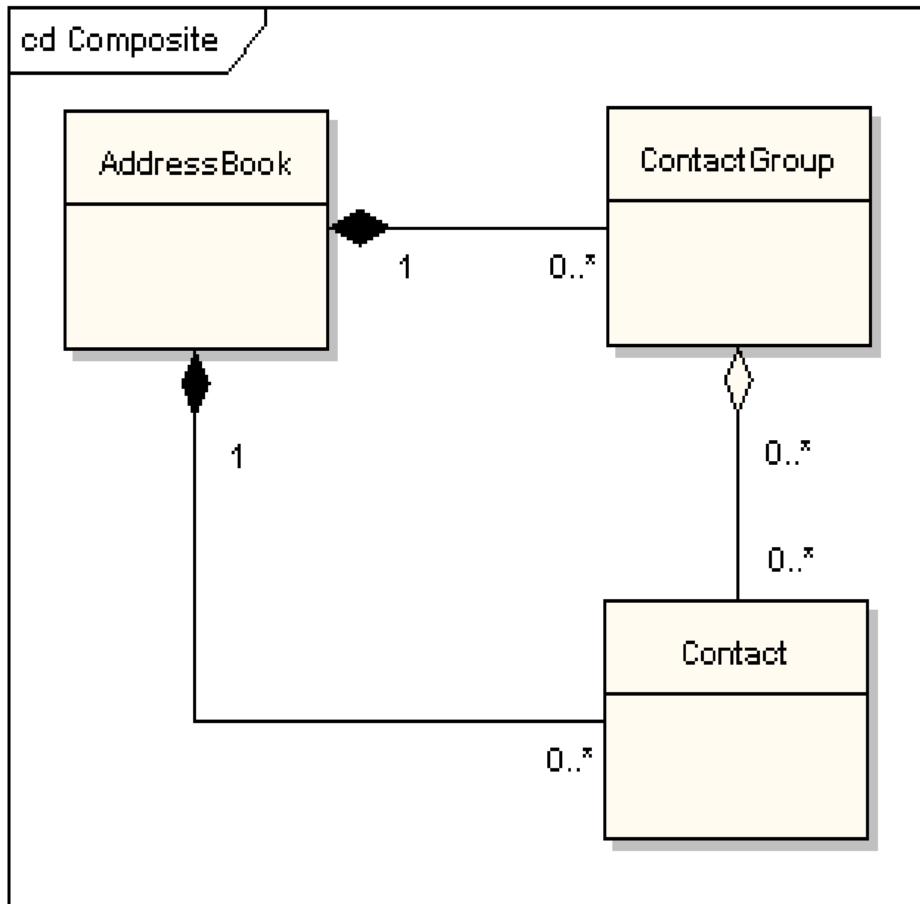
  (np. referencja do obiektu typu *Team* występuje w obiekcie typu *Player jako atrybut* oraz kolekcja referencji obiektów typu *Player* w obiekcie klasy *Team jako atrybut*)

Role

Role to oblicze, jakie prezentuje klasa przy jednym końcu drugiej klasie na drugim końcu asocjacji (instancja typu *Player* gra (*+playsFor*) dla instancji typu *Team*)



Agregacja (Aggregation) - oznacza elementy składające się z innych elementów

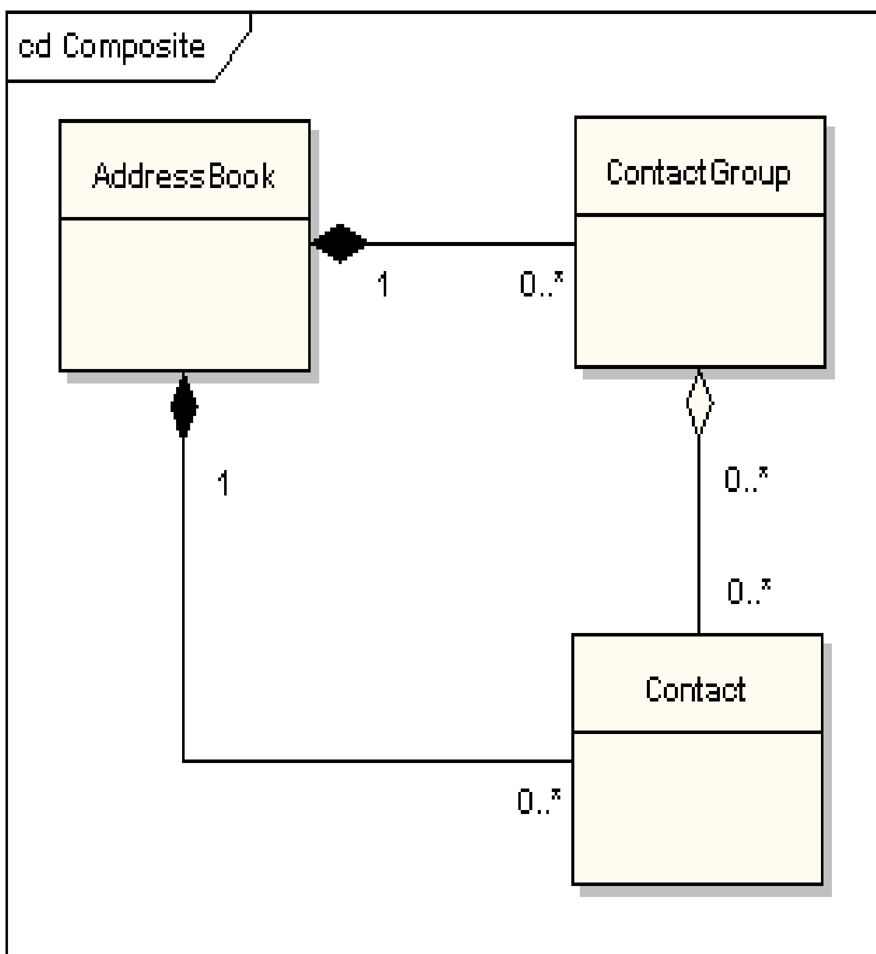


- jest tranzytywna, symetryczna, może być rekursywna
- jest wyrażana za pomocą rombów białych i czarnych, umieszczonych przy klasach agregujących
- **romby czarne-** silna agregacja (agregacja kompozytowa) oznaczająca, że przy usuwaniu obiektu klasy agregującej usuwany jest obiekt klasy agregowanej
- **romby białe** – słaba agregacja nie pociąga za sobą usuwania z pamięci obiektów agregowanych, gdy usuwany jest obiekt agregujący

Agregacja (Aggregation) – (cd) jest implementowana podobnie jak związek typu Association

Obiekt typu ***AddressBook***

- atrybut typu kolekcja referencji do obiektów typu ***Contact*** - *strona jeden do wiele*
- atrybut typu kolekcja referencji obiektów typu ***ContactGroup*** - *strona jeden do wiele*



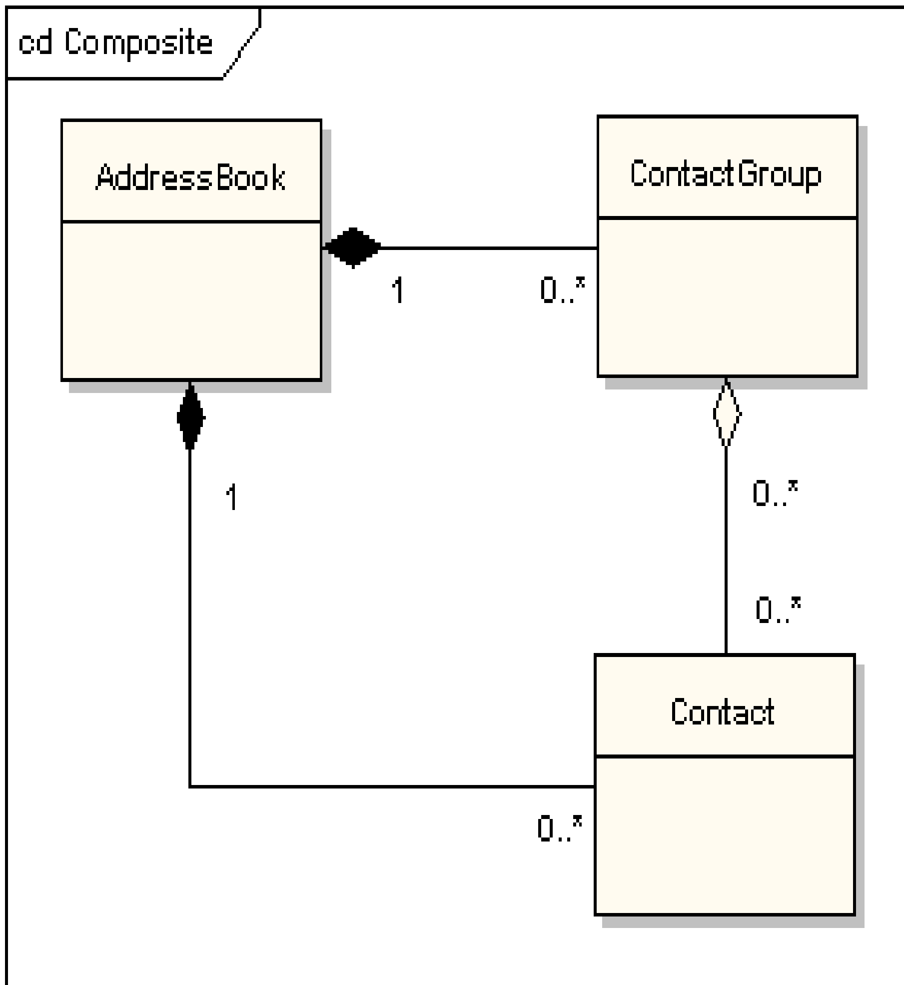
Obiekt typu ***ContactGroup:***

- atrybut typu kolekcja referencji do obiektów typu ***Contact*** (*strona wiele do wiele*)
- *atrybut typu* rerefencja obiektu typu ***AddressBook*** (*strona wiele do jeden*)

Obiekt typu ***Contact***

- atrybut typu kolekcja obiektów typu ***ContactGroup*** (*strona wiele do wiele*)
- *atrybut typu* rerefencja obiektu typu ***AddressBook*** (*strona wiele do jeden*)

Agregacja (Aggregation) – (cd) usuwanie obiektów powiązanych relacją agregacji



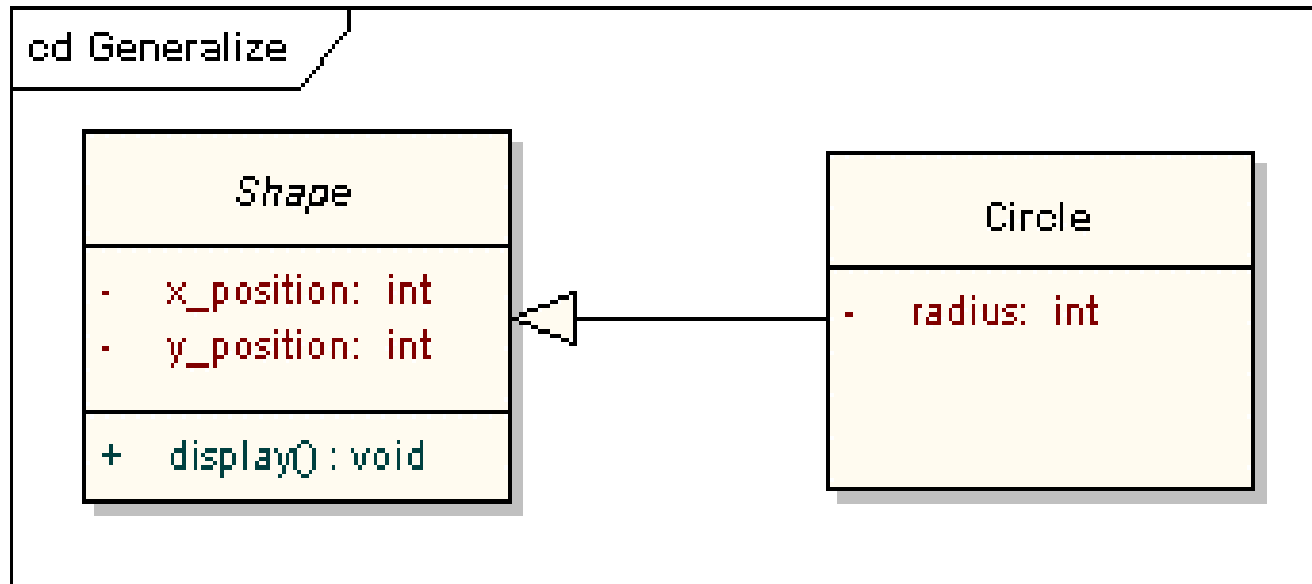
Obiekt typu *AddressBook* **agreguje w sposób silny** wiele obiektów klasy *ContactGroup oraz Contact*.

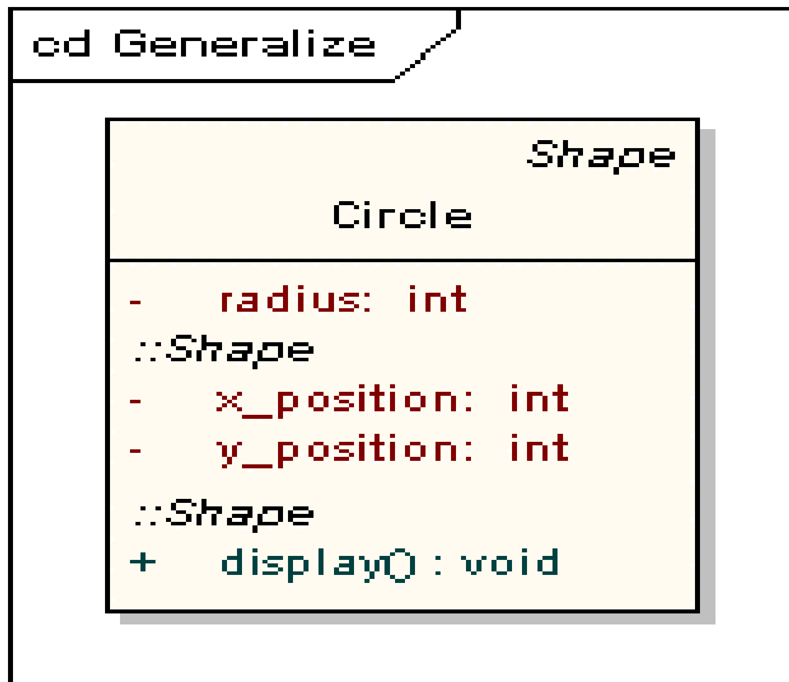
Usunięcie obiektu typu AddressBook pociąga za sobą usuniecie wszystkich obiektów typu *Contact* i *ContactGroup*,

Obiekt typu *ContactGroup* **agreguje w sposób słaby** wiele obiektów klasy *Contact*.

Usunięcie obiektu typu *Contact Gropup* nie pociąga za sobą usuwania obiektów typu *Contact*

Generalizacja czyli dziedziczenie (Generalization)

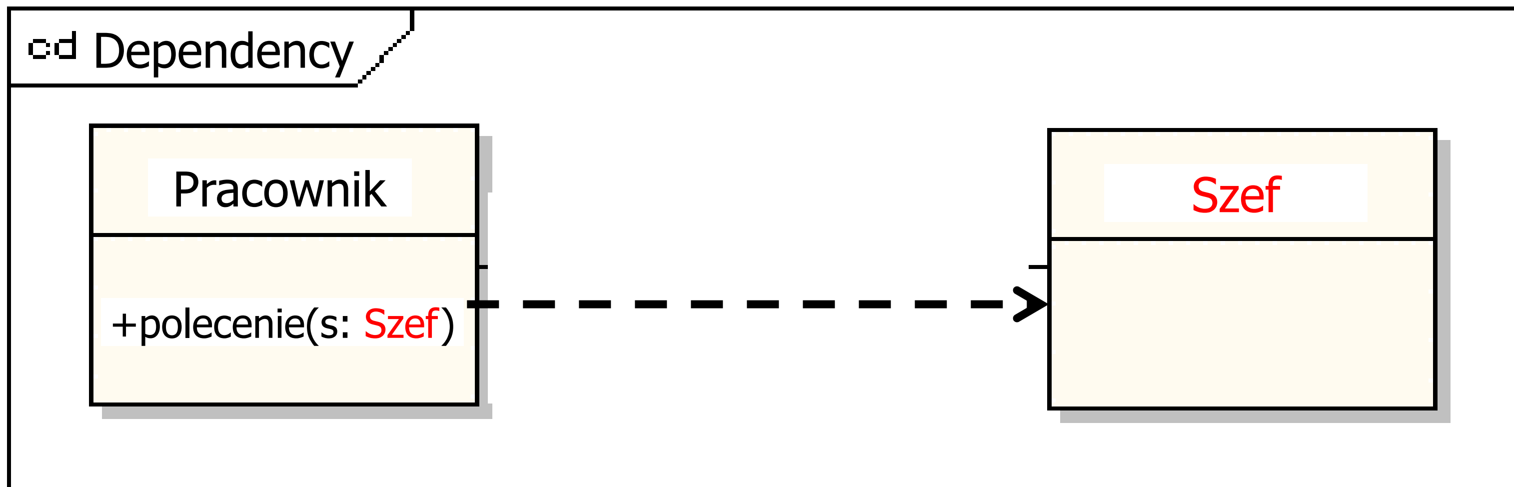




Używana do oznaczania **dziedziczenia**

- **strzałka wychodzi z klasy** dziedziczącej do klasy, po której dziedziczy
- np. klasa Circle dziedziczy atrybuty ***x_position, y_position*** **i metodę display() po klasie Shape** oraz dodaje atrybut **radius**

Zależność (Dependency)



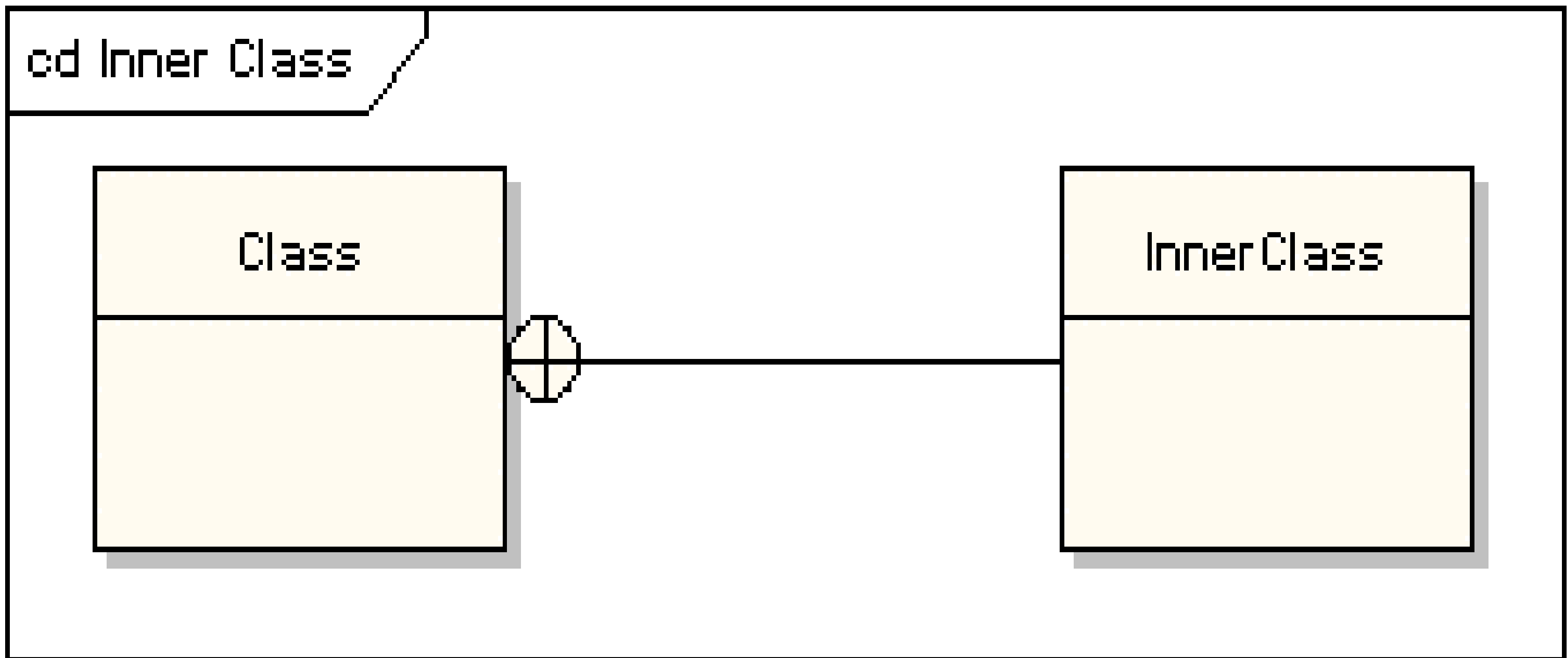
- zależności są używane do modelowania powiązań między elementami modelu we wczesnej fazie projektowania, jeśli nie można określić precyzyjnie typu powiązania. Stanowią one wtedy związek użycia **(<<usage>>)**.
- strałka przerywana wskazuje grotem na klasę, od której coś zależy.
- Później są one uzupełniane o stereotypy: «instantiate», «trace», «import» itp. lub zastąpione innym specjalizowanym połączeniem
- **implementacja zależności: klasa z operacją jest klasą zależną, natomiast parametr tej operacji jest obiektem typu klasy, od której coś zależy**

Specjalizacja zależności (Trace)

- łączy elementy modelu o tym samym przeznaczeniu, wymaganiach lub tym samym momencie zmian
- ma znaczenie informacyjne

