Programowanie obiektowe

Wykład 12

Marcin Młotkowski

6 czerwca 2019

Plan wykładu

Plan wykładu

Kiedy dziedziczyć

Wskazówka

Podklasa przedefiniowuje operację nadklasy

lub

dodaje nową funkcjonalność

Zły przykład

```
class ListaJednokier
{
    Object obj;
    ListaJednokier nast;
    void dodaj(Object obj);
}

class ListaDwukier : ListaJednokier {
    ListaDwukier poprz;
    ListaDwukier nast;
    void dodaj (Object obj); // na początek
    void naKoniec (Object obj); // na koniec
}
```

Analiza przykładu

Klasa ListaDwukier ma zupełnie inną implementację niż klasa ListaJednokier, nie korzysta ani z odziedziczonych pól, ani z odziedziczonych metod.

Klasy ListaDwukier i ListaJednokier mają podobne interfejsy.

Morał

Identyczny interfejs nie musi implikować dziedziczenia.

Przesłanką do dziedziczenia jest wykorzystanie implementacji z nadklasy (dziedziczenie implementacji) i rozszerzenie funkcjonalności

Co z tym zrobić

Wspólna klasa abstrakcyjna

```
abstract class ListaAbstrakcyjna
{
    void public dodaj(Object obj);
}
```

Wspólny interfejs

```
interface ILista
{
    void dodaj(Object obj);
}
```

Przypomnienie

Klasy powinne mieć precyzyjnie określone zadanie.

W przypadku "szerokiej" funkcjonalności klasy lepiej poskładać ją z mniejszych.

Przykład

Wersja prosta

```
class osoba {
    string Imie, Nazwisko;
    public void edycja() { ... }
    public void odczyt() { ... }
    public void zapis() { ... }
}
```

Bardziej uniwersalna

```
class osoba {
    string Imie, Nazwisko;
    Edytor e = new EdytorQt();
    BazaDanych bd = new BSDQLite();
}
```