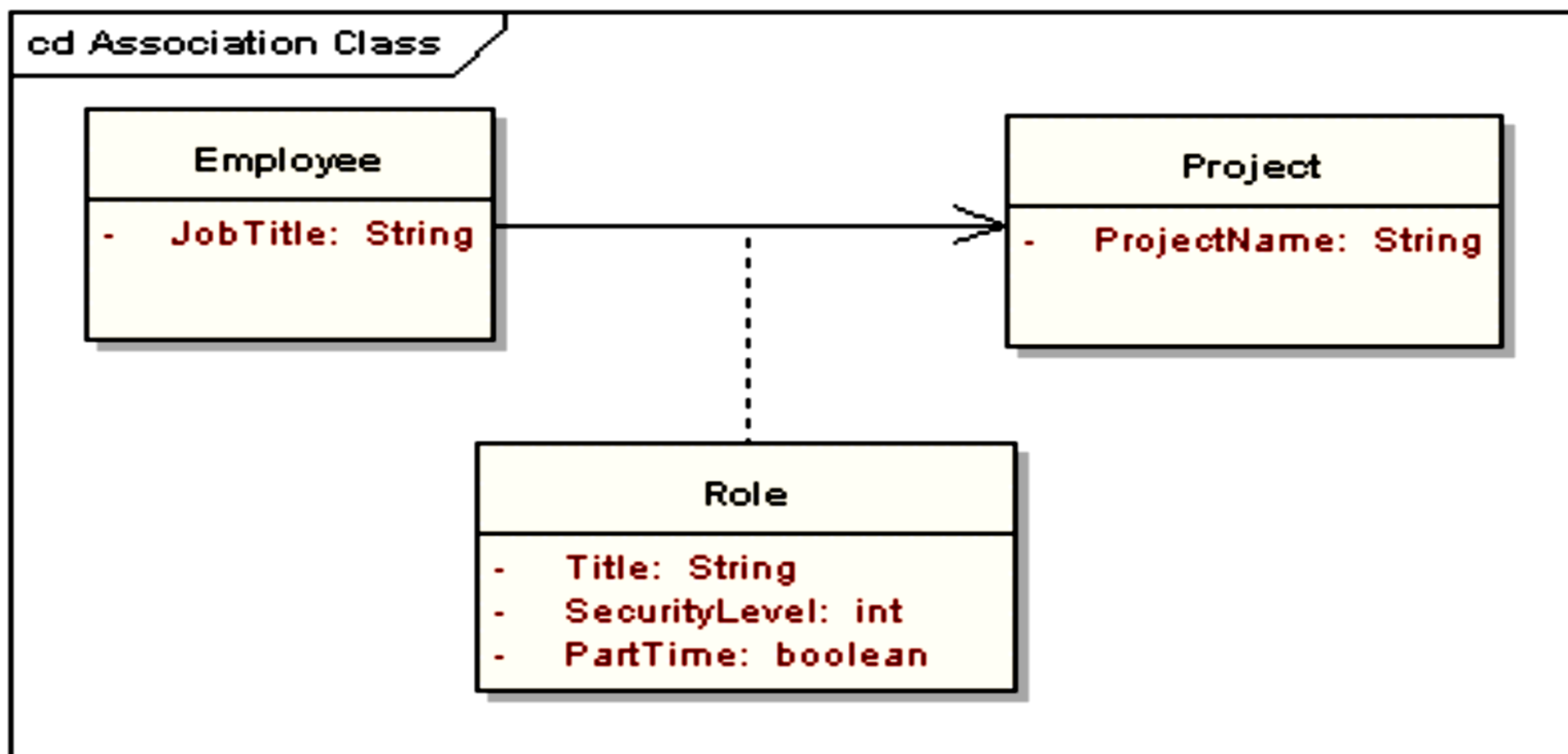
Zagnieżdżenie (Nesting)

- symbol zagnieżdżenia oznacza, że klasa, do której symbol jest dołączony, posiada zagnieżdżoną klasę dołączoną z drugiej strony zagnieżdżenia
- np. Klasa ***Class*** ma zagnieżdżoną klasę ***InnerClass***

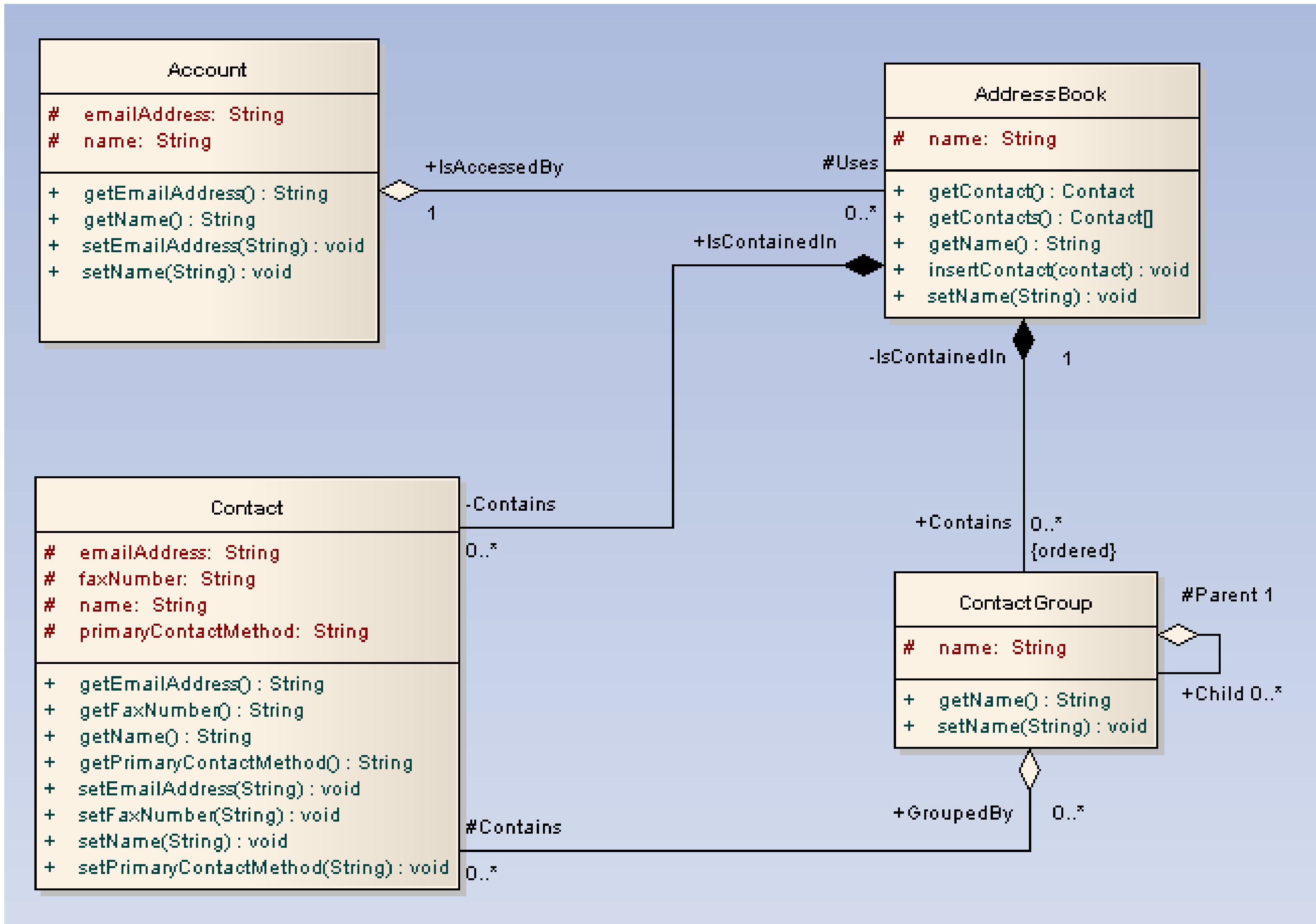


Klasa powiązań (Association Class)

- uzupełnia powiązane obiekty o atrybuty i metody
- np. powiązanie między projektem (obiekt klasy *Project*) a wykonawcą (obiekt klasy *Employee*) dodatkowo jest opisane za pomocą składowych obiektu klasy *Role*. Obiekt klasy *Role* jest przypisany w powiązaniu do jednej pary obiektów klas *Employee* i *Project*, które dodatkowo opisuje jako konkretnego pracownika wykonującego dany projekt



Podsumowanie – diagram klas



Diagramy klas, diagramy sekwencji

1. **Wprowadzenie**
2. **Syntaktyka diagramów klas**

   https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html

3. **Identyfikacja elementów diagramów klas**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]

Modele analizy i projektu – typy klas na diagramach klas

Produkt	Opis produktu (reprezentowanego w języku UML)
a) klasy typu „Control” Warstwy: prezentacji **biznesowa,** integracji	•reprezentują koordynację (*coordination*), sekwencje (*sequencing*), transakcje (*transactions*), sterowanie (*control*) •często są używane do hermetyzacji sterowania odniesionego do przypadku użycia dla każdej warstwy tzn hermetyzują warstwę biznesową dla warstwy prezentacji oraz warstwę integracji dla warstwy biznesowej; •klasy te modelują dynamikę systemu czyli główne akcje (*actions*) i przepływ sterowania (*control flows*) i przekazują działania do klas warstwy prezentacji, biznesowej oraz integracji ;

b) klasy typu „Entity” - formalnie obiekty realizowane przez system, często przedstawiane jako logiczne struktury danych (*logical data structure*) **Warstwa biznesowa**	• **używane do modelowania informacji o długim okresie istnienia i często niezmiennej (*persistent*);** • **klasy realizowana jako obiekty typu „real-life” lub zdarzenia typu „real-life”;** • **są wyprowadzane z modelu analizy** • **mogą zawierać specyfikację złożonego zachowania reprezentowanej informacji**
c) klasy typu „Boundary” **Warstwa klienta**	• klasy te reprezentują abstrakcje: okien, formularzy, interfejsów komunikacyjnych, interfejsów drukarek, sensorów, terminali i API (również nieobiektowych); • jedna klasa odpowiada jednemu użytkownikowi typu aktor • używane do modelowania interakcji między systemem i aktorami czyli użytkownikami (*users*) lub zewnętrznymi systemami;

Identyfikacja klas

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Zidentyfikuj zbiór klas, które współpracują ze sobą w celu wykonania poszczególnych czynności
- Określ zbiór zobowiązań każdej klasy
- Rozważ zbiór klas jako całość: podziel na mniejsze te klasy, które mają zbyt wiele zobowiązań; scal w większe te klasy, które mają zbyt mało zobowiązań
- Rozpatrz sposoby wzajemnej kooperacji tych klas i porozdzielaj ich zobowiązania tak, aby żadna z nich była ani zbyt złożona ani zbyt prosta
- Elementy nieprogramowe (urządzenia) przedstaw w postaci klasy i odróżnij go za pomocą własnego stereotypu; jeśli ma on oprogramowanie, może być traktowany jako węzeł diagramu klas w celu rozwijania tego oprogramowania
- Zastosuj typy pierwotne (tabele, wyliczenia, typy proste np. boolean itp)

Identyfikacja związków: zależność (Dependency)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

Modelowanie zależności

- Utworzyć zależności między klasą z operacją, a klasą użytą jako parametr tej operacji
- Stosuj zależności tylko wtedy, gdy modelowany związek nie jest strukturalny

Identyfikacja związków: generalizacja czyli dziedziczenie (Generalization)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Ustaliwszy zbiór klas poszukaj zobowiązań, atrybutów i operacji wspólnych dla co najmniej dwóch klas
- Przenieś te wspólne zobowiązania, atrybuty i operacje do klasy bardziej ogólnej; jeśli to konieczne, utwórz nową klasę, do której zostaną przypisane te właśnie byty (uważaj z wprowadzaniem zbyt wielu poziomów generalizacji)
- Zaznacz, że klasy szczegółowe dziedziczą po klasie ogólnej, to znaczy uwzględnij uogólnienia biegnące od każdego potomka do bardziej ogólnego przodka
- Stosuj uogólnienia tylko wtedy, gdy masz do czynienia ze związkiem „jest rodzajem”; dziedziczenie wielobazowe często można zastąpić agregacją
- Wystrzegaj się wprowadzania cyklicznych uogólnień
- Utrzymuj uogólnienia w pewnej równowadze; krata dziedziczenia nie powinna być zbyt głęboka (pięć lub więcej poziomów już budzi wątpliwości) ani zbyt szeroka (lepiej wprowadzić pośrednie klasy abstrakcyjne)

Identyfikacja związków strukturalnych: powiązanie (Association) , agregacja (Aggregation)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Rozważ, czy w wypadku każdej pary klas jest konieczne przechodzenie od obiektów jednej z nich do obiektów drugiej
- Rozważ, czy w wypadku każdej pary klas jest konieczna inna interakcja między obiektami jednej z nich a obiektami drugiej niż tylko przekazywanie ich jako parametrów; jeśli tak, uwzględnij powiązanie między tymi klasami, w przeciwnym wypadku jest to zależność użycia. Ta metoda identyfikacji powiązań jest oparta na zachowaniu
- Dla każdego powiązania określ liczebność (szczególnie wtedy, kiedy nie jest to 1 - wartość domyślna) i nazwy ról (ponieważ ułatwiają zrozumienie modelu)
- Jeśli jedna z powiązanych klas stanowi strukturalną lub organizacyjną całość w porównaniu z klasami z drugiego końca związku, które wyglądają jak części, zaznacz przy niej specjalnym symbolem, że chodzi o agregację.
- Stosuj powiązania głównie wtedy, kiedy między obiektami zachodzą związki strukturalne

Identyfikacja wzorców projektowych (wstęp do wykładu 6)

- Dobrze zbudowany system obiektowy jest pełen wzorców obiektowych
- Wzorzec to zwyczajowo przyjęte rozwiązanie typowego problemu w danym kontekście
- Strukturę wzorca przedstawia się w postaci diagramu klas
- Zachowanie się wzorca przedstawia się za pomocą diagramu sekwencji
- Wzorce projektowe: Wzorzec reprezentuje powiązanie problemu z rozwiązaniem **(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)**

- Każdy wzorzec składa się z trzech części, które wyrażają związek między konkretnym kontekstem, problemem i rozwiązaniem **(Christopher Aleksander)**
- Każdy wzorzec to trzyczęściowa reguła, która wyraża związek między konkretnym kontekstem, rozkładem sił powtarzającym się w tym kontekście i konfiguracją oprogramowania pozwalająca na wzajemne zrównoważenie się tych sił w celu rozwiązania zadania. **(Richar Gabriel)**
- Wzorzec to pomysł, który okazał się użyteczny w jednym rzeczywistym kontekście i prawdopodobnie będzie użyteczny w innym. **(Martin Fowler)**

Przykład identyfikacji klas
Analiza wspólności (perspektywa koncepcji, model analizy – wykład 1)

Przykład z wykładu 3, Przykład 3 z wykładu 4 (cd)

Diagram przypadków użycia (wykład 4 część 1, przykład 3)

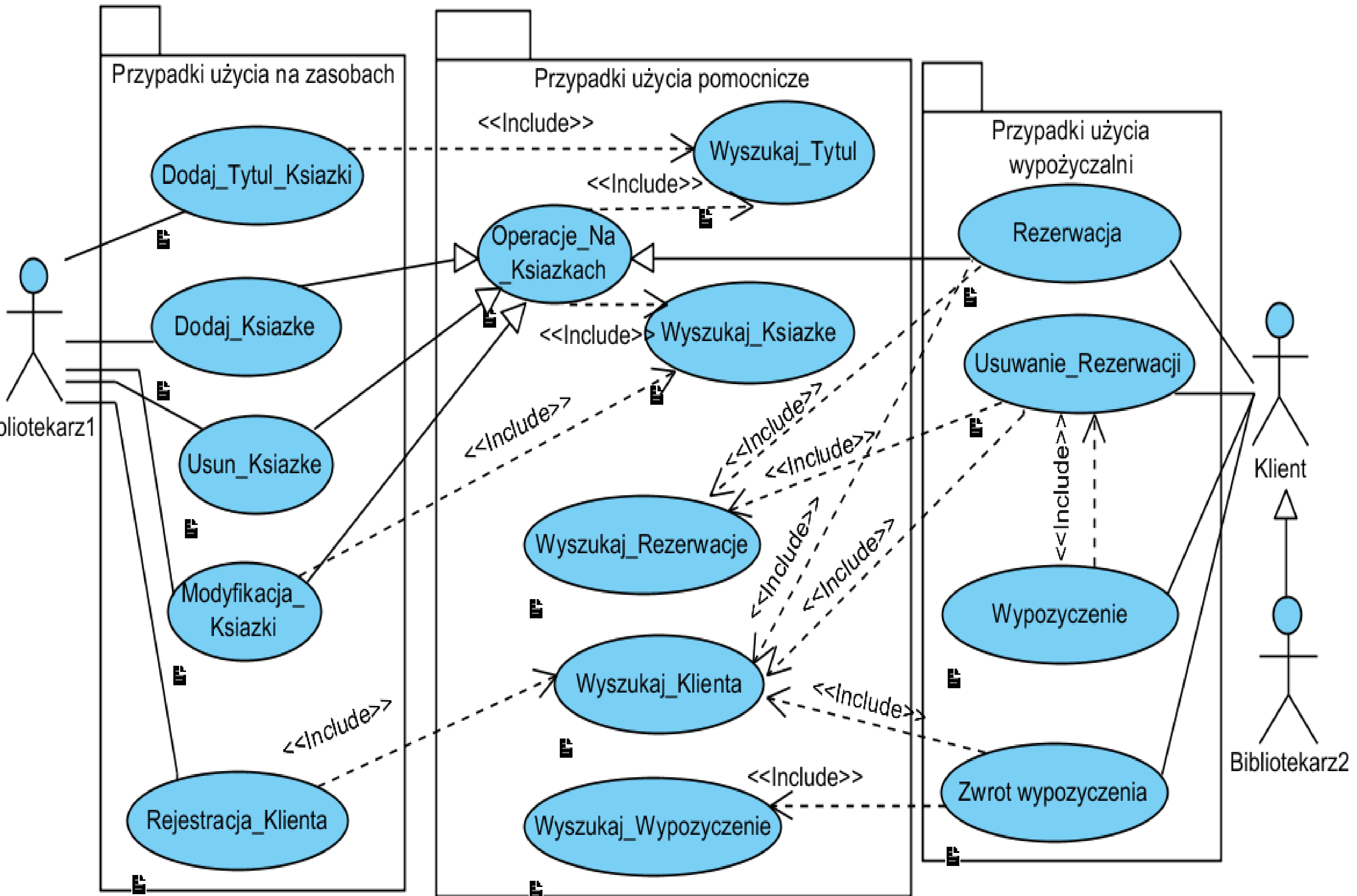
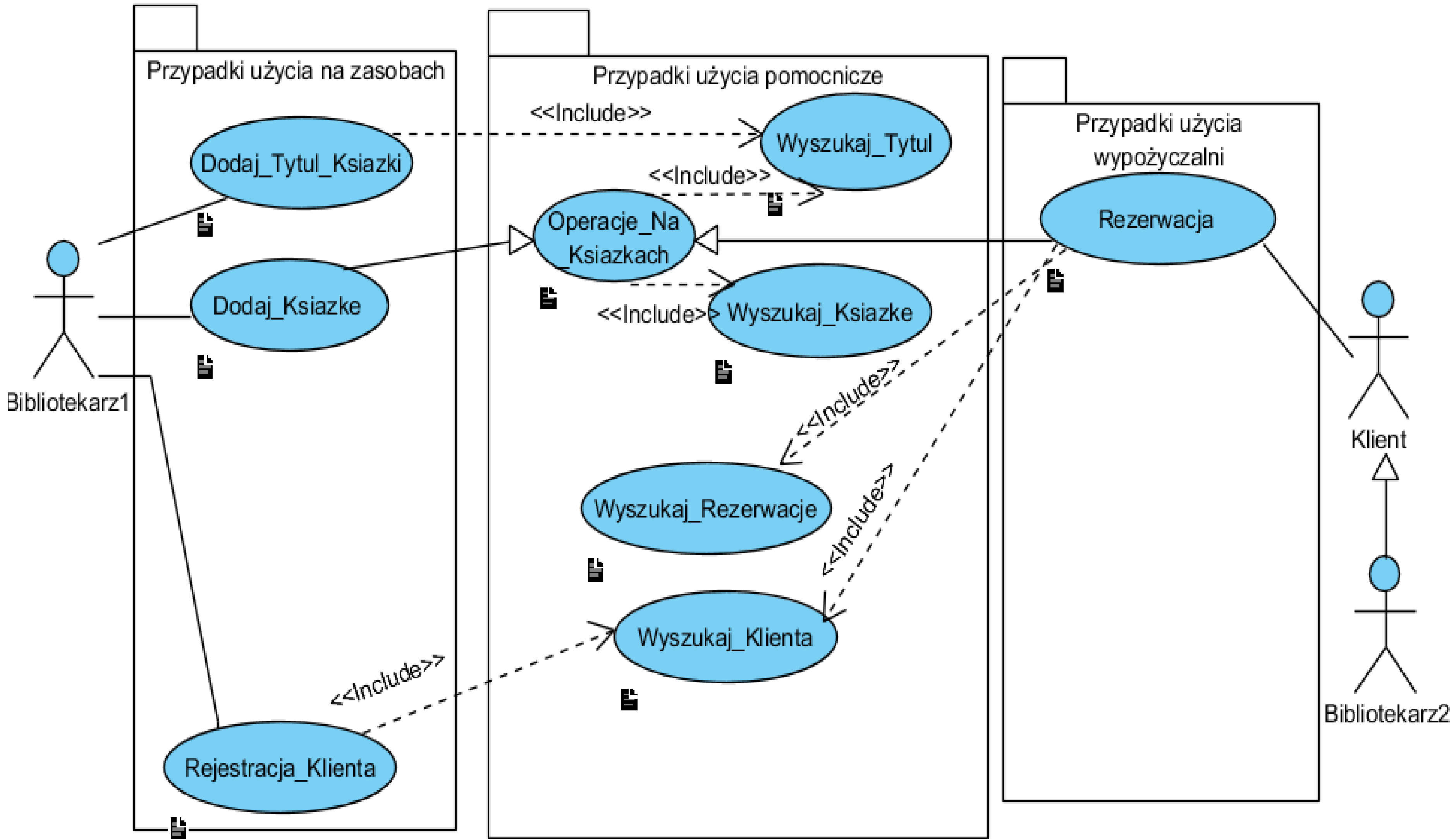


Diagram przypadków użycia (wykład 4 część 1, przykład 3) – wybrany fragment



Identyfikacja klas – etap 1 Analiza wspólności (perspektywa koncepcji, model analizy – wykład 1)

- Wykryto **cztery główne klasy** typu **„Entity”** ze względu na odpowiedzialność:
  - **TitleBook** (PU: **Dodaj_Tytul_Ksiazki, Dodaj_Ksiazke, Rezerwacja**),
  - **Book** (PU: **Dodaj_Ksiazke, Rezerwacja**),
  - **Client** (PU: **Rejestracja_Klienta, Rezerwacja**)
  - **Reservation** (PU: **Rezerwacja**)

Identyfikacja klas – etap 2
Analiza zmienności (perspektywa specyfikacji, model projektowy – wykład 1)

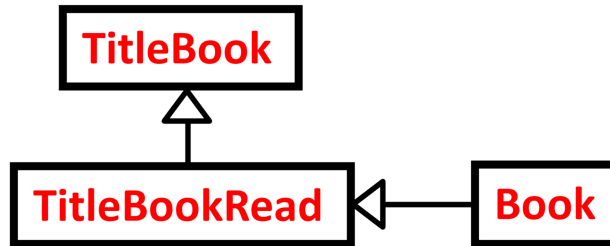
- Wykryto **dziedziczenie** w właściwościach tytułów książek, które określają, które książki są nagrane, a które są drukowane Zdefiniowano klasę pochodną:
  - **TitleBookRead** typu **„Entity”**, która dziedziczy od klasy **TitleBook i zawiera informację o aktorze, która czyta książkę na nagraniu** (PU: **Dodaj_Tytul_Ksiazki, Dodaj_Ksiazke, Rezerwacja**)

Analiza wspólności i zmienności - identyfikacja typów relacji

Oszacowania dla przyjętego modelu powiązań:

- Liczba obiektów z typu **TitleBookRead** : 5000,
- Przybliżony największy rozmiar obiektu typu **TitleBookRead**: R1
- Przybliżony największy rozmiar obiektu typu **Book**: R2
- Średnia liczba książek na 1 obiekt z rodziny typu **TitleBookRead**: 50
- Liczba wszystkich książek: 250000

Oszacowania dla częściowo równoważnego modelu dziedziczenia:



- Liczba obiektów z rodziny **TitleBookRead**: 5000,
- Przybliżony największy rozmiar obiektu z tyupu **TitleBookRead**: R1
- Przybliżony największy rozmiar obiektu typu **Book**: R1+R2
- Średnia liczba książek na 1 obiekt z rodziny typu **TitleBook**: 50
- Liczba wszystkich książek: 250000

	Przyjęty model powiązań	**Model oparty na dziedziczeniu**
Rozmiar pamięci	**5000*R1 + 5000*50*R2**	5000*R1 + 5000*50*(R1 + R2)
Liczba przeszukań obiektów typu **Book**	**od 1 do (5000 + 50)**	Od 1 do 5000*50

Analiza zmienności (c.d)

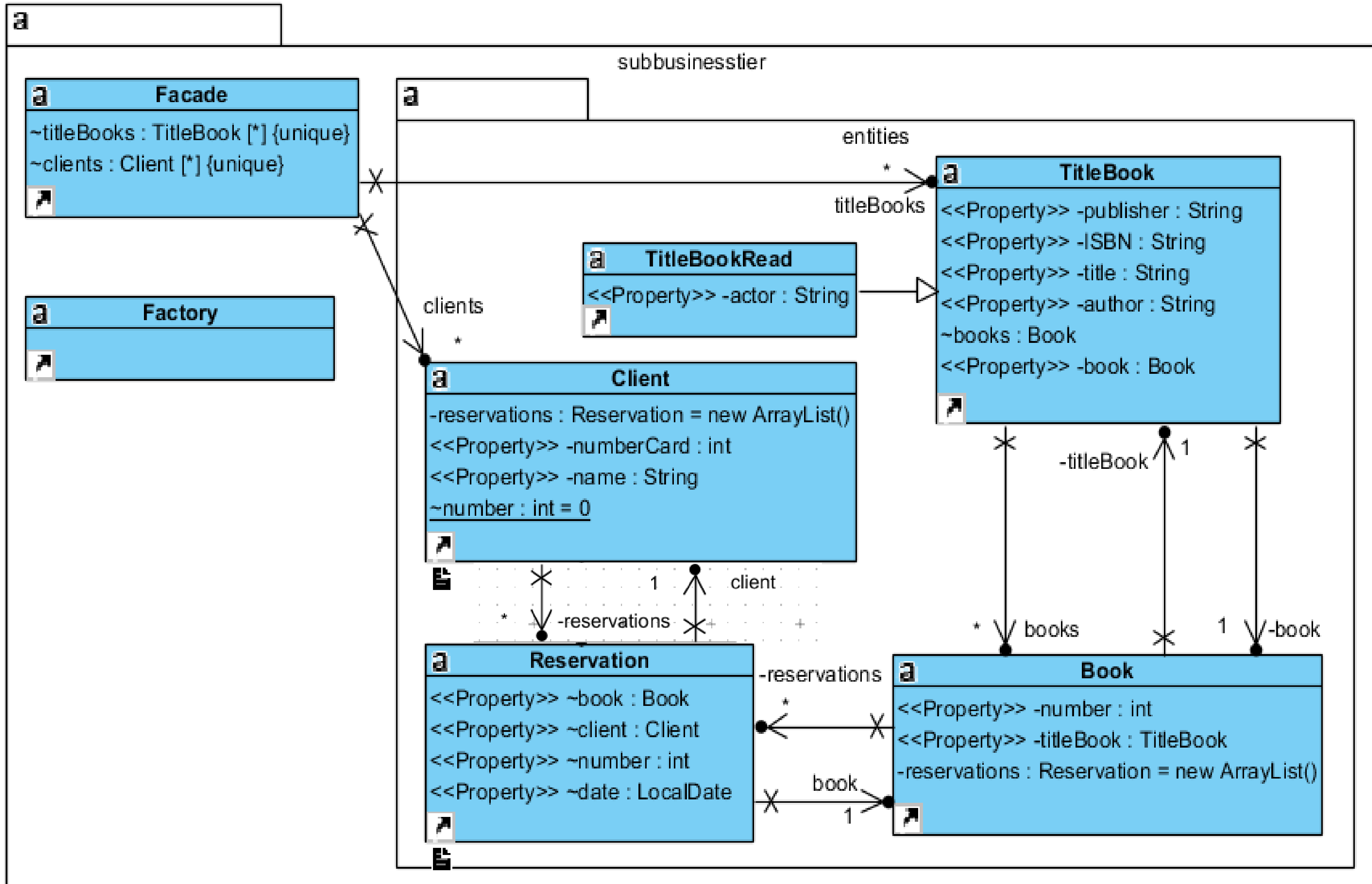
- **Wykryto związki:**
  - asocjacji dwukierunkowejmiędzy obiektem typu **TitleBook** i obiektami typu **Book**: obiekt typu **TitleBook** zarządza zbiorem obiektów typu **Book**, a każdy obiekt typu Book jest powiązany tylko z jednym obiektem typu TitleBook, który określa jego tytuł oraz, czy jest książką nagraną, czy też papierową (PU Dodaj_Tytul, Dodaj_Ksiazke).
  - asocjacji dwukierunkowej między klasą **Rezerwacja** i klasami **Book** i **Client**. Obiekty typu **Client** i **Book** posiadają zbiór obiektów typu **Reservation**, a obiekt typu **Reservation** jest powiązany tylko z jednym obiektem typu **Client** i jednym obiektem typu **Book**
  - Podstawą identyfikacji jest PU **Rezerwacja**

Analiza zmienności (c.d)

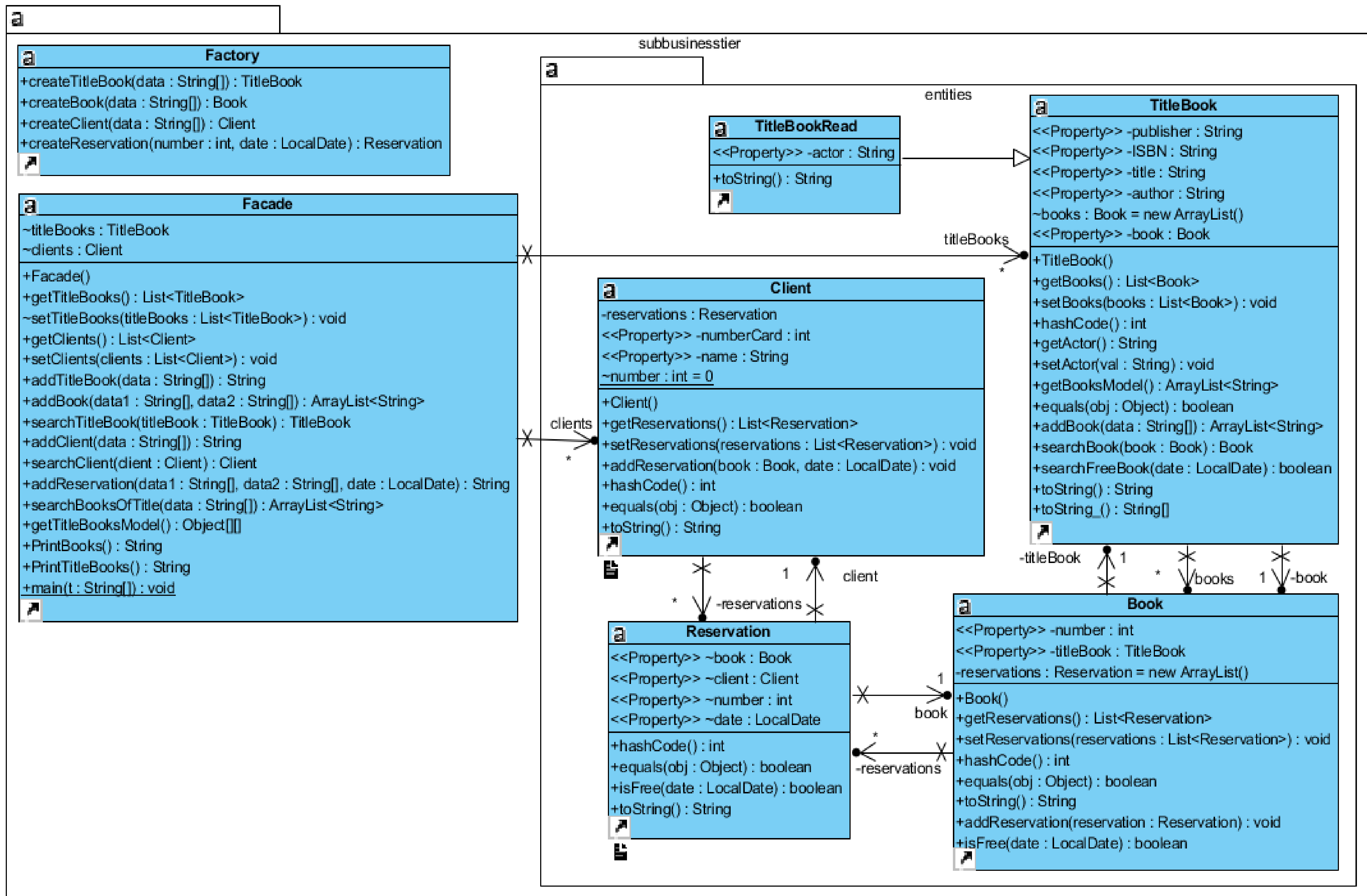
- **Zastosowano wzorzec strukturalny typu Fasada:**
  - klasę fasadową **Facade** typu „**Control**” do oddzielenia przetwarzania obiektów typu „**Entity**” od pozostałej części systemu

- **Zastosowano wzorzec wytwórczy typu Fabryka**
  - klasę typu „**Control**” jako fabrykę obiektów (**Factory**) do tworzenia różnych typów produktów – czyli obiektów typu **TitleBook**, **TitleBookRead**, **Book**, **Reservation, Client**

Wynik
Początkowa definicja diagramu klas – zdefiniowano powiązania między klasami



Rezultat – diagram klas uzyskany w procesie projektowania (przebieg pokazany w dodatku do wykładu 5)



Klasa Facade udostępnia metody logiki biznesowej – implementacja przypadków użycia wywoływanych przez aktorów na diagramie przypadków użycia

```
package subbusinesstier;
import java.time.LocalDate;
import java.time.Month;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import subbusinesstier.entities.Client;
import subbusinesstier.entities.TitleBook;
public class Facade {
    List<TitleBook> titleBooks;
    List<Client> clients;
    public Facade()                                        {   }
    public List<TitleBook> getTitleBooks()                 {   }
    public void setTitleBooks(List<TitleBook> titleBooks)  {   }
    public List<Client> getClients()                       {   }
    public void setClients(List<Client> clients)           {   }
```

```
public TitleBook searchTitleBook(TitleBook titleBook) {}

public Client searchClient(Client client) {}

public String addClient(String data[]) {}

public String addTitleBook(String data[]) {}

public ArrayList<String> addBook(String data1[], String data2[]) {}

public String addReservation(String data1[], String data2[], LocalDate date) {}

//pomocnicze metody
    public ArrayList<String> searchBooksOfTitle(String data[])  {}
    public Object[][] getTitleBooksModel()                      {}
    public String PrintBooks()                                  {}
    public String PrintTitleBooks()                             {}
    public static void main(String t[])                         {}
}
```

PU Operacje_Na_Ksiazkach

PU Rejestracja_Klienta

PU Dodaj_Tytul_Ksiazki

PU Dodaj_Ksiazke

PU Rezerwacja

Diagramy klas, diagramy sekwencji

1. **Wprowadzenie**
2. **Syntaktyka diagramów klas**

   https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html
3. **Identyfikacja elementów diagramów klas**

   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]
4. **Diagramy sekwencji UML**

   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

Diagramy UML 2 – część czwarta

Na podstawie

UML 2.0 Tutorial

https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html

Dwa rodzaje diagramów UML 2

Diagramy UML <u>modelowania strukturalnego</u>

- **Diagramy pakietów**
- ***Diagramy klas***
- **Diagramy obiektów**
- **Diagramy mieszane**
- **Diagramy komponentów**
- **Diagramy wdrożenia**

Diagramy UML <u>modelowania zachowania</u>

- ***Diagramy przypadków użycia***
- ***Diagramy aktywności***
- **Diagramy stanów**
- **Diagramy komunikacji**
- ***Diagramy sekwencji***
- **Diagramy czasu**
- **Diagramy interakcji**

Diagramy sekwencji (Sequence Diagrams)

- wyrażają **interakcje w czasie** (wiadomości wymieniane między obiektami jako poziome strzałki wychodzące od linii życia jednego obiektu i wchodzące do linii życia drugiego obiektu)
- wyrażają dobrze **komunikację** między obiektami i zarządzanie przesyłaniem wiadomości
- **nie są używane do wyrażania złożonej logiki proceduralnej**
- **są używane do modelowanie scenariusza przypadku użycia**

Linie życia (Lifelines)

Linie życia reprezentują indywidualne uczestniczenie obiektu w diagramie. Posiadają one często prostokąty zawierające nazwę i typ obiektu.

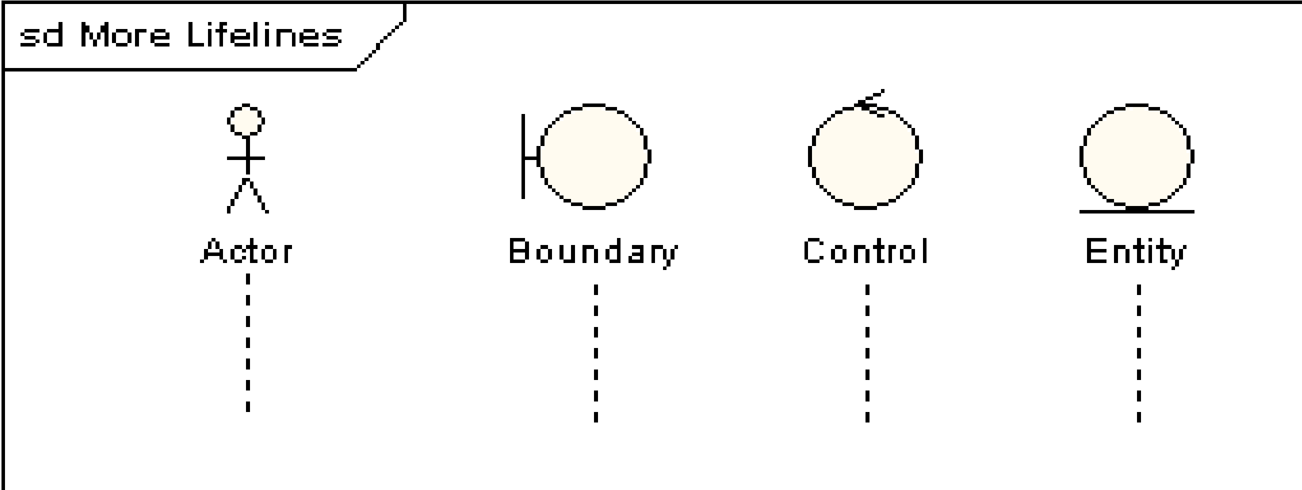
sd Lifelines

Self

Lifeline

Instance :Class

Czasem diagram sekwencji zawiera **linię życia aktora**. Oznacza to, że właścicielem diagramu sekwencji jest **przypadek użycia**. Elementy oznaczające ***obiekty typu „boundary”, „control”*** , ***„Entity”*** mają również swoje linie życia.

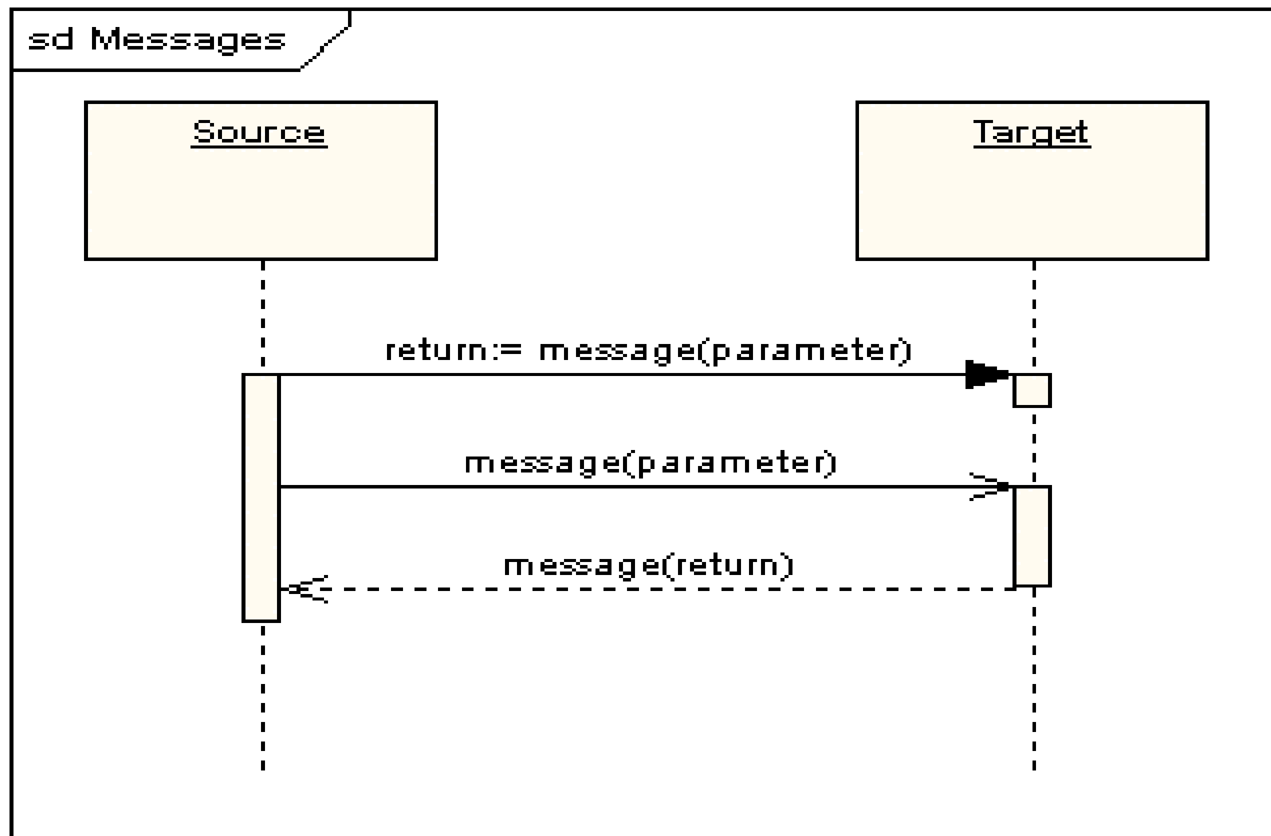


Wiadomości (Messages)

- są wyświetlane jako strzałki.
- mogą być ***kompletne, zgubione i znalezione;***
- mogą być ***synchroniczne i asychroniczne***
- Mogą być typu wywołanie operacji **(*call)*** lub sygnał (***signal***)
- **dla wywołań operacji (call) wyjście strzałki z linii życia oznacza, że obiekt ten wywołuje metodę obiektu, do którego strzałka dochodzi**