Wzorce projektowe

Poznane wzorce

- Singleton
- MVC
- Szablon i Strategia

Wzorce projektowe

Poznane wzorce

- Singleton
- MVC
- Szablon i Strategia

Źródło wzorców

Wzorce projektowe, E. Gamma, R. Helm, R. Johnson, J. Vlissides

Plan wykładu

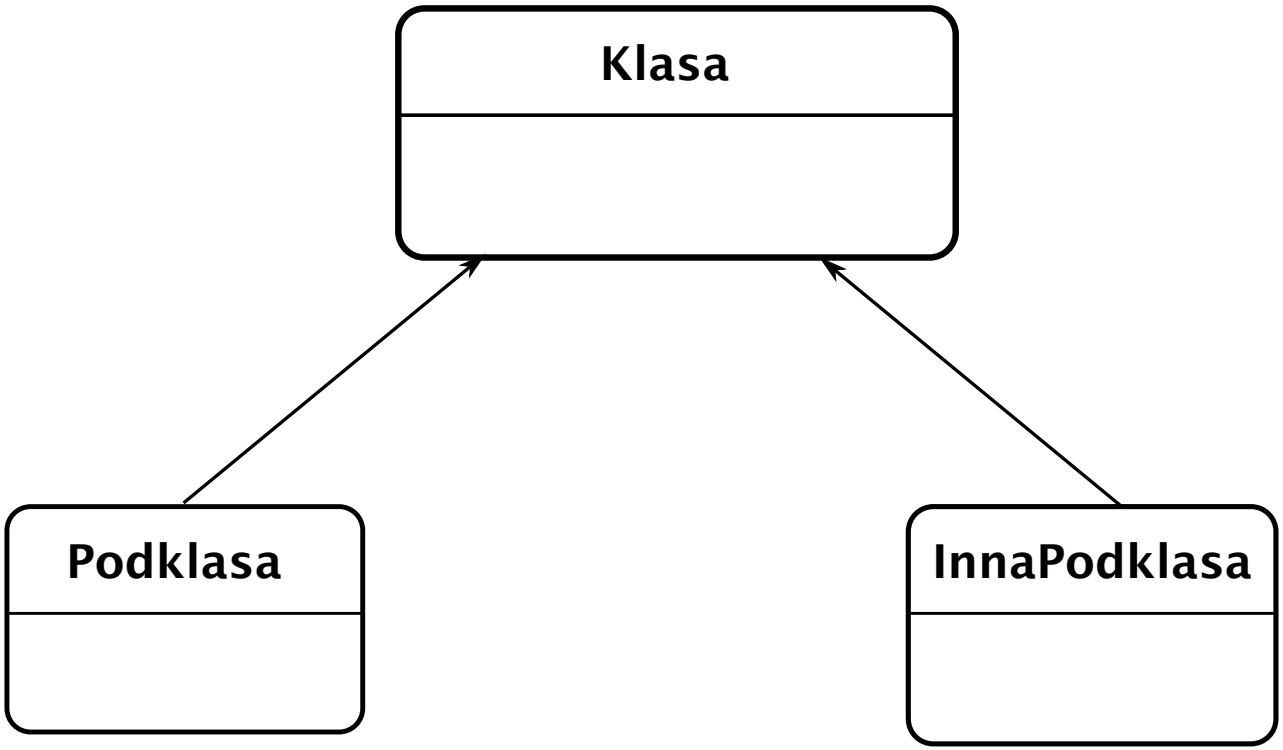
* obiektowe

- Analiza obiektowa
- Projektowanie obiektowe
- Programowanie obiektowe