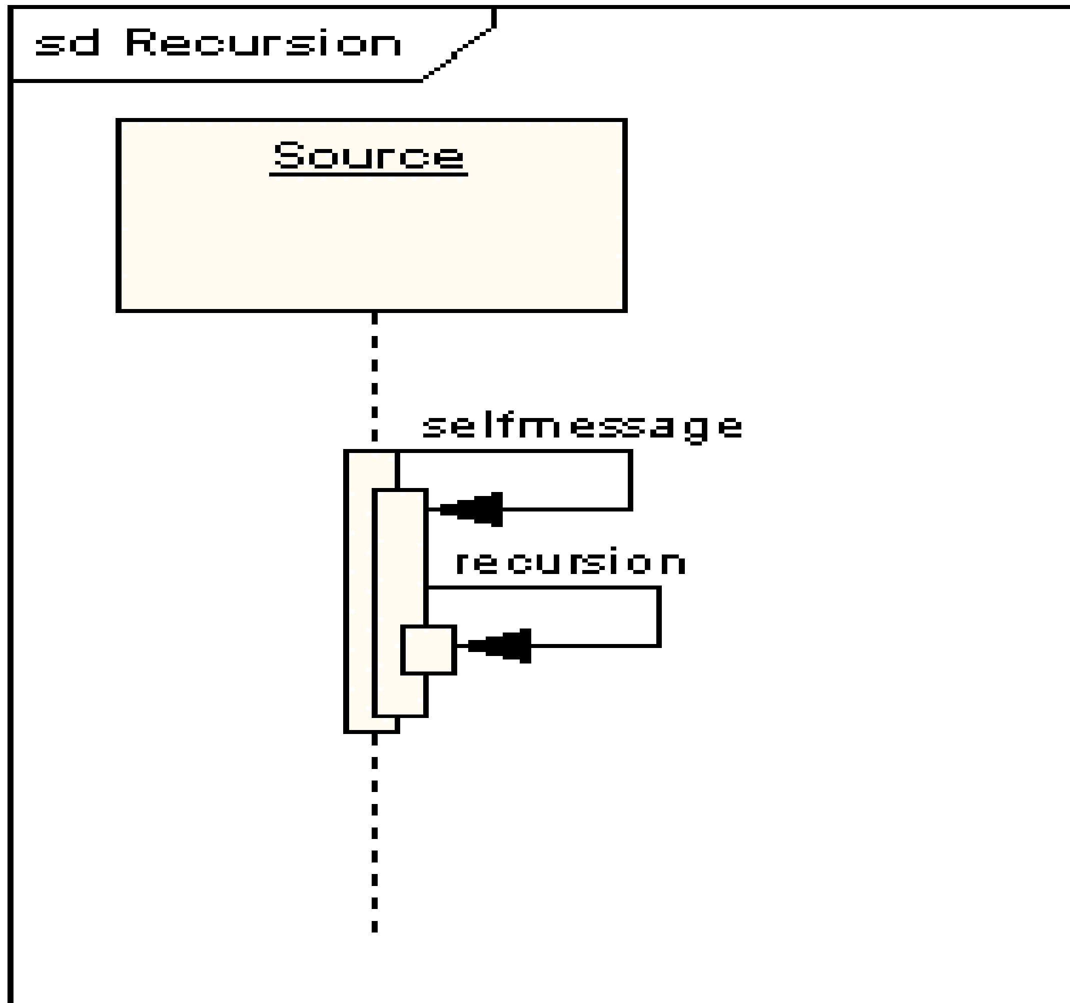
Wykonywanie interakcji (Execution Occurrence)

1. pierwsza wiadomość jest synchroniczna, kompletna i posiada return (wywołanie metody obiektu Target przez obiekt przez Source),
2. druga wiadomość jest asynchroniczna (wywołanie metody obiektu Target przez obiekt przez Source),
3. trzecia wiadomość jest asynchroniczną wiadomością typu return (przerywana linia – return metody asynchronicznej obiektu Target).



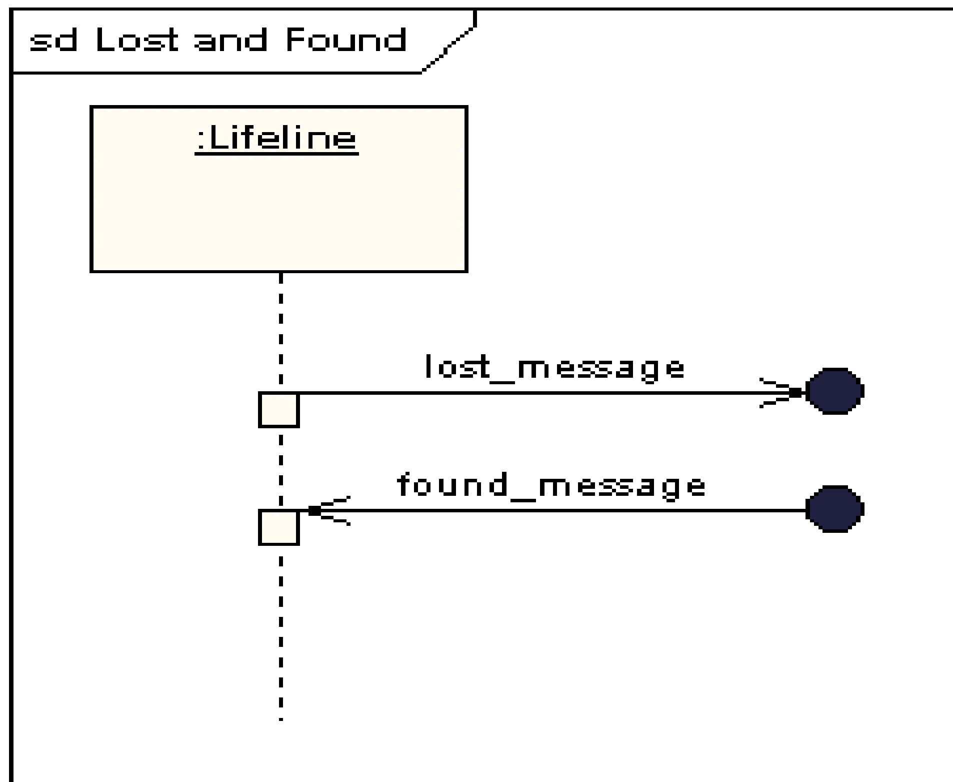
Własne wiadomości (Self Message)

Własne wiadomości reprezentują rekursywne wywoływanie operacji albo jedna operacja wywołuje inną operację należącą do tego samego obiektu.



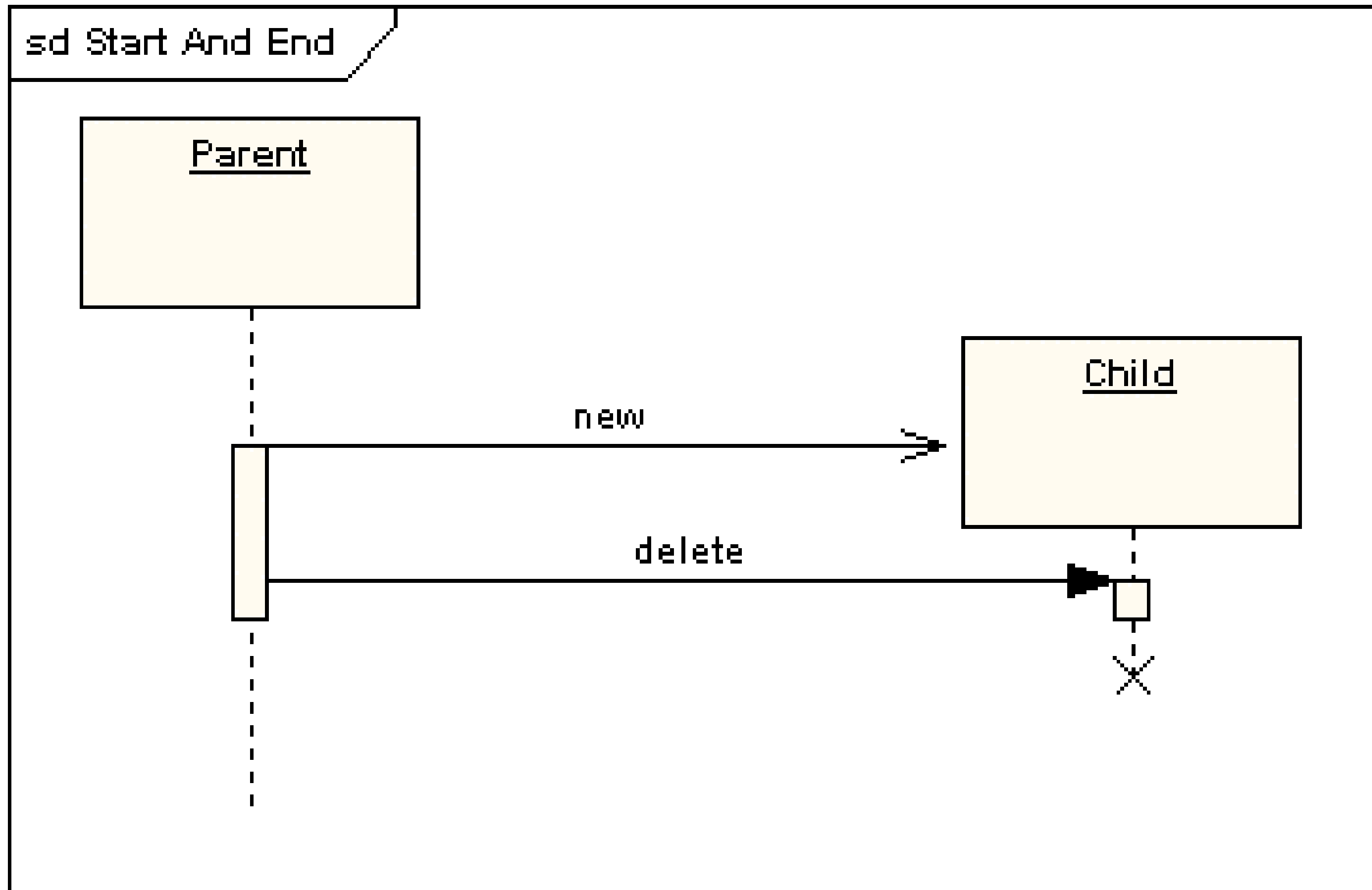
Zgubione i znalezione wiadomości (Lost and Found Messages)

- **Zgubione wiadomości** są wysłane i nie docierają do obiektu docelowego lub nie są pokazane na bieżącym diagramie.
- **Znalezione wiadomości** docierają od nieznanego nadawcy albo od nadawcy, który nie jest pokazany na bieżącym diagramie.



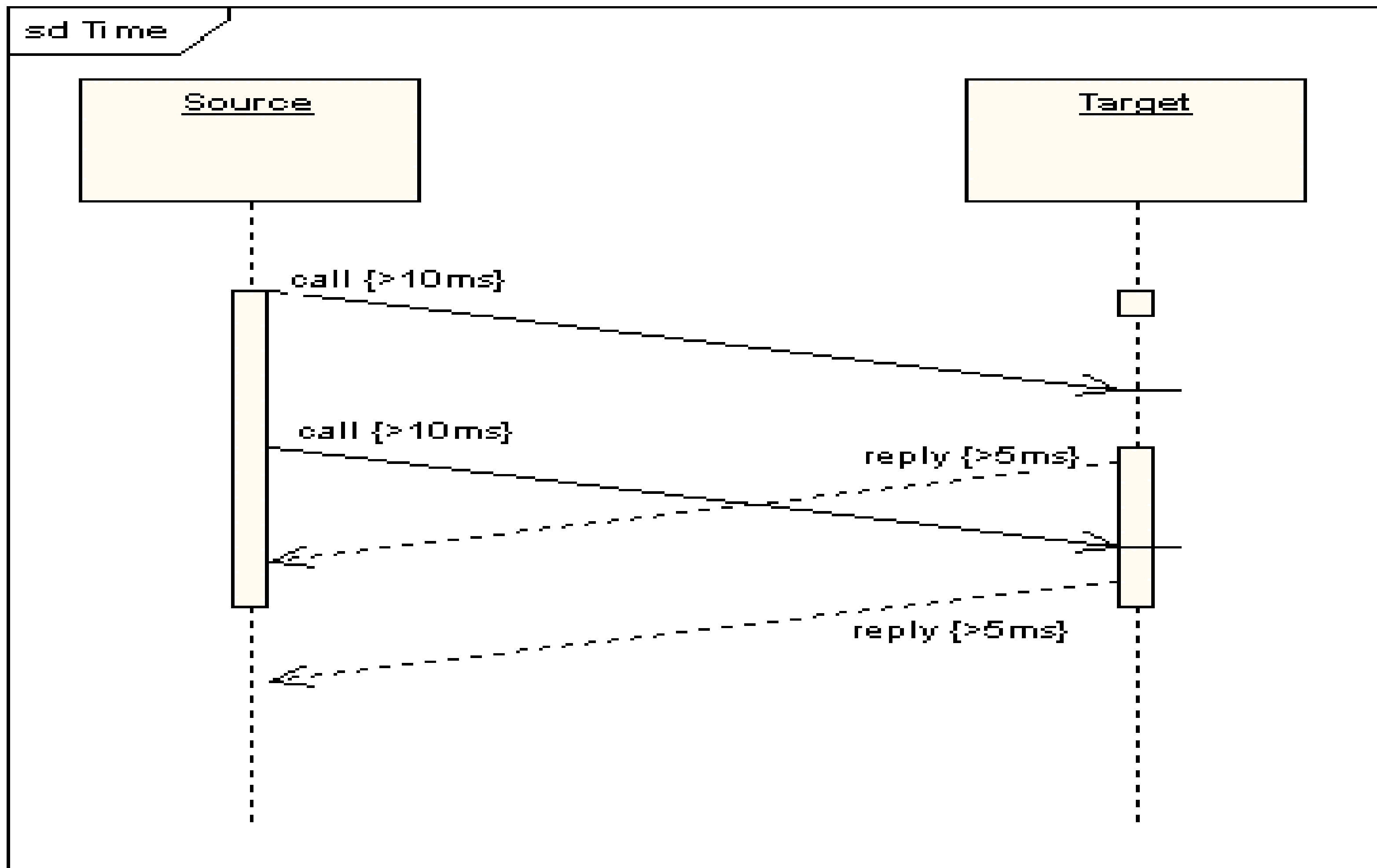
Start linii życia i jej koniec (Lifeline Start and End)

Oznacza to tworzenie (typu **Create Message**) i usuwanie obiektu (symbol **X**)



Ograniczenia czasowe (Duration and Time Constraints)

Domyślnie, wiadomość jest poziomą linią. W przypadku, gdy należy ukazać opóźnienia czasu wynikające z czasu podjętych akcji przez obiekt po otrzymaniu wiadomości, wprowadza się ukośne linie wiadomości.

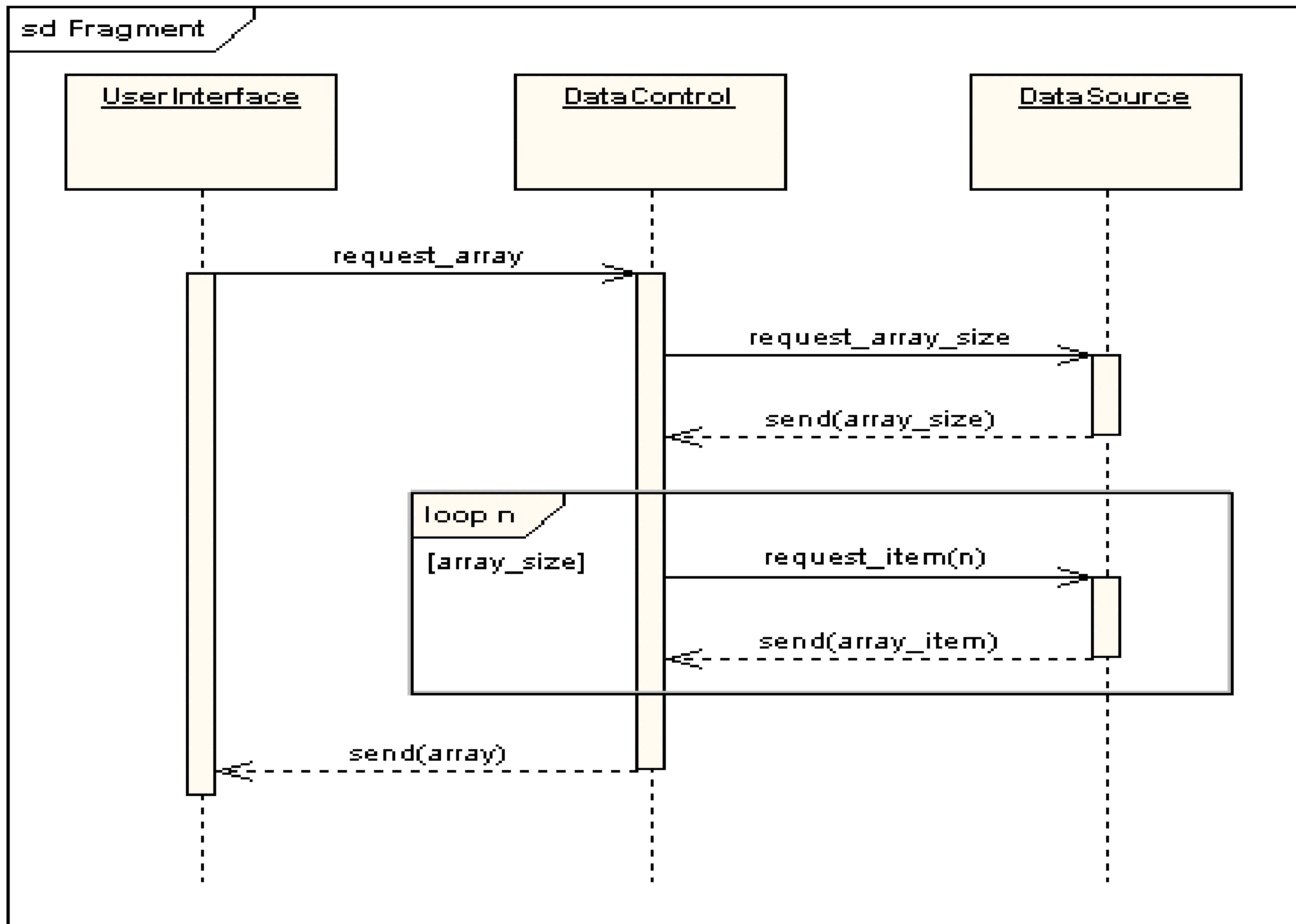


Złożone modelowanie sekwencji wiadomości

Fragmenty ujęte w ramki umożliwiają:

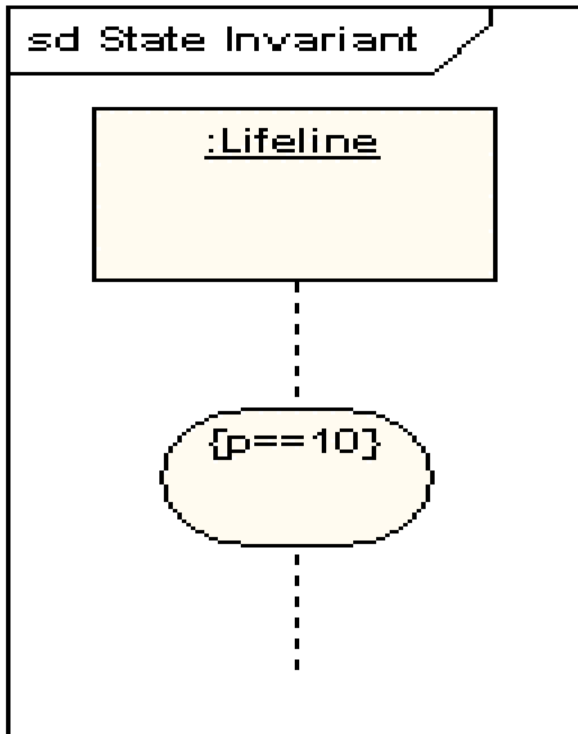
1. **fragmenty opcjonalne** (oznaczone "**opt**") modelują konstrukcje np **if…then…else**
2. **fragmenty alternatywne** (oznaczone "**alt**") modelują konstrukcje np **switch**.
3. **pętla** (oznaczony "**loop**") oznacza powtarzanie interakcji we fragmencie.
4. **fragment break** modeluje alternatywną sekwencję zdarzeń dla pozostałej części diagramu.
5. **fragment równoległy** (oznaczony "**par**") modeluje proces równoległy.
6. **słaba sekwencja** (oznaczona "**seq**") zamyka pewna liczbę sekwencji, w której wszystkie wiadomości muszą być wykonane przed rozpoczęciem innych wiadomości z innych fragmentów, z wyjątkiem tych wiadomości, **które nie dzielą linii życia oznaczonego fragmentu**.
7. **dokładna sekwencja** (oznaczona jako "**strict**") zamyka wiadomości, które muszą być wykonane w określonej kolejności
8. **fragment negatywny** (oznaczony "**neg**") zamyka pewną liczbę niewłaściwych wiadomości
9. **fragment krytyczny** (oznaczony jako „**critical**") zamyka sekcję krytyczną.
10. **fragment ignorowany** (oznaczony jako "**ignored**") deklaruje wiadomość/ci nieistotne
11. **fragment rozważany-** tylko ważne są wiadomości w tym fragmencie
12. **fragment asercji** (oznaczony "**assert**") eliminuje wszystkie sekwencje wiadomości, które są objęte danym operatorem, jeśli jego wynik jest fałszywy

Pętla Wykonanie w pętli fragmentu diagramu sekwencji



Stan niezmienny lub ciągły (State Invariant /Continuations)

- **Stan niezmienny** jest oznaczany symbolem prostokąta z zaokrąglonymi wierzchołkami.
- **Stany ciągłe** są oznaczone takim samym symbolem, obejmującym kilka linii życia