Programowanie obiektowe

- Implementacja klas wskazanych w analizie
- Implementacja związków
- Uszczegółowienie, tj. dodanie klas

Analiza obiektowa

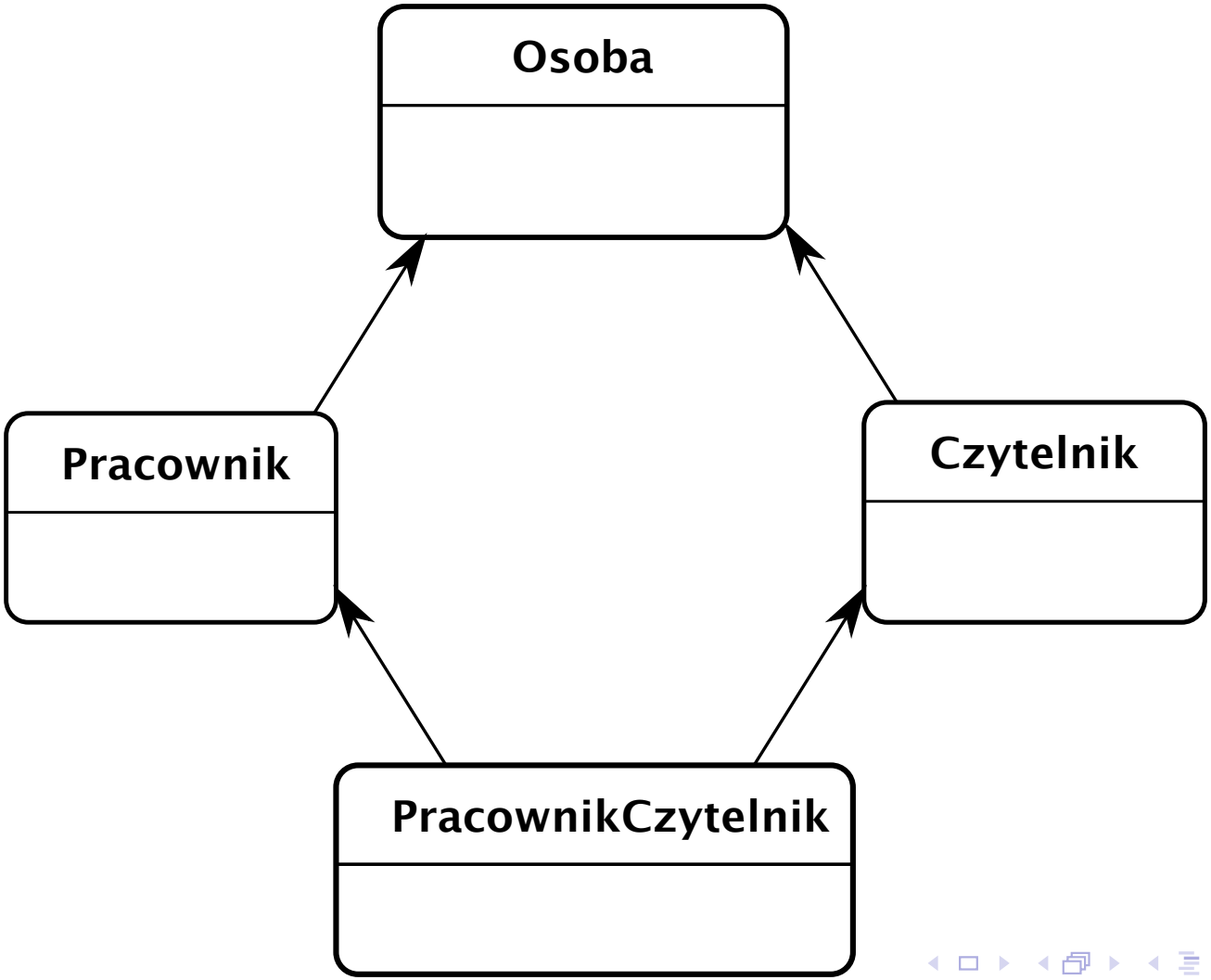


Implementacja schematu

Dziedziczenie

```
abstract class Klasa {
    ...
}
class Podklasa : Klasa {
    ...
}
class InnaPodklasa : Klasa {
    ...
}
```