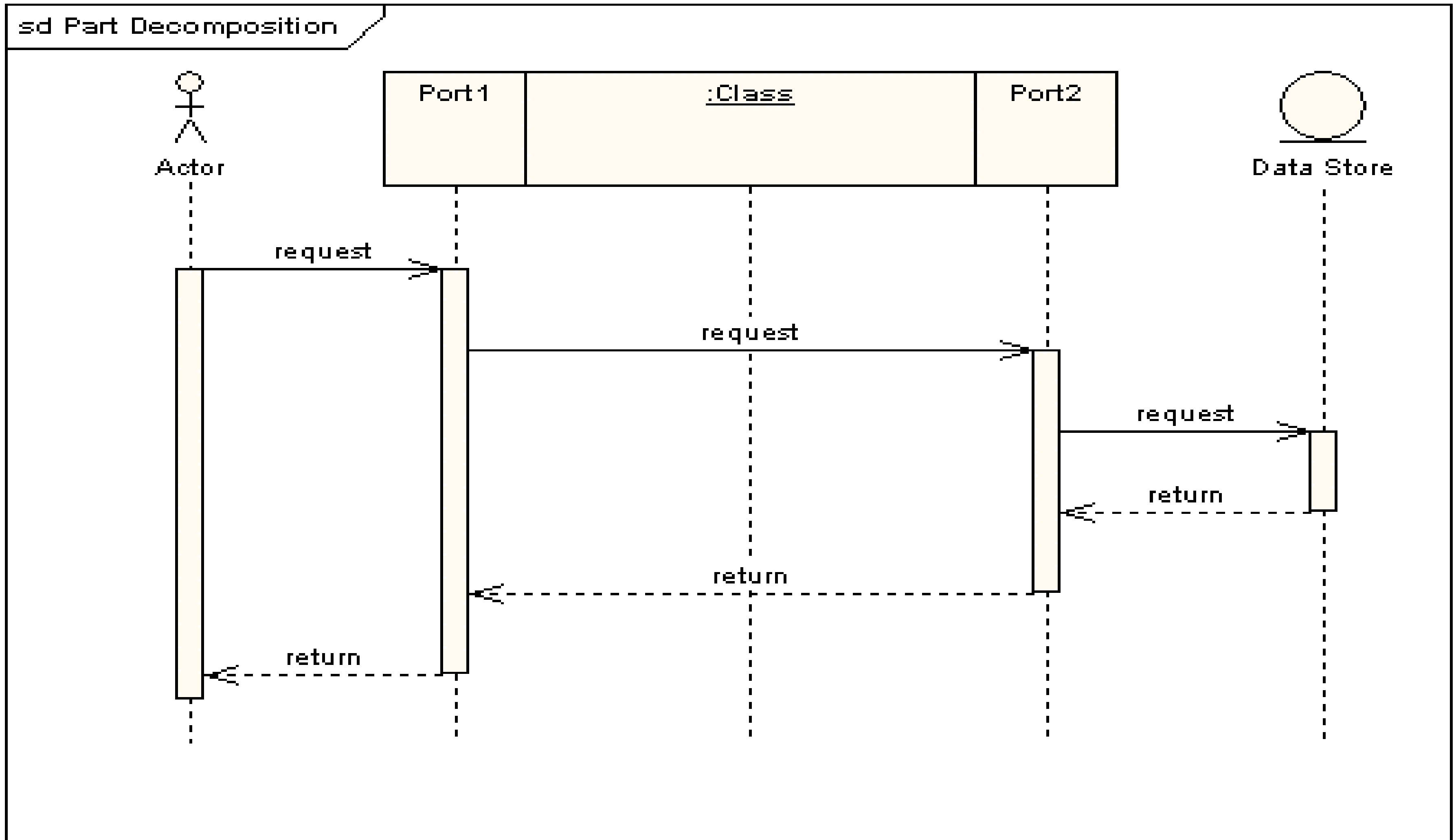
Brama (Gate)

Oznacza przekazywanie wiadomości na zewnątrz między fragmentem i pozostałą częścią diagramu (linie życia, inne fragmenty)

sd Gate
:Source
:Target
critical
Gate
msg
msg

Dekompozycja (Part Decomposition)

Obiekt ma więcej niż jedną linę życia (np. typu Class). Pozwala to pokazać zagnieżdżone protokoły przekazywanych wiadomości np. wewnątrz obiektu i na zewnętrz (w przykładzie typu Class)



Diagramy klas, diagramy sekwencji

1. **Wprowadzenie**
2. **Syntaktyka diagramów klas**

   https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html
3. **Identyfikacja elementów diagramów klas**

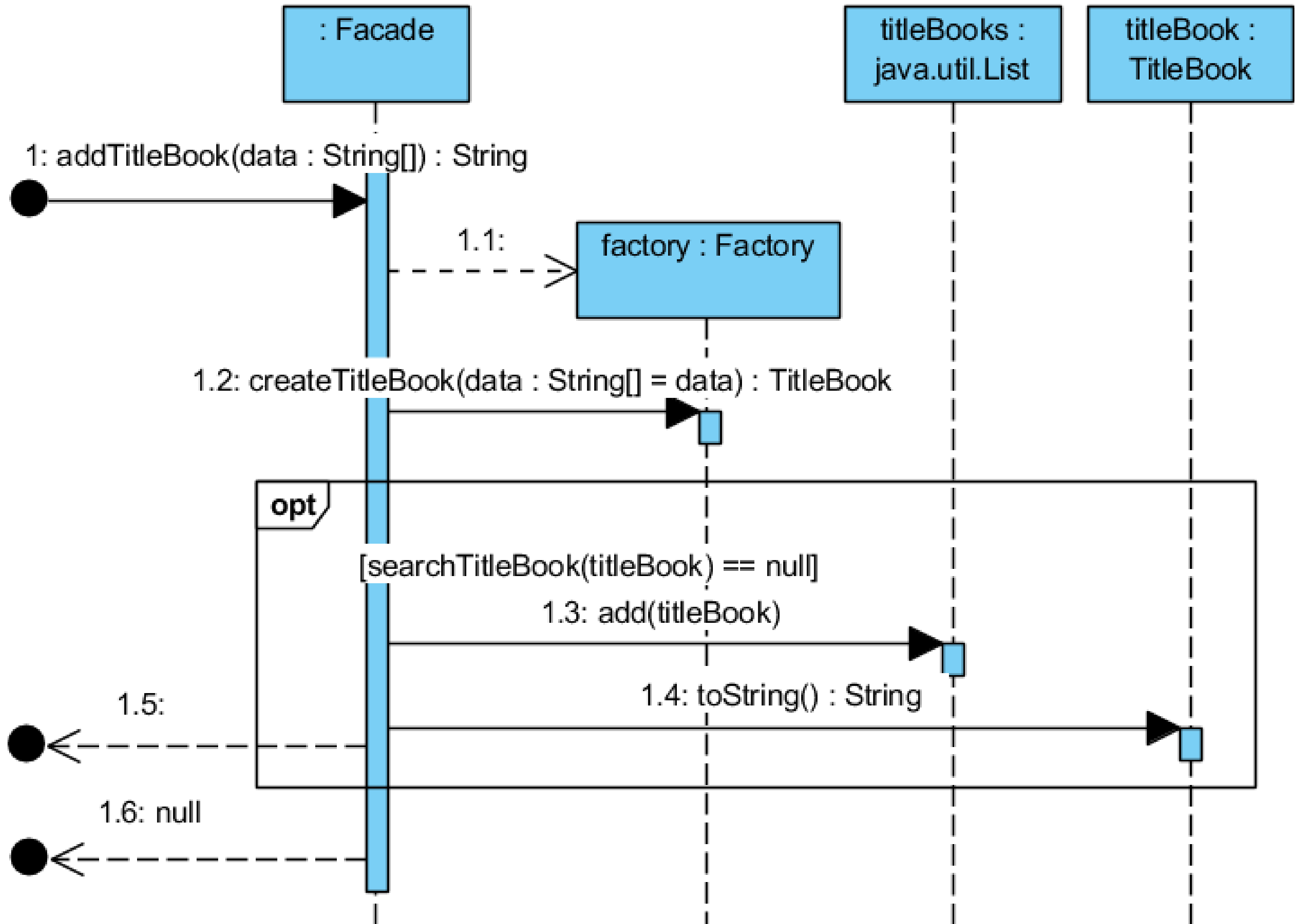
   [Shalloway A.,Trott James R.,Projektowanie zorientowane obiektowo. Wzorce projektowe. Gliwice, Helion, 2005]
4. **Diagramy sekwencji UML**

   https://sparxsystems.com/resources/tutorials/uml2/sequence-diagram.html
5. **Przykłady diagramów sekwencji – kontynuacja przykładu z bieżącego wykładu**

Iteracja 1
Projekt przypadku użycia
„ **Dodaj_Tytul_Ksiazki**”

za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

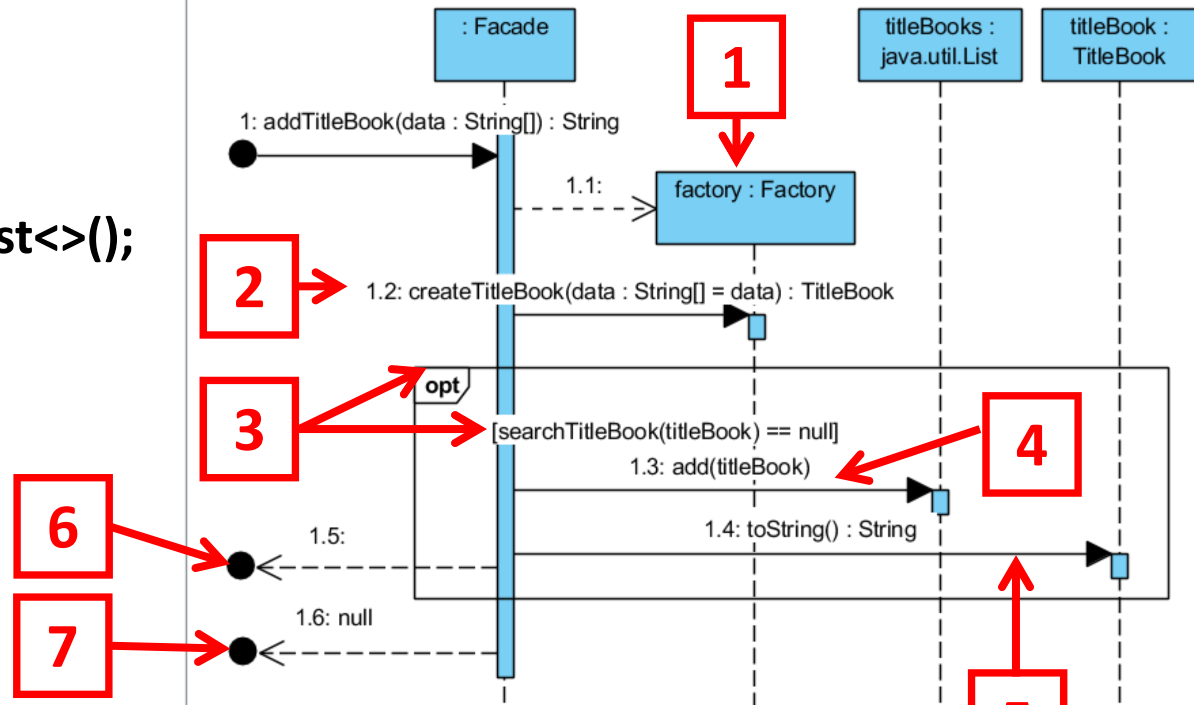
(1) Wstawianie nowego tytułu: public String addTitleBook(String data[])



```java
//class Facade
List<TitleBook> titleBooks;
List<Client> clients;
public Facade() {
    titleBooks = new ArrayList<>();
    clients = new ArrayList();
}

public String addTitleBook(String data[]) {
    Factory factory = new Factory();
    TitleBook titleBook = factory.createTitleBook(data);
    if (searchTitleBook(titleBook) == null) {
        titleBooks.add(titleBook);
        return titleBook.toString();
    }
    return null;
}
```

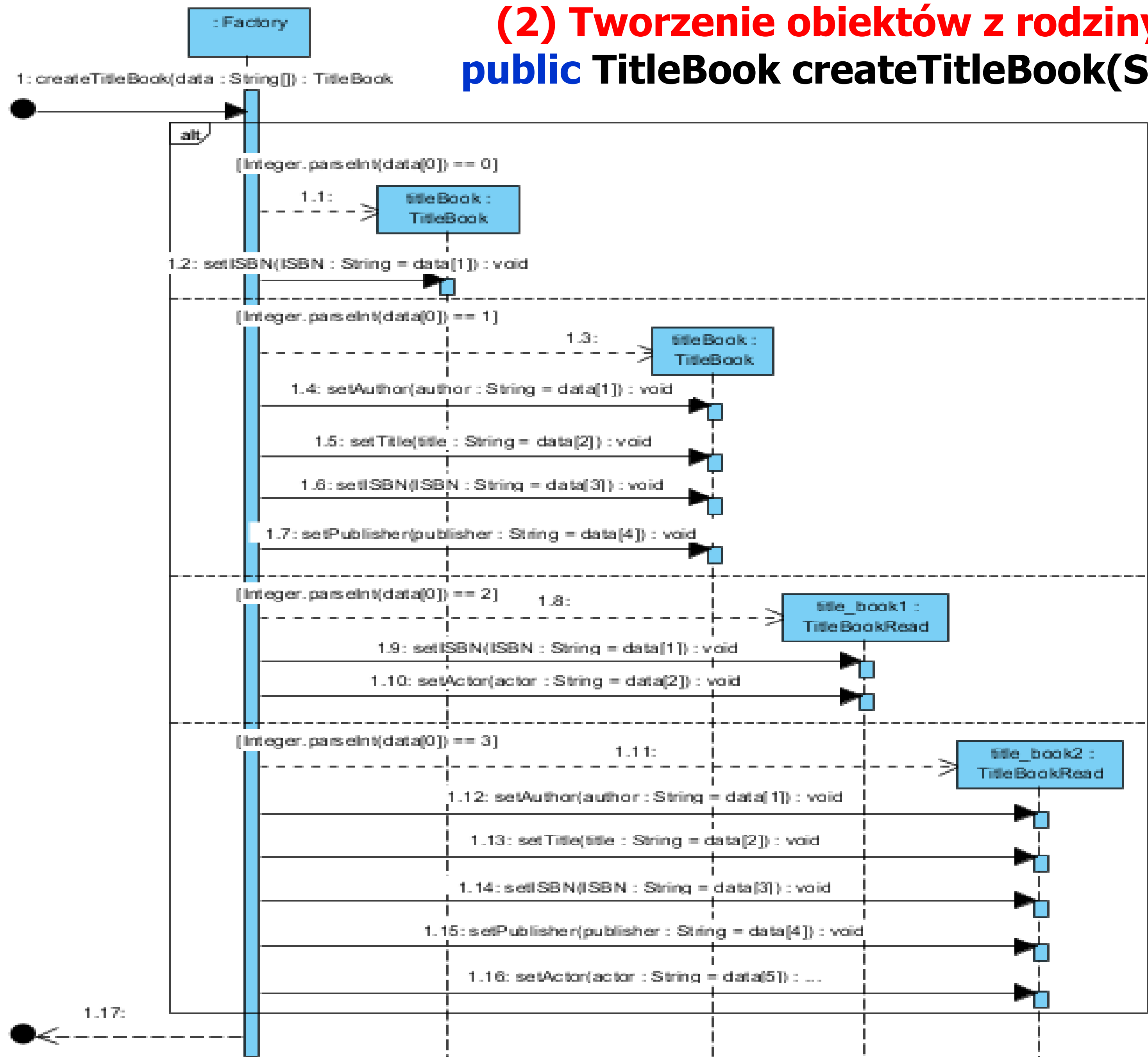
```java
//class Facade
   List<TitleBook> titleBooks;
   List<Client> clients;
   public Facade() {
      titleBooks = new ArrayList<>();
      clients = new ArrayList();
   }
```




```java
  public String addTitleBook(String data[]) {
      Factory factory = new Factory();                       //1
      TitleBook titleBook = factory.createTitleBook(data);   //2
      if (searchTitleBook(titleBook) == null) {              //3
         titleBooks.add(titleBook);                          //4
         return titleBook.toString();                        //5, 6
      }
      return null;                                           //7
  }
```

(2) Tworzenie obiektów z rodziny TitleBook

public TitleBook createTitleBook(String data[])

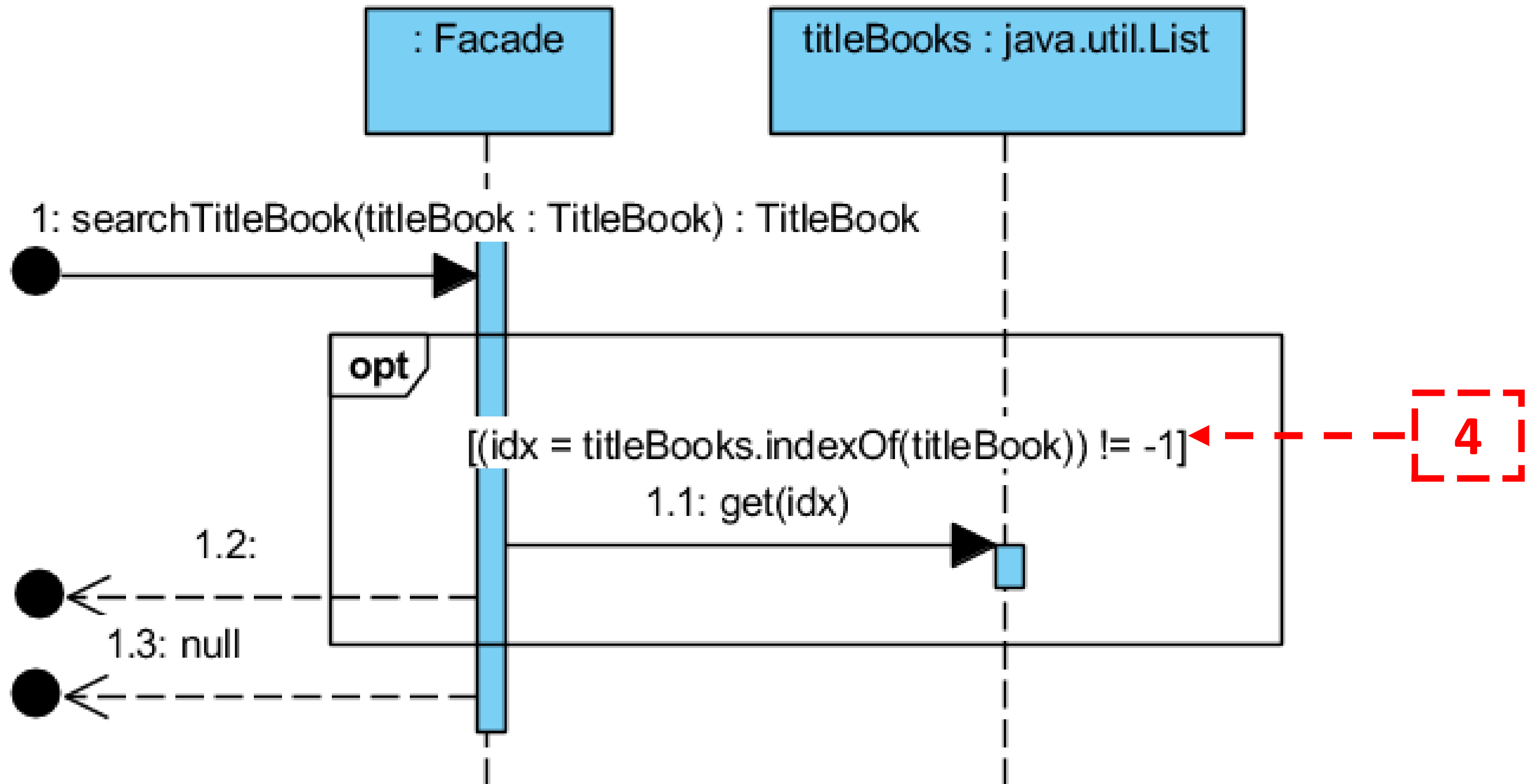


```
public class Factory {      //Factory -decyzje na poziomie tworzenia kodu

    public TitleBook createTitleBook(String data[]) {
     TitleBook titleBook = null;
     switch (Integer.parseInt(data[0]))      //what_title_book_type
     {
        case 0:
           titleBook = new TitleBook();      //TitleBook object for searching
           titleBook.setISBN(data[1]);
           break;
        case 1:
           titleBook = new TitleBook();      //TTitleBook object for persisting
           titleBook.setAuthor(data[1]);
           titleBook.setTitle(data[2]);
           titleBook.setISBN(data[3]);
           titleBook.setPublisher(data[4]);
           break;
```

```
    case 2:
        TitleBookRead title_book1 = new TitleBookRead();
                                        //TitleBookRead object for searching
        title_book1.setISBN(data[1]);
        title_book1.setActor(data[2]);
        titleBook = title_book1;
        break;
     case 3:
        TitleBookRead title_book2 = new TitleBookRead();
                                        //TitleBookRead object for persisting
        title_book2.setAuthor(data[1]);
        title_book2.setTitle(data[2]);
        title_book2.setISBN(data[3]);
        title_book2.setPublisher(data[4]);
        title_book2.setActor(data[5]);
        titleBook = title_book2;
        break;  }
   return titleBook;
}
```

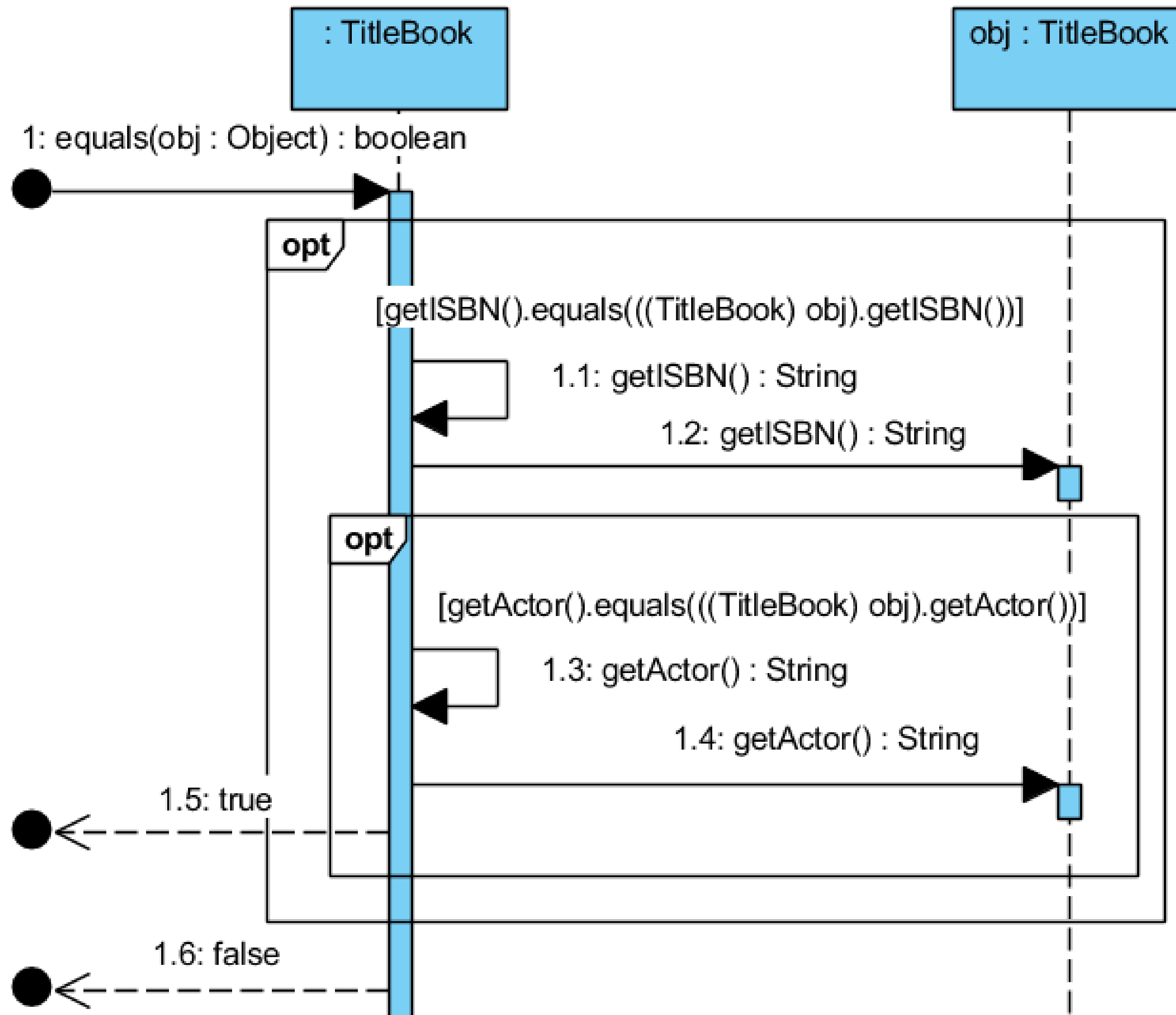
(3) Wyszukiwanie obiektu z rodziny TitleBook



```java
//class Facade
public TitleBook searchTitleBook(TitleBook titleBook) {
    int idx;
    if ((idx = titleBooks.indexOf(titleBook)) != -1) {
      return  titleBooks.get(idx);
    }
    return null;
  }
```

```java
public int indexOf(Object o) {
      if (o == null) {
         for (int i = 0; i < size; i++)
            if (elementData[i]==null)
               return i;
      } else {
         for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
               return i;  }
      return -1;  }
```

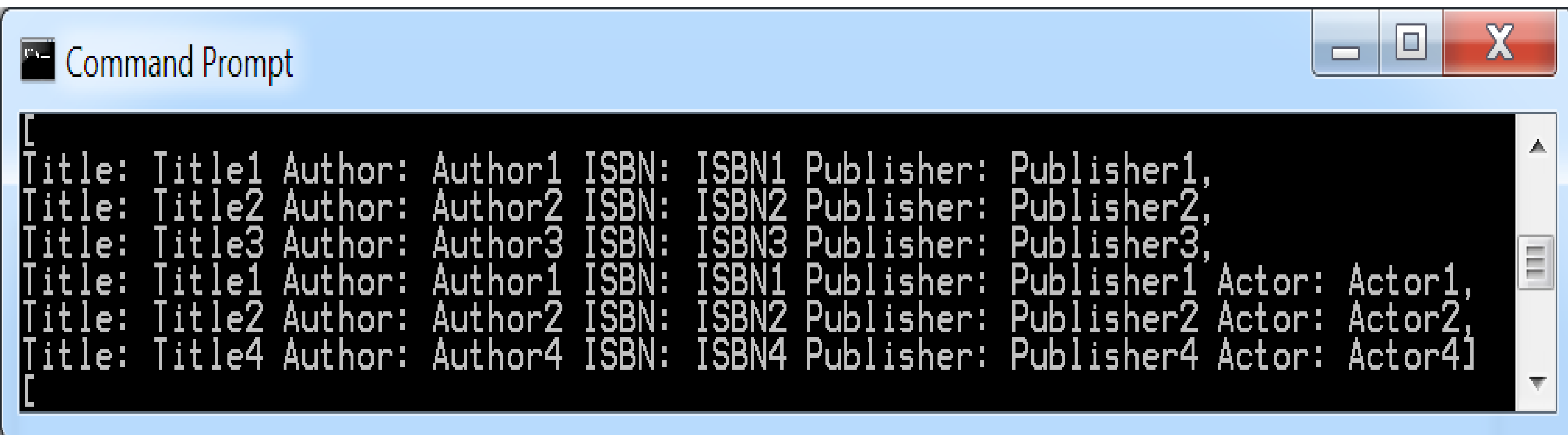
(4) Metoda equals w klasie TitleBook: public boolean equals(Object obj)



//class TitleBook

```
@Override
  public boolean equals(Object obj) {
    boolean result = false;
    if (getISBN().equals(((TitleBook) obj).getISBN())) {
      if (getActor().equals(((TitleBook) obj).getActor())) {
        result = true;
      }
    }
    return result;
  }
```

```java
public static void main(String args[])
{    Facade ap = new Facade();
     String title1[] = {"1", "Author1", "Title1", "ISBN1", "Publisher1"},
            dtitle1[] = {"0", "ISBN1"};
     String title2[] = {"1", "Author2", "Title2", "ISBN2", "Publisher2"},
            dtitle2[] = {"0", "ISBN2"};
     String title3[] = {"1", "Author3", "Title3", "ISBN3", "Publisher3"},
            dtitle3[] = {"0", "ISBN5"};
     String title4[] = {"3", "Author1", "Title1", "ISBN1", "Publisher1", "Actor1"},
            dtitle4[] = {"2", "ISBN1", "Actor1"};
     String title5[] = {"3", "Author2", "Title2", "ISBN2", "Publisher2", "Actor2"};
     String title6[] = {"3", "Author4", "Title4", "ISBN4", "Publisher4", "Actor4"},
             dtitle5[] = {"2", "ISBN4", "Actor4"};
     ap.addTitleBook(title1);       //dodawanie tytułów książek
     ap.addTitleBook(title2);
     ap.addTitleBook(title2);
     ap.addTitleBook(title3);
     ap.addTitleBook(title4);
     ap.addTitleBook(title5);
     ap.addTitleBook(title5);
     ap.addTitleBook(title6);
     String lan = ap.getTitleBooks().toString();
```

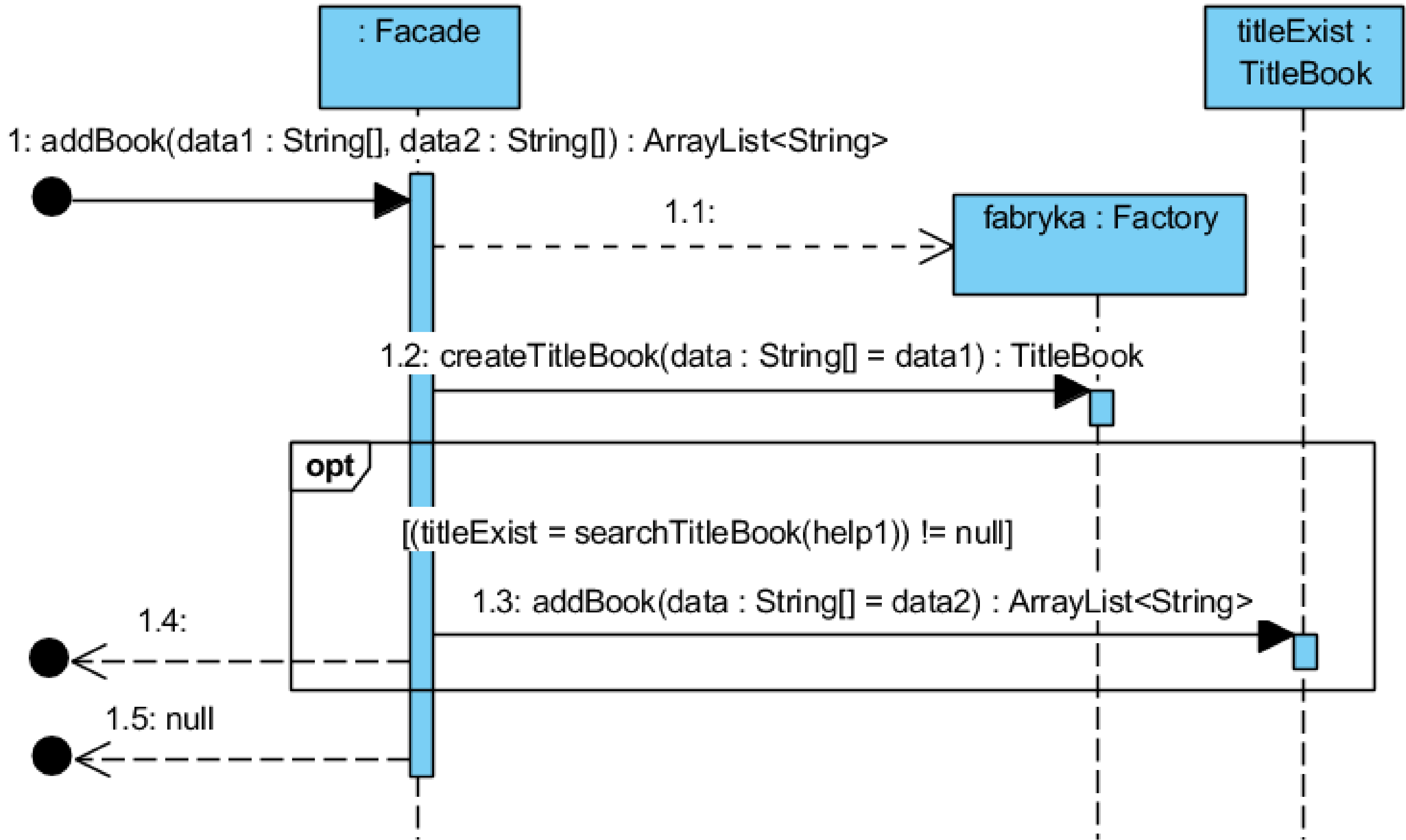

Command Prompt
[
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2,
Title: Title3 Author: Author3 ISBN: ISBN3 Publisher: Publisher3,
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Actor: Actor1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Actor: Actor2,
Title: Title4 Author: Author4 ISBN: ISBN4 Publisher: Publisher4 Actor: Actor4]
[

Dodatek – kolejne iteracje rozwoju tworzenia oprogramowania (kontynuacja przykładu z bieżącego wykładu)

Iteracja 2
Projekt przypadku użycia
„**Dodaj_Ksiazke**”
za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

(5) Wstawianie nowej książki o podanym tytule – 1-y etap

public ArrayList<String> addBook(String data1[], String data2[])

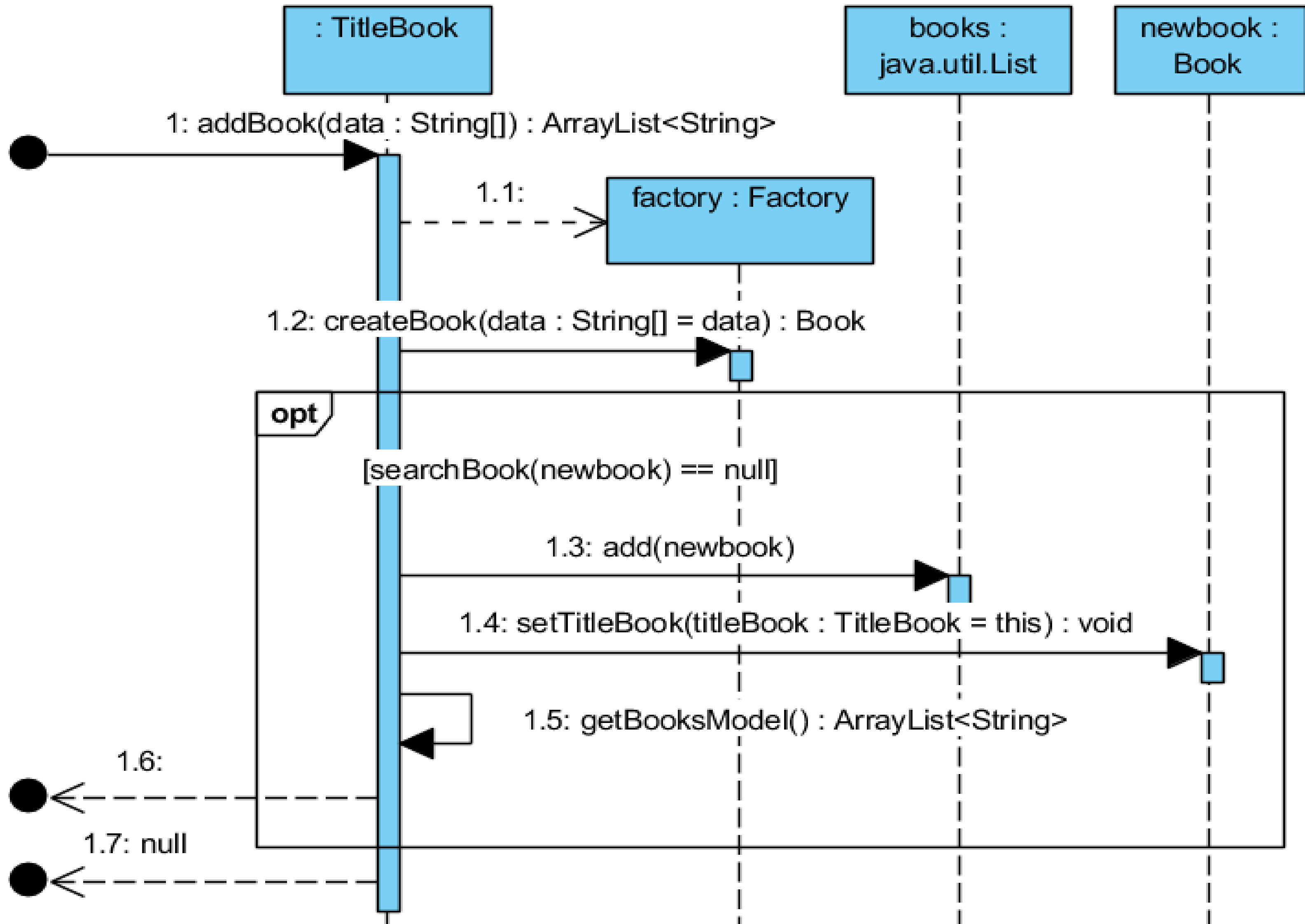


```
                    //class Facade
  List<TitleBook> titleBooks;
  List<Client> clients;
  public Facade() {
     titleBooks = new ArrayList<>();
     clients = new ArrayList();
  }

public ArrayList<String> addBook(String data1[], String data2[]) {
     TitleBook help1, titleExist;
     Factory fabryka = new Factory();
     help1 = fabryka.createTitleBook(data1); //metoda ponownie
                                             //użyta
     if ((titleExist = searchTitleBook(help1)) != null) {  //metoda
         return titleExist.addBook(data2);        //ponownie użyta
       }
     return null;
  }
```

(6) Wstawianie nowej książki o podanym tytule – 2-i etap

public ArrayList<String> addBook(String data[])



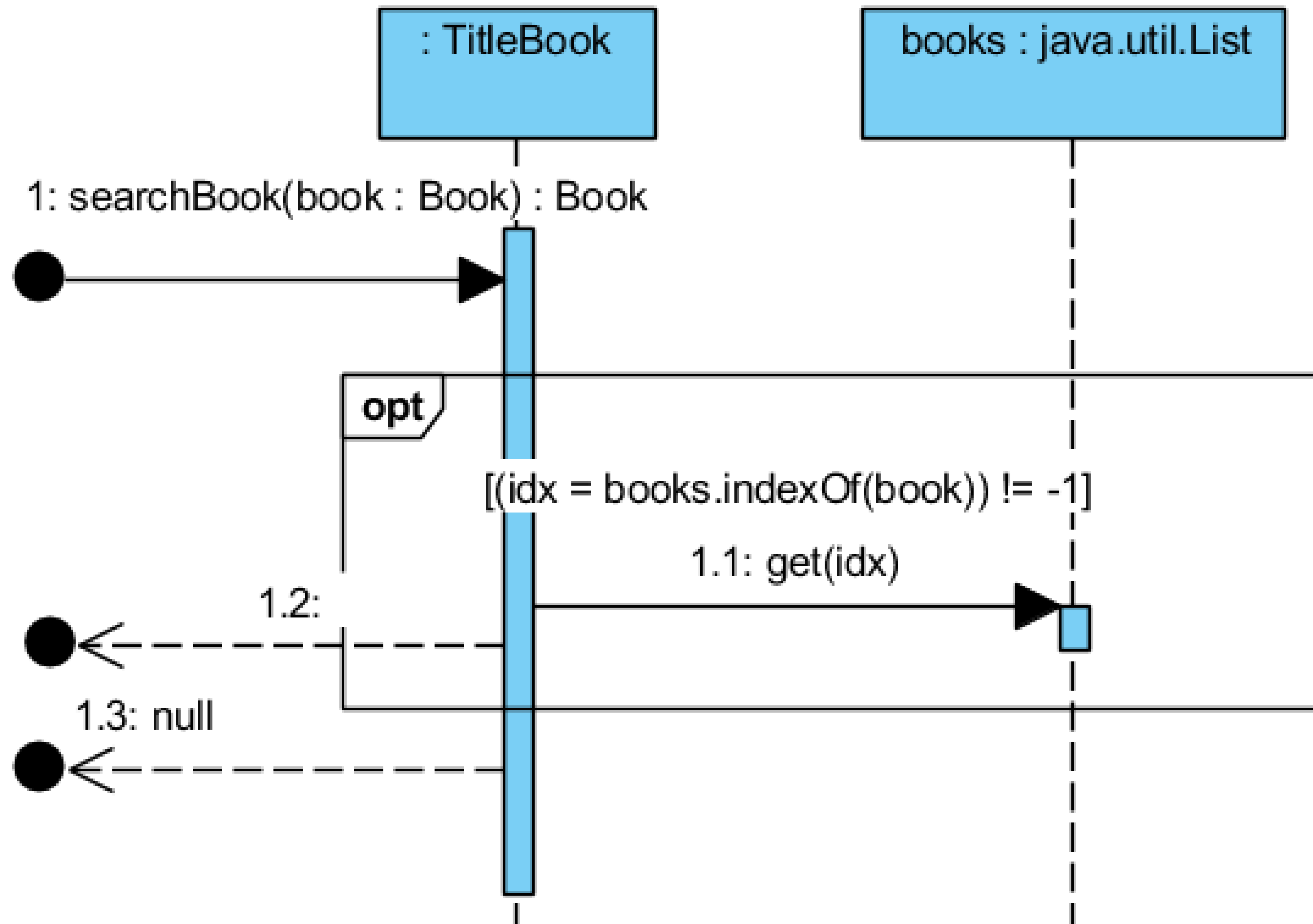
//class TitleBook

```
List<Book> books;
public TitleBook() {
    books = new ArrayList();
}

public ArrayList<String> addBook(String data[]) {
    Factory factory = new Factory();
    Book newbook;
    newbook = factory.createBook(data);
    if (searchBook(newbook) == null) {
        books.add(newbook);
        newbook.setTitleBook(this);
        return getBooksModel();
    }
    return null;
}
```

(7) Szukanie książki

public Book searchBook(Book book)