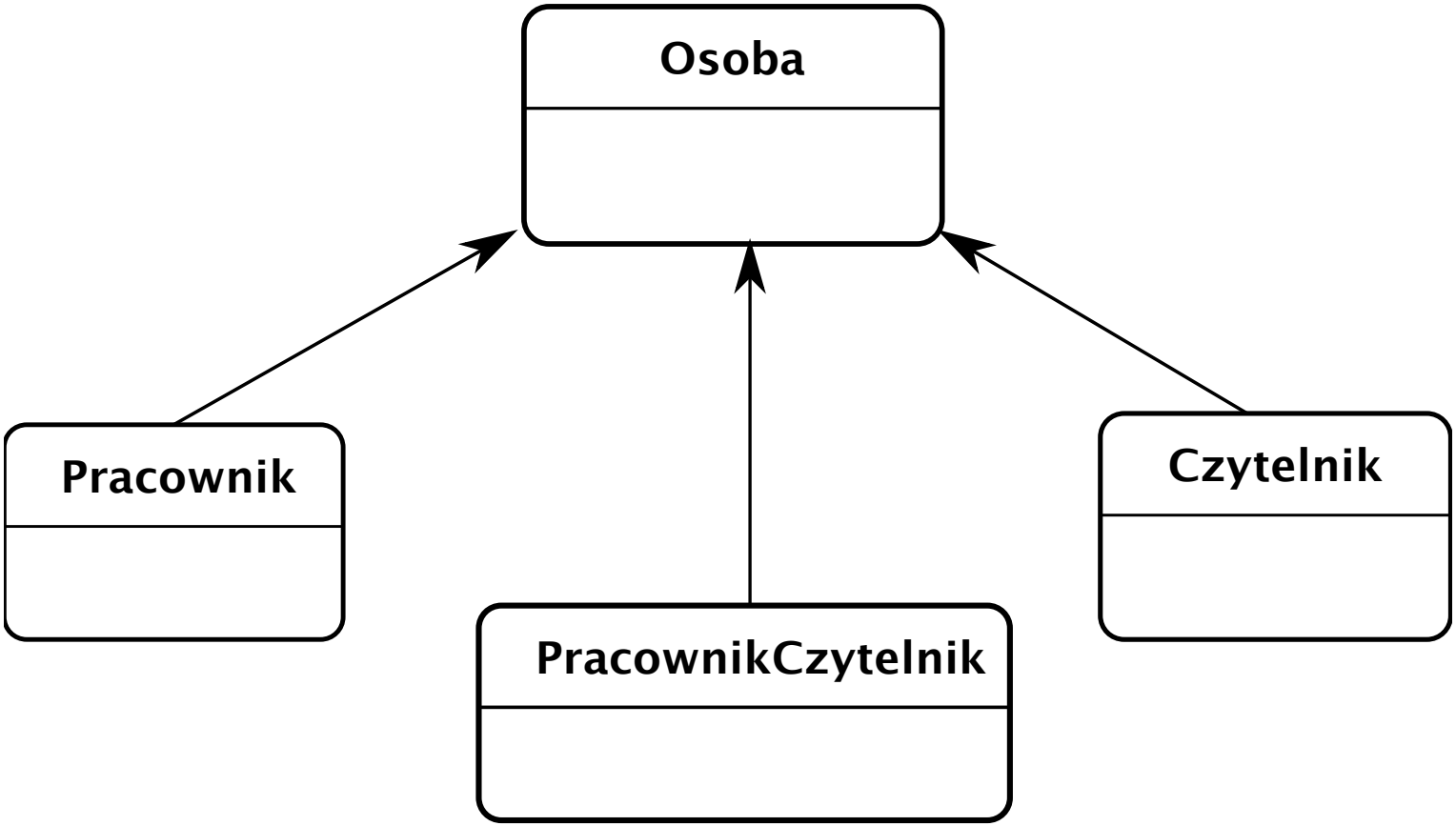
Bardziej skomplikowane zadanie



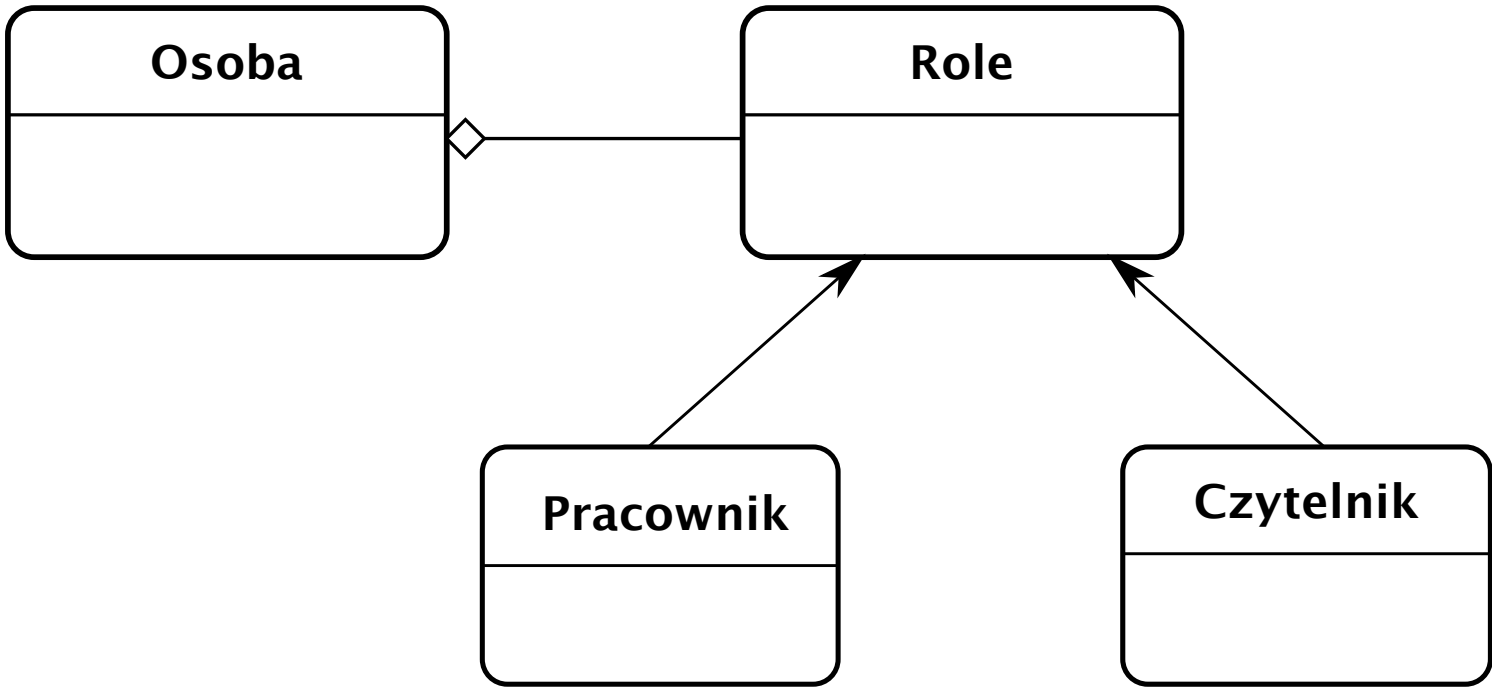
Implementacja

Implementować w języku posiadającym wielodziedziczenie: Python, C++.

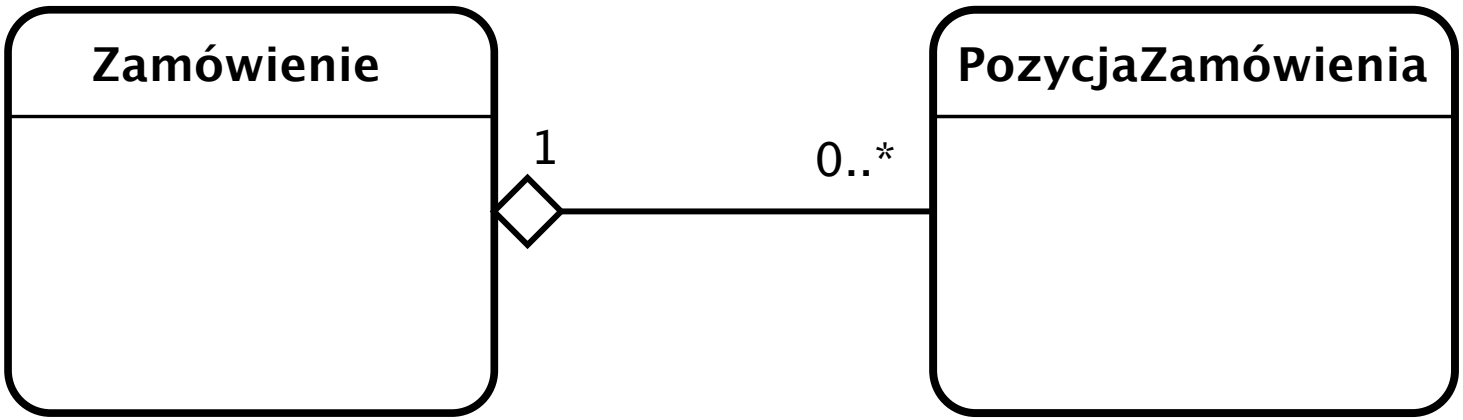
Spłaszczenie hierarchii



Podział na role



Agregacja — przypomnienie



Implementacja agregacji

Kolekcje

- Kolekcje pojawiają się jako dodatkowe klasy, nieuwzględniony w ogólnym projekcie
- Kolekcje są obecne w większości (wszystkich?) liczących się środowiskach programistycznych

Samodzielna implementacja kolekcji, 1. podejście

```
class Osoba {
    String Nazwisko;
    Osoba następnik;
    void dołącz(Osoba o) { ... }
}
```

Samodzielna implementacja kolekcji, 1. podejście

```
class Osoba {
    String Nazwisko;
    Osoba następnik;
    void dołącz(Osoba o) { ... }
}
```

Ocena implementacji

- Klasa łączy funkcje listy i Osoby
- Wymaga implementacji mechanizmów listowych dla każdej klasy osobno
- Kłopot z listą pustą

Implementacja kolekcji, 2. podejście

```
class Lista {
    Osoba val;
    Lista następnik;
    void dołącz(Osoba o){ ... }
}
```

Implementacja kolekcji, 2. podejście

```
class Lista {
    Osoba val;
    Lista następnik;
    void dołącz(Osoba o){ ... }
}
```

Ocena implementacji

- Klasa Osoba jest czystą klasą
- Kłopot listą pustą

Implementacja kolekcji, 3. podejście

```
class Lista {
    ElemListy lista;
    bool empty();
    void dołącz(Osoba o);
}

class ElemListy {
    Osoba val;
    ElemListy następnik;
}
```

Ocena implementacji

Zalety

- Klasy mają dokładnie określone zadania
- Klasę Lista można wykorzystywać do przechowywania obiektów innych klas

Wady

Rośnie liczba klas i zależności między nimi.

Implementacja związków między obiektami

- Poprzez referencje
- Utworzenie nowej klasy reprezentującej związek

Przykład

Małżeństwo

Prosty system

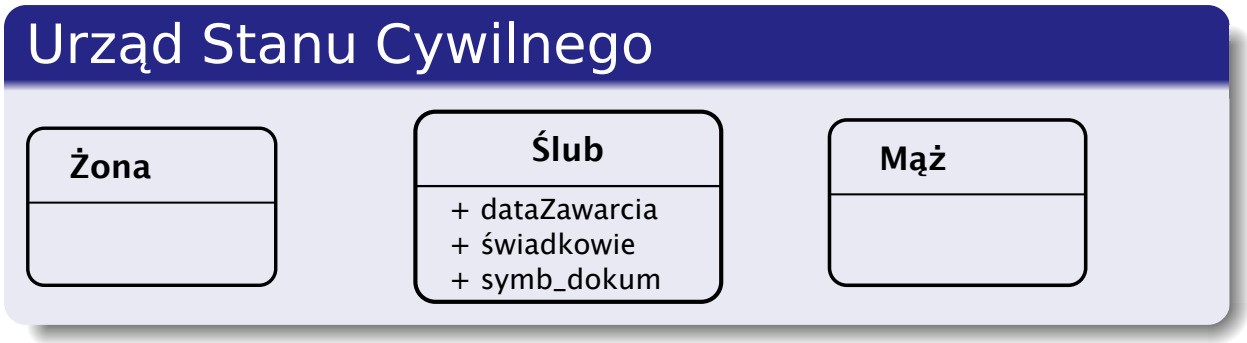
zwykła
referencja

Przykład

Małżeństwo

Prosty system

zwykła
referencja



Skąd się jeszcze biorą obiekty

- Przechowywanie danych
- Interfejsy użytkownika
- Aplikacja jako obiekt (singleton)
- ...