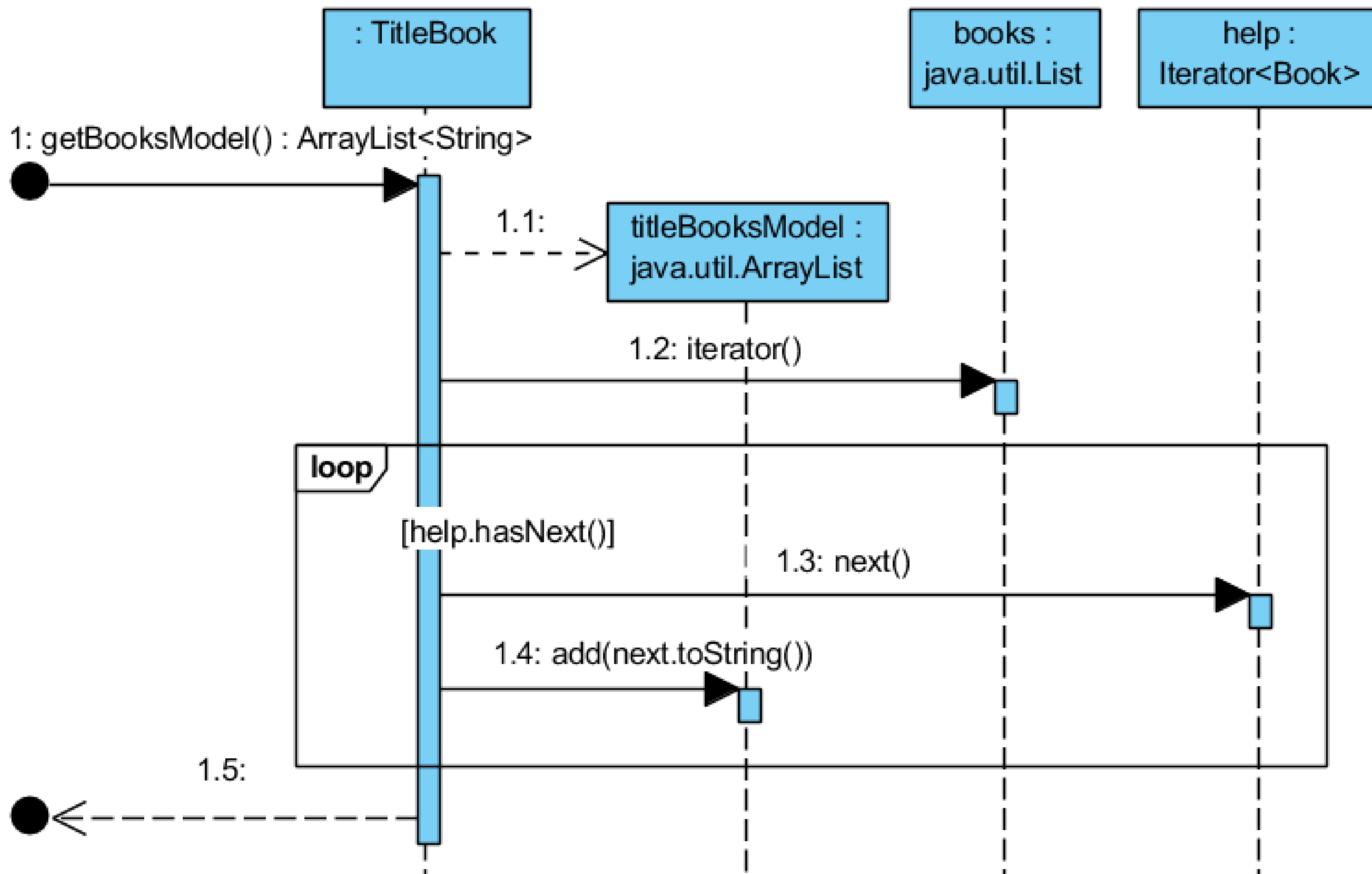


```
                              //class TitleBook
List<Book> books;
  public TitleBook() {
     books = new ArrayList();
  }

public Book searchBook(Book book) {
     int idx;
     if ((idx = books.indexOf(book)) != -1) {
       return (Book) books.get(idx);
     }
     return null;
  }
```

(8)Pobranie danych o książkach należących do obiektu z rodziny TitleBook

public ArrayList<String> getBooksModel()



```java
//class TitleBook
List<Book> books;
  public TitleBook() {
    books = new ArrayList();
  }

public ArrayList<String> getBooksModel() {
    ArrayList<String> titleBooksModel = new ArrayList<>();
    Iterator<Book> help = books.iterator();
    while (help.hasNext()) {
      Book next = help.next();
      titleBooksModel.add(next.toString());
    }
    return titleBooksModel;
  }
```

```
public static void main(String args[])
{    Facade ap = new Facade();
     String title1[] = {"1", "Author1", "Title1", "ISBN1", "Publisher1"},
            dtitle1[] = {"0", "ISBN1"};
     String title2[] = {"1", "Author2", "Title2", "ISBN2", "Publisher2"},
            dtitle2[] = {"0", "ISBN2"};
     String title3[] = {"1", "Author3", "Title3", "ISBN3", "Publisher3"},
            dtitle3[] = {"0", "ISBN5"};
     String title4[] = {"3", "Author1", "Title1", "ISBN1", "Publisher1", "Actor1"},
            dtitle4[] = {"2", "ISBN1", "Actor1"};
     String title5[] = {"3", "Author2", "Title2", "ISBN2", "Publisher2", "Actor2"};
     String title6[] = {"3", "Author4", "Title4", "ISBN4", "Publisher4", "Actor4"},
             dtitle5[] = {"2", "ISBN4", "Actor4"};
     ap.addTitleBook(title1);         //dodawanie tytułów książek
     ap.addTitleBook(title2);
     ap.addTitleBook(title2);
     ap.addTitleBook(title3);
     ap.addTitleBook(title4);
     ap.addTitleBook(title5);
     ap.addTitleBook(title5);
     ap.addTitleBook(title6);
     String lan = ap.getTitleBooks().toString();
```

// cd metody main – dodawanie książek

```
System.out.println(lan);
String book1[] = {"0", "1"};
String book2[] = {"0", "2"};
ArrayList<String> pom = ap.addBook(dtitle1, book1);
if (pom != null)    System.out.print(pom);
pom = ap.addBook(dtitle2, book1);
if (pom != null)    System.out.print(pom);
pom = ap.addBook(dtitle2, book1);
if (pom != null)    System.out.print(pom);
pom = ap.addBook(dtitle2, book2);
if (pom != null)   System.out.print(pom);
pom = ap.addBook(dtitle3, book2);
if (pom != null)   System.out.print(pom);
pom = ap.addBook(dtitle4, book1);
if (pom != null)   System.out.print(pom);
pom = ap.addBook(dtitle5, book2);
if (pom != null)   System.out.print(pom);
```

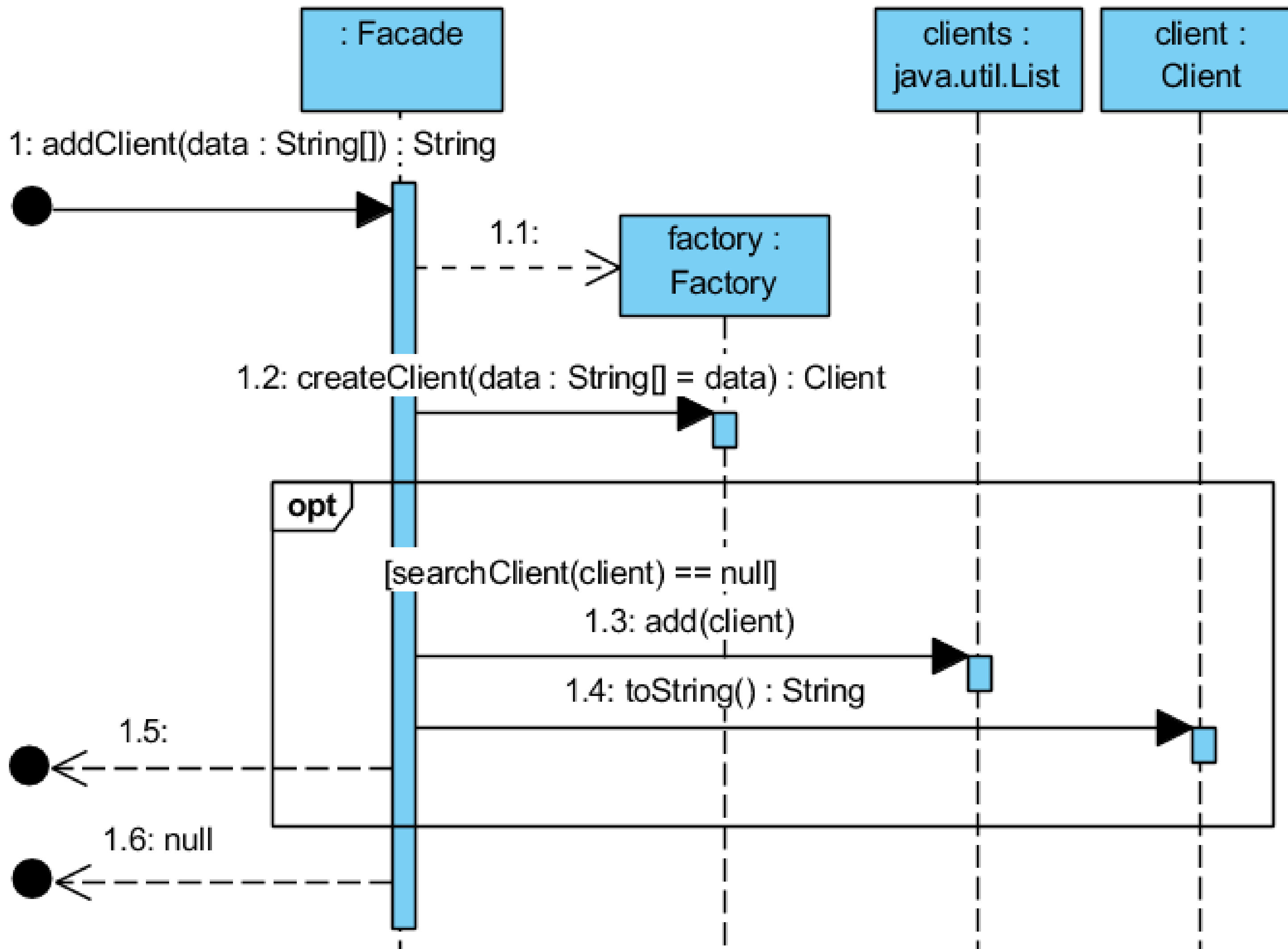
Command Prompt

```
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2,
Title: Title3 Author: Author3 ISBN: ISBN3 Publisher: Publisher3,
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Actor: Actor1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Actor: Actor2,
Title: Title4 Author: Author4 ISBN: ISBN4 Publisher: Publisher4 Actor: Actor4]
[
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Number: 1][
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 1][
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 2][
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Actor: Actor1 Number: 1][
Title: Title4 Author: Author4 ISBN: ISBN4 Publisher: Publisher4 Actor: Actor4 Number: 2]
```

Iteracja 3
Projekt przypadku użycia
„ **Rejestracja_Klienta**”
za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

(9) Wstawianie nowego klienta wypożyczalni

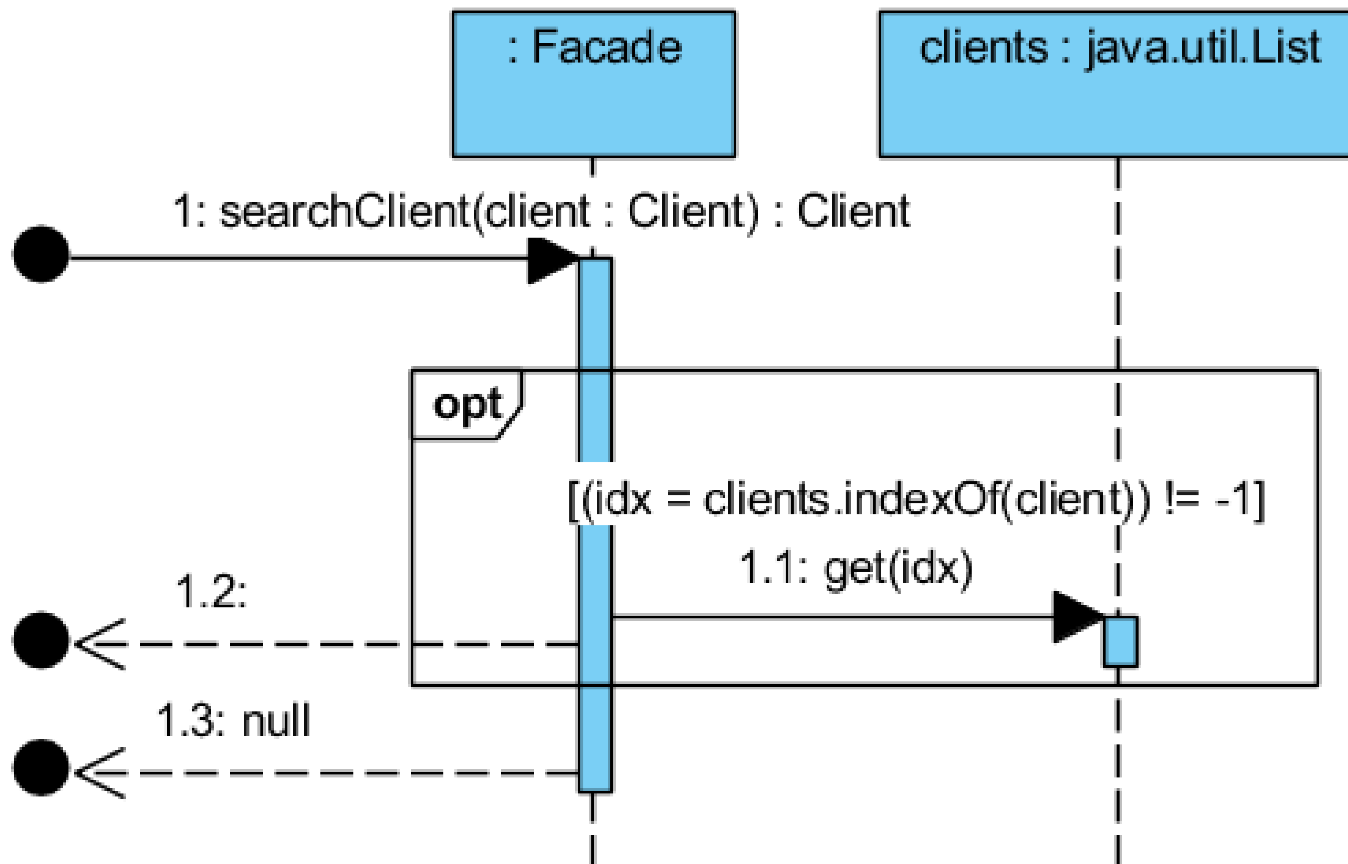
public String addClient(String data[])



```java
//class Facade
List<TitleBook> titleBooks;
List<Client> clients;
public Facade() {
    titleBooks = new ArrayList<>();
    clients = new ArrayList();
}

public String addClient(String data[]) {
    Factory factory = new Factory();
    Client client = factory.createClient(data);
    if (searchClient(client) == null) {
        clients.add(client);
        return client.toString();
    }
    return null;
}
```

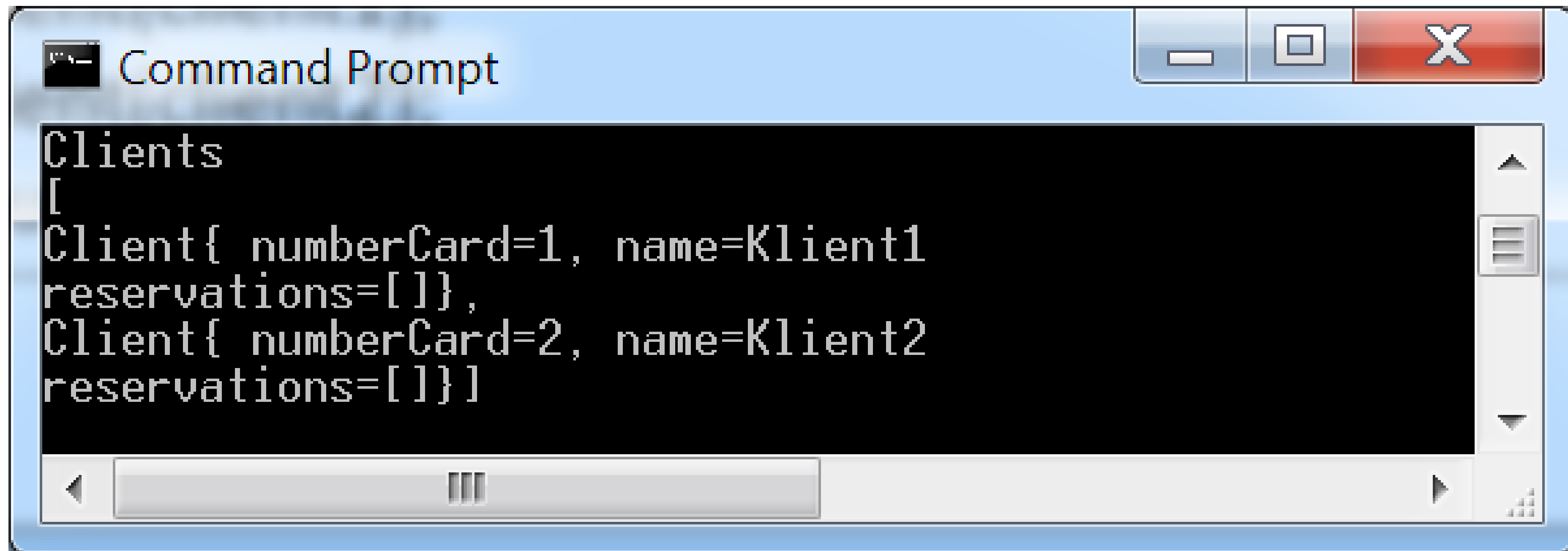
(10) Szukanie klienta
public Client searchClient(Client client)



```
                    //class Facade
  List<TitleBook> titleBooks;
  List<Client> clients;
  public Facade() {
    titleBooks = new ArrayList<>();
    clients = new ArrayList();
  }

public Client searchClient(Client client) {
    int idx;
    if ((idx = clients.indexOf(client)) != -1) {
      return clients.get(idx);
    }
    return null;
  }
```

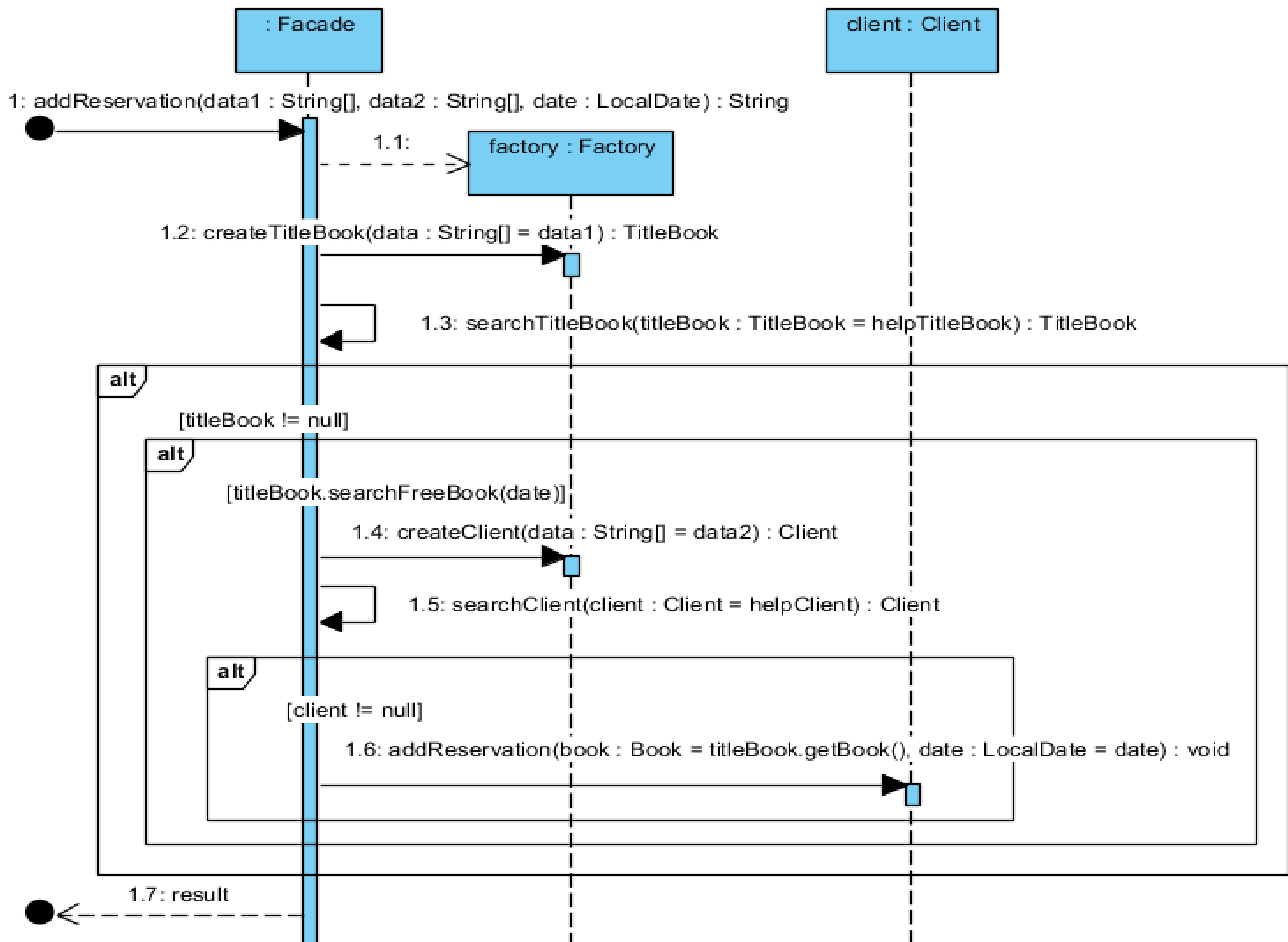
```java
//dodawanie klientów
        System.out.println("\nClients");
String[] client1 = {"1", "Klient1", "1"}, dclient1 = {"0", "1"};
String[] client2 = {"1", "Klient2", "2"}, dclient2 = {"0", "2"}, dclient3 = {"0", "3"};
        ap.addClient(client1);
        ap.addClient(client1);
        ap.addClient(client2);
        System.out.println(ap.clients);
```



Iteracja 4
Projekt przypadku użycia
„**Rezerwacja**”