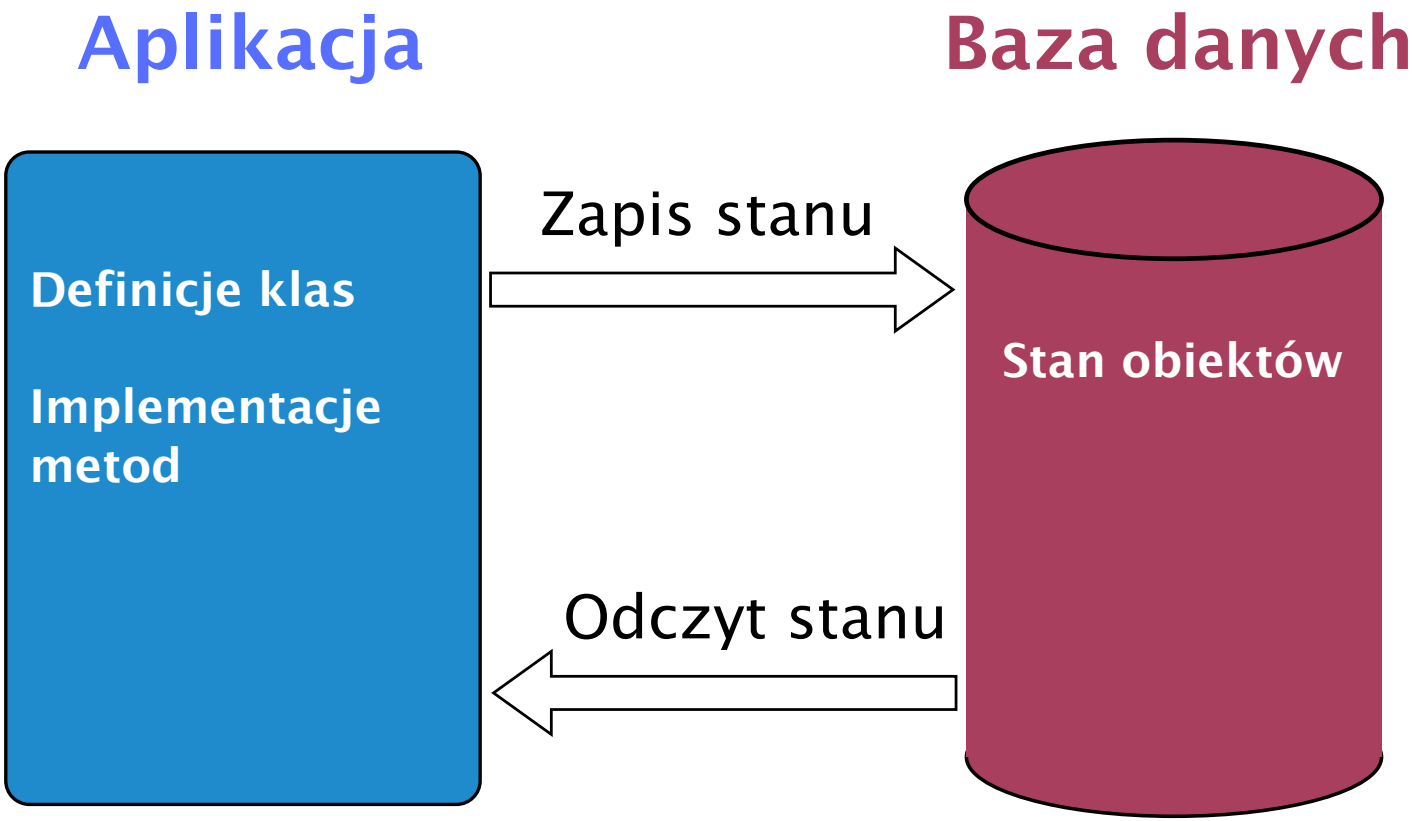
Plan wykładu

Trwałość (persistence)

Definicja

Cecha danej, zmiennej lub obiektu oznaczająca zachowanie jej wartości dłużej niż czas pojedynczego uruchomienia programu.

Przechowywanie stanu w BD



Scenariusz użycia trwałego obiektu

- Odczyt obiektu (stanu obiektu z bd)
- Modyfikacja stanu
- Wywołanie metod obiektu
- Zapis

Zagadnienia

- Zwykle przechowywany jest stan obiektu a nie obiekt (tj. klasa, nadklasa, metody itp)
- Gorliwe czy leniwe aktualizacje (odczyt/zapis) stanu
- Mechanizm aktualizacji zmian (dodawanie nowych pól, zmiana typów pól w nadklasie etc)
- Pytanie: jaka część stanu powinna być przechowywana?

Naturalne podejście

- połączenie aplikacji obiektowej z istniejącym systemem baz danych;
- rozszerzenie środowiska obiektowego o możliwości przetwarzania dużych danych;
- zbudowanie całego środowiska od początku;
- rozszerzenie systemu BD o właściwości obiektowe.

Łączenie środowiska obiektowego z systemem baz danych

- Środowiska obiektowe: JAVA, C#, C++, Python ...
- Systemy baz danych: MySQL, Oracle, PostgreSQL, Sybase, Microsoft SQL Server ...

Składowe integracji języka z bazą danych

- Odwzorowanie cech obiektowych w relacyjnych bazach danych (object-relational mapping, ORM)
- Implementacja warstwy pośredniej (zapytania)

Porównanie modelu relacyjnego z obiektowym

Model obiektowy

- Ukrywanie danych i dostęp tylko przez wskazany interfejs
- Relacje między obiektami: asocjacje, dziedziczenie, kompozycja
- wywołania metod
- Tożsamość obiektów

Model relacyjny

- Rekord
- Tabela: kolekcja rekordów tego samego typu
- Związki między tabelami
- Język zapytań

Java Data Object

- specyfikacja;
- obiekty mogą być pamiętane w plikach, relacyjnych i obiektowych bazach danych;
- szczegóły (gorliwość, leniwość, etc) są definiowane w zewnętrznych plikach xml;
- przykładowa realizacja: Apache JDO, DataNucleus

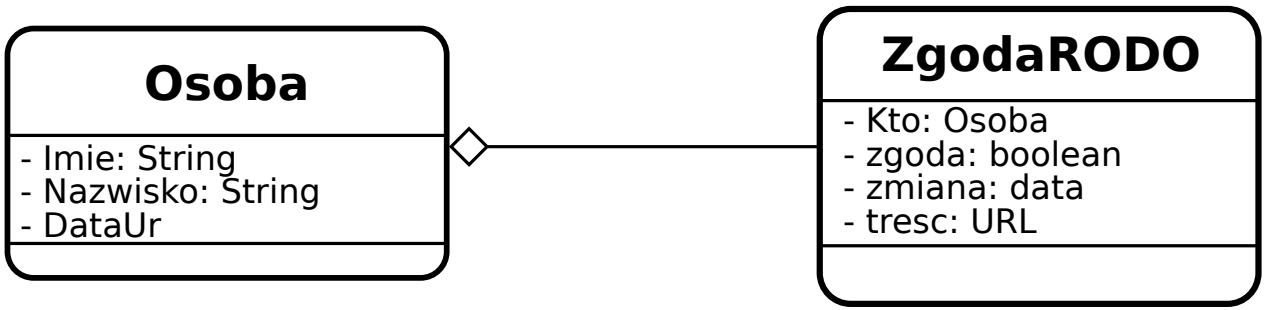
Java Persistence API (JPA)

- specyfikacja;
- głównie do przechowywania stanu obiektów w relacyjnych BD;
- szczegóły są definiowane za pomocą annotacji lub w zewnętrznych plikach;
- JPA Query Language: język zapytań przypominający SQL;
- implementacje: Hibernate, TopLink.

Przykład

Rejestr wyrażonych zgód w ramach dyrektywy RODO. Rejestr dla każdej osoby przechowuje rejestr udzielonych bądź odwołanych zgód.

Model danych w UML



Implementacja klasy Osoba

```
import javax.persistence.*;

public class Osoba
{


    private Integer id;
    String Imie;
    String Nazwisko;
    GregorianCalendar DataUr;
}
```

Implementacja klasy Osoba

```
import javax.persistence.*;

@Entity
@Table
public class Osoba implements Seralizable
{
    @Id
    @GeneratedValue
    private Integer id;
    String Imie;
    String Nazwisko;
    GregorianCalendar DataUr;
}
```

Implementacja klasy ZgodaRODO

```
@Entity
@Table
public class ZgodaRODO implements Seralizable
{
    @ManyToOne
    Osoba kto;
    boolean Zgoda = false;
    GregorianCalendar zmiana;
    URL Tresc;
}
```

Plik persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" ...
             version="2.0">
  <persistence-unit name="ii.progrobiekt.jpa">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>Osoba</class>
    <class>ZgodaRODO</class>

    <properties>
      <property name="dialect" value="org.hibernate.dialect.SQLiteDia
      <property name="javax.persistence.jdbc.driver" value="org.sqlit
      <property name="javax.persistence.jdbc.url" value="jdbc:sqlite:
      <property name="javax.persistence.jdbc.user" value="" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="hibernate.show_sql" value="true" />
      <property name="format_sql" value="true" />
    </properties>
  </persistence-unit>
```

Podstawowe operacje

Create
Read
Update
Delete

Szkielet programu