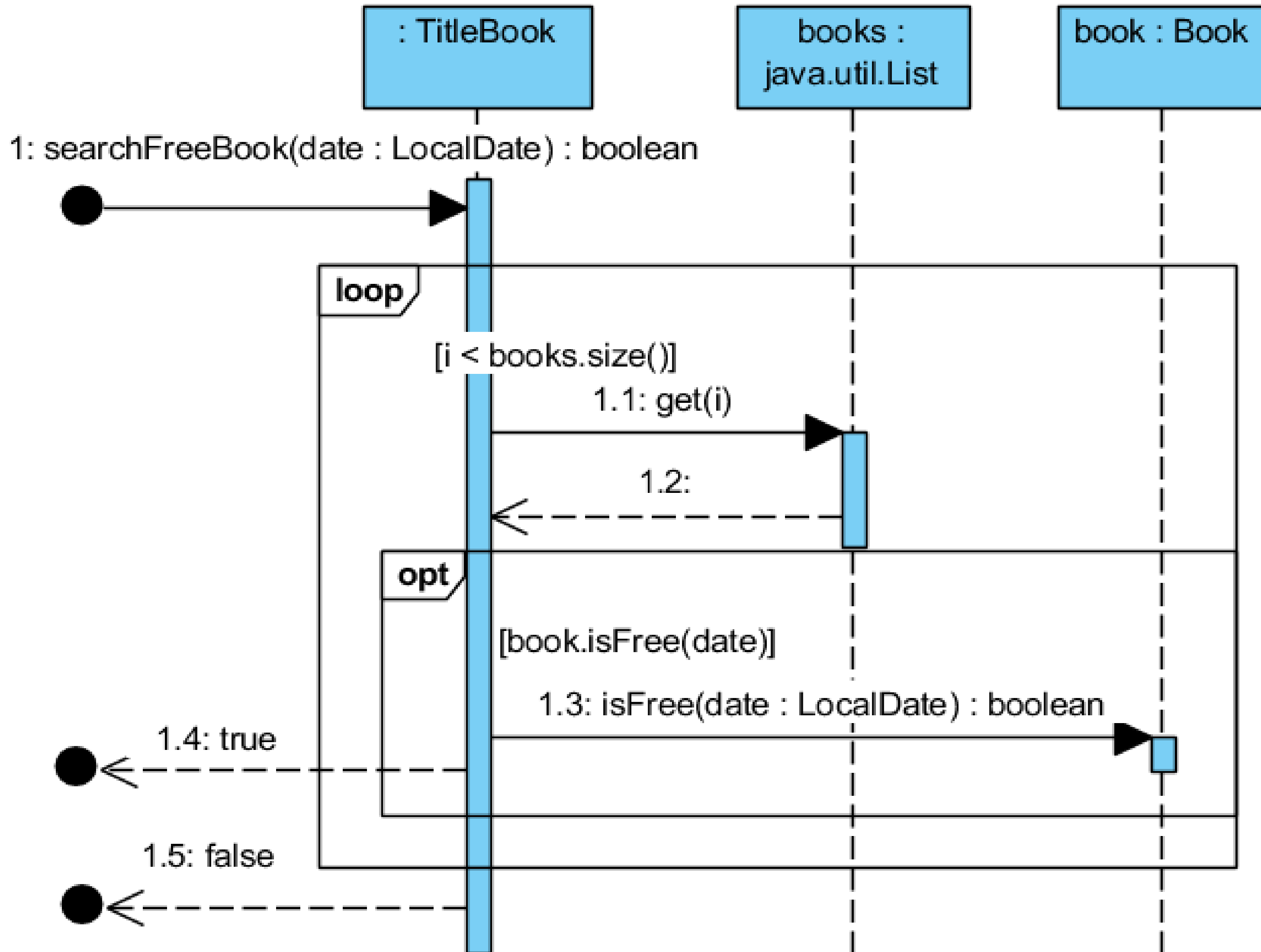
za pomocą diagramu sekwencji i diagramu klas. Diagram klas jest uzupełniany metodami zidentyfikowanymi podczas projektowania scenariusza przypadku użycia za pomocą diagramu sekwencji.

(11) Rezerwacja książki: public String addReservation(String data1[], String data2[], LocalDate date)



```
                         //class Facade
public String addReservation(String data1[], String data2[], LocalDate date) {
    String result;
    Factory factory = new Factory();
    TitleBook helpTitleBook = factory.createTitleBook(data1), titleBook;
                                                        // metoda użyta ponownie
    titleBook = this.searchTitleBook(helpTitleBook);   // metoda użyta ponownie
    if (titleBook != null)
       if (titleBook.searchFreeBook(date)) {                  //nowa metoda
          Client helpClient = factory.createClient(data2), client;  // metoda użyta
                                                              //ponownie
          client = this.searchClient(helpClient);          // metoda użyta ponownie
          if (client != null) {
             client.addReservation(titleBook.getBook(), date);   //nowa metoda
             result = "reserved";                          //wykonana rezerwacja
          } else  result = "no such a client";             //brak klienta
       } else  result = "no free book";                    //brak wolnej ksiazki
     else  result = "no such a title";                     //brak tytułu
    return result;                      //zwrócenie informacji o wyniku rezerwacji
 }
```

(12) Wyszukiwanie wolnej książki do rezerwacji
public boolean searchFreeBook(LocalDate date)

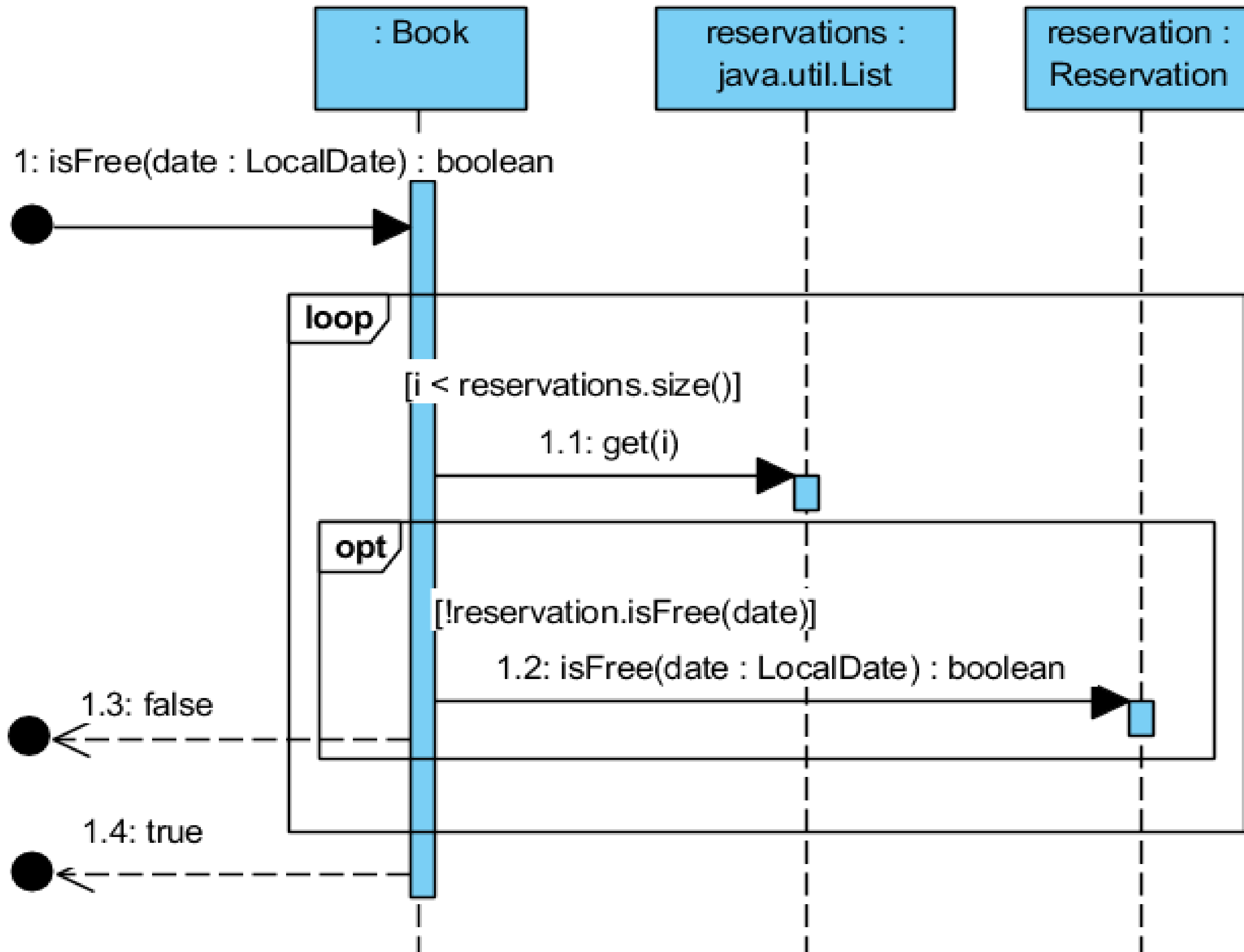


```java
//class TitleBook
List<Book> books;
public TitleBook() {
    books = new ArrayList();
}
private Book book;   //atrybut book przechowuje obiekt typu
                     //Book wyszukany do rezerwacji

public boolean searchFreeBook(LocalDate date) {
    for (int i = 0; i < books.size(); i++) {
        book = books.get(i);
        if (book.isFree(date))
            return true;
    }
    return false;
}
```

(13)Sprawdzanie przez książkę, czy ma wolny termin rezerwacji

public boolean isFree(LocalDate date)

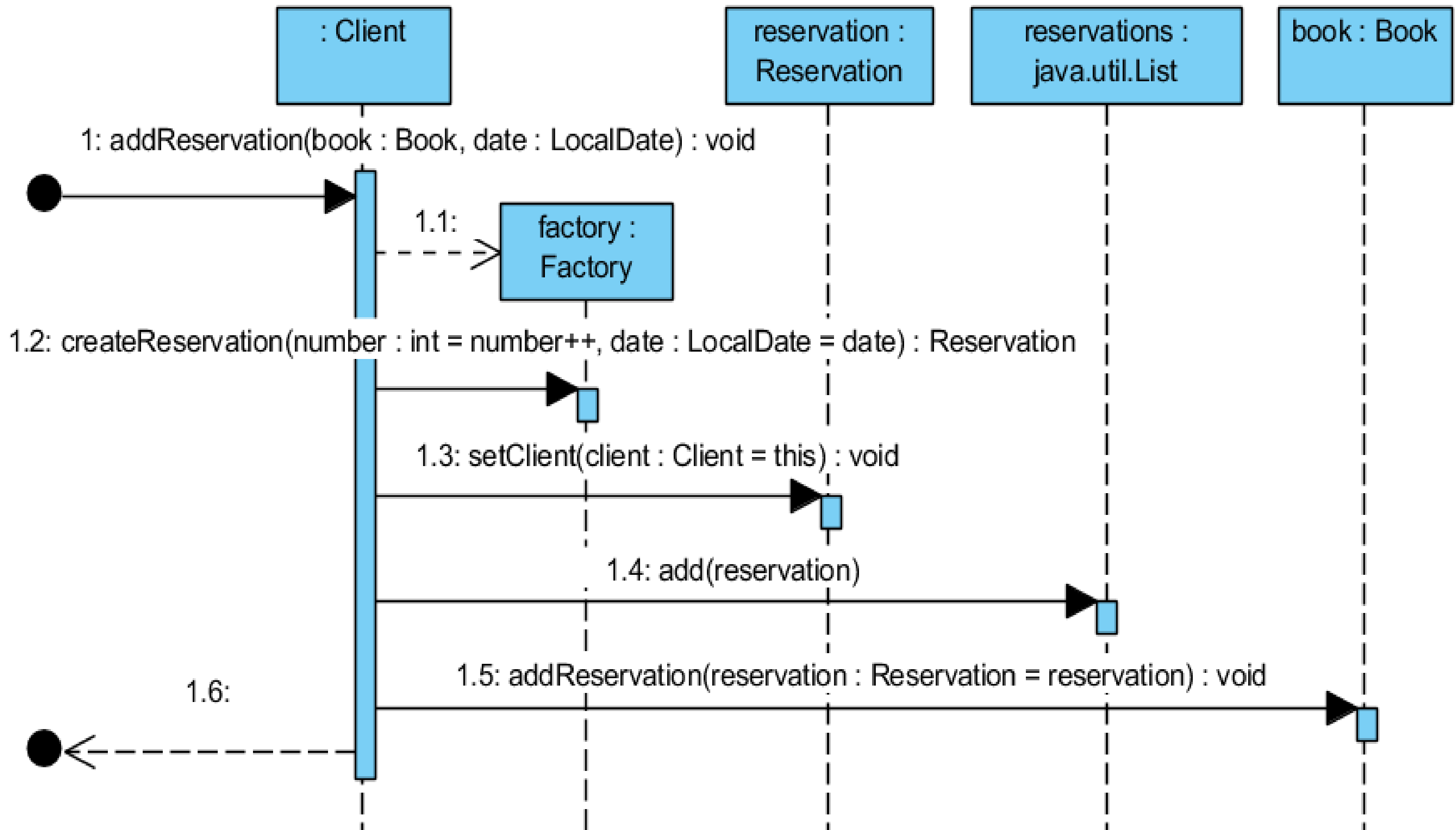


```java
//class Book
 private List<Reservation> reservations;
 public Book() {
      reservations = new ArrayList();
   }

public boolean isFree(LocalDate date) {
      Reservation reservation;
      for (int i = 0; i < reservations.size(); i++) {
         reservation = reservations.get(i);
         if (!reservation.isFree(date)) {
            return false;
         }
      }
      return true;
   }
```

(14) Wykonanie rezerwacji przez obiekt typu Client – 1-y etap

public void addReservation(Book book, LocalDate date)

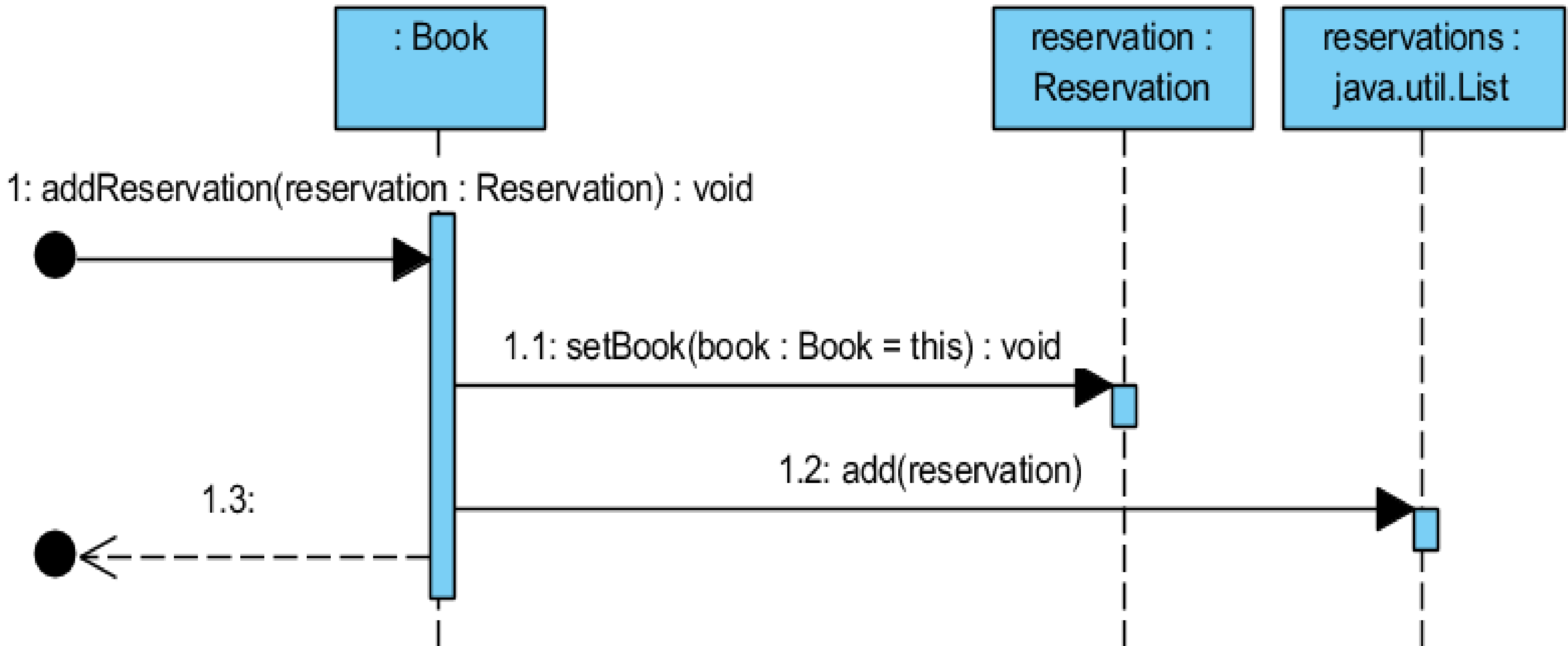


```
                    //class Client
 private List<Reservation> reservations;
 public Client() {
      reservations=new ArrayList();
   }

public void addReservation(Book book, LocalDate date)
  {
     Factory factory=new Factory();
     Reservation reservation=
                   factory.createReservation(number++, date);
     reservation.setClient(this);
     reservations.add(reservation);
     book.addReservation(reservation);
  }
```

(15)Wykonanie rezerwacji przez obiekt typu Book – 2-i etap

public void addReservation(Reservation reservation)



```
// class Book

private List<Reservation> reservations;
 public Book() {
      reservations = new ArrayList();
   }

 public void addReservation(Reservation reservation) {
      reservation.setBook(this);
      reservations.add(reservation);
   }
```

```java
public static void main(String args[])
{     Facade ap = new Facade();
      String title1[] = {"1", "Author1", "Title1", "ISBN1", "Publisher1"},
             dtitle1[] = {"0", "ISBN1"};
      String title2[] = {"1", "Author2", "Title2", "ISBN2", "Publisher2"},
             dtitle2[] = {"0", "ISBN2"};
      String title3[] = {"1", "Author3", "Title3", "ISBN3", "Publisher3"},
             dtitle3[] = {"0", "ISBN5"};
      String title4[] = {"3", "Author1", "Title1", "ISBN1", "Publisher1", "Actor1"},
             dtitle4[] = {"2", "ISBN1", "Actor1"};
      String title5[] = {"3", "Author2", "Title2", "ISBN2", "Publisher2", "Actor2"};
      String title6[] = {"3", "Author4", "Title4", "ISBN4", "Publisher4", "Actor4"},
              dtitle5[] = {"2", "ISBN4", "Actor4"};
      ap.addTitleBook(title1); //dodawanie tytułów książek
      ap.addTitleBook(title2);
      ap.addTitleBook(title2);
      ap.addTitleBook(title3);
      ap.addTitleBook(title4);
      ap.addTitleBook(title5);
      ap.addTitleBook(title5);
      ap.addTitleBook(title6);
      String lan = ap.getTitleBooks().toString();
```

```java
//dodawanie książek
    System.out.println(lan);
    String book1[] = {"0", "1"};
    String book2[] = {"0", "2"};
    ArrayList<String> pom = ap.addBook(dtitle1, book1);
    if (pom != null)    System.out.print(pom);
    pom = ap.addBook(dtitle2, book1);
    if (pom != null)    System.out.print(pom);
    pom = ap.addBook(dtitle2, book1);
    if (pom != null)    System.out.print(pom);
    pom = ap.addBook(dtitle2, book2);
    if (pom != null)   System.out.print(pom);
    pom = ap.addBook(dtitle3, book2);
    if (pom != null)   System.out.print(pom);
    pom = ap.addBook(dtitle4, book1);
    if (pom != null)   System.out.print(pom);
    pom = ap.addBook(dtitle5, book2);
    if (pom != null)   System.out.print(pom);
```

Command Prompt

```
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2,
Title: Title3 Author: Author3 ISBN: ISBN3 Publisher: Publisher3,
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Actor: Actor1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Actor: Actor2,
Title: Title4 Author: Author4 ISBN: ISBN4 Publisher: Publisher4 Actor: Actor4]
[
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Number: 1][
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 1][
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 1,
Title: Title2 Author: Author2 ISBN: ISBN2 Publisher: Publisher2 Number: 2][
Title: Title1 Author: Author1 ISBN: ISBN1 Publisher: Publisher1 Actor: Actor1 Number: 1][
Title: Title4 Author: Author4 ISBN: ISBN4 Publisher: Publisher4 Actor: Actor4 Number: 2]
```

```
  //dodawanie klientów
      System.out.println("\nClients");
      String[] client1 = {"1", "Klient1", "1"}, dclient1 = {"0", "1"};
      String[] client2 = {"1", "Klient2", "2"}, dclient2 = {"0", "2"}, dclient3 = {"0", "3"};
      ap.addClient(client1);
      ap.addClient(client1);
      ap.addClient(client2);
      System.out.println(ap.clients);
 //dodawanie rezerwacji
      System.out.println("\nReservations");
      LocalDate[] dates = {LocalDate.of(2018, Month.JANUARY, 20),
                           LocalDate.of(2018, Month.MARCH, 20)};
      System.out.println(ap.addReservation(dtitle1, dclient1, dates[0]));
      System.out.println(ap.addReservation(dtitle1, dclient2, dates[0]));
      System.out.println(ap.addReservation(dtitle2, dclient2, dates[0]));
      System.out.println(ap.addReservation(dtitle2, dclient3, dates[1]));
      System.out.println(ap.addReservation(dtitle2, dclient1, dates[0]));
      System.out.println(ap.addReservation(dtitle2, dclient3, dates[0]));
       System.out.println("\nClients");
       System.out.println(ap.clients);              //rezerwacje klientów
   }
}
```

Command Prompt

```
Clients
[
Client{ numberCard=1, name=Klient1
reservations=[]},
Client{ numberCard=2, name=Klient2
reservations=[]}]

Reservations
reserved
no free book
reserved
no such a client
reserved
no free book

Clients
[
Client{ numberCard=1, name=Klient1
reservations=[Reservation{ number=0, date=2018-01-20}, Reservation{ number=2, date=2018-01-20}]},
Client{ numberCard=2, name=Klient2
reservations=[Reservation{ number=1, date=2018-01-20}]}]
```

Rezultat – diagram klas uzyskany w procesie projektowania