```
EntityManagerFactory sessionFactory =
  Persistence.createEntityManagerFactory("ii.progobiekt.jpa");
EntityManager entityManager =
        sessionFactory.createEntityManager();
entityManager.getTransaction().begin();
```

```

```

```
entityManager.getTransaction().commit();
entityManager.close();
```

Create

```
entityManager.persist( new Osoba() );
entityManager.persist( new Osoba() );
```

Create

```
entityManager.persist( new Osoba() );
entityManager.persist( new Osoba() );
```

```
INSERT INTO OSOBA (Nazwisko, Imie, DataUr VALUES
("", "", NULL)
```

Read

```
List<Osoba> result =
    entityManager.
    createQuery("from Osoba", Osoba.class ).
    getResultList();
for (Osoba osoba : result )
{
    System.out.println(osoba.ToString());
}
```

Read

```
List<Osoba> result =
    entityManager.
    createQuery("from Osoba", Osoba.class ).
    getResultList();
for (Osoba osoba : result )
{
    System.out.println(osoba.ToString());
}
```

```
SELECT * FROM OSOBA
```

Update

- Jeśli obiekt został pobrany z BD i zmodyfikowany wewnątrz transakcji, wtedy jest automatycznie zapisany po wywołaniu metody `.commit();`
- można zaznaczyć, że obiekt jest *brudny*, gdy manager nie zauważy zmiany (np. zmiany w tablicy).

Delete

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaDelete<User> deleteUser =
    cb.createCriteriaDelete(Osoba.class);
Root<User> cUser = deleteUser.from(Osoba.class);
List<Predicate> predicates = new ArrayList<Predicate>();
predicates.add(cb.equal(cUser.get("id"), id_to_delete));
deleteUser.where(predicates.toArray(new Predicate[] ));
Query query = entityManager.createQuery(deleteUser);
query.executeUpdate();
```

Delete

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaDelete<User> deleteUser =
    cb.createCriteriaDelete(Osoba.class);
Root<User> cUser = deleteUser.from(Osoba.class);
List<Predicate> predicates = new ArrayList<Predicate>();
predicates.add(cb.equal(cUser.get("id"), id_to_delete));
deleteUser.where(predicates.toArray(new Predicate[] ));
Query query = entityManager.createQuery(deleteUser);
query.executeUpdate();
```

```
DELETE FROM OSOBA WHERE OSOBA.id=1203
```

Utworzenie/aktualizacja schematu bazy danych

Każda implementacja ma swoje narzędzia do automatyzacji.

- odpowiedni wpis w pliku `persistence.xml` nakazujący utworzenie/aktualizację BD;
- odpowiedni kod w naszym programie;
- narzędzia zewnętrzne.

C#: ADO.NET

- ADO - ActiveX Data Objects
- ADO.NET – środowisko dostępu do danych w BD

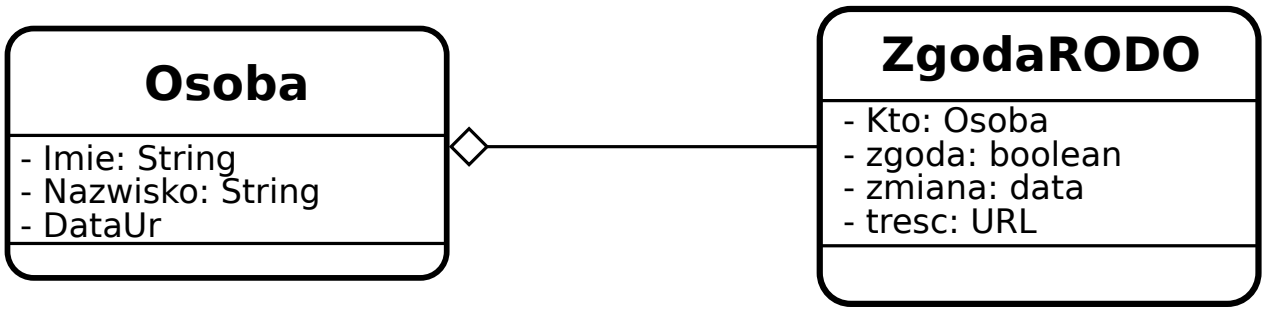
Entity Framework

- Część środowiska ADO.NET
- Powstanie: 2008 rok
- Projekt danych w postaci konceptualnego modelu danych (Entity Data Model)
- Automatyczne odwzorowanie na RBD i model obiektowy
- Zapisywanie modelu danych w pliku XML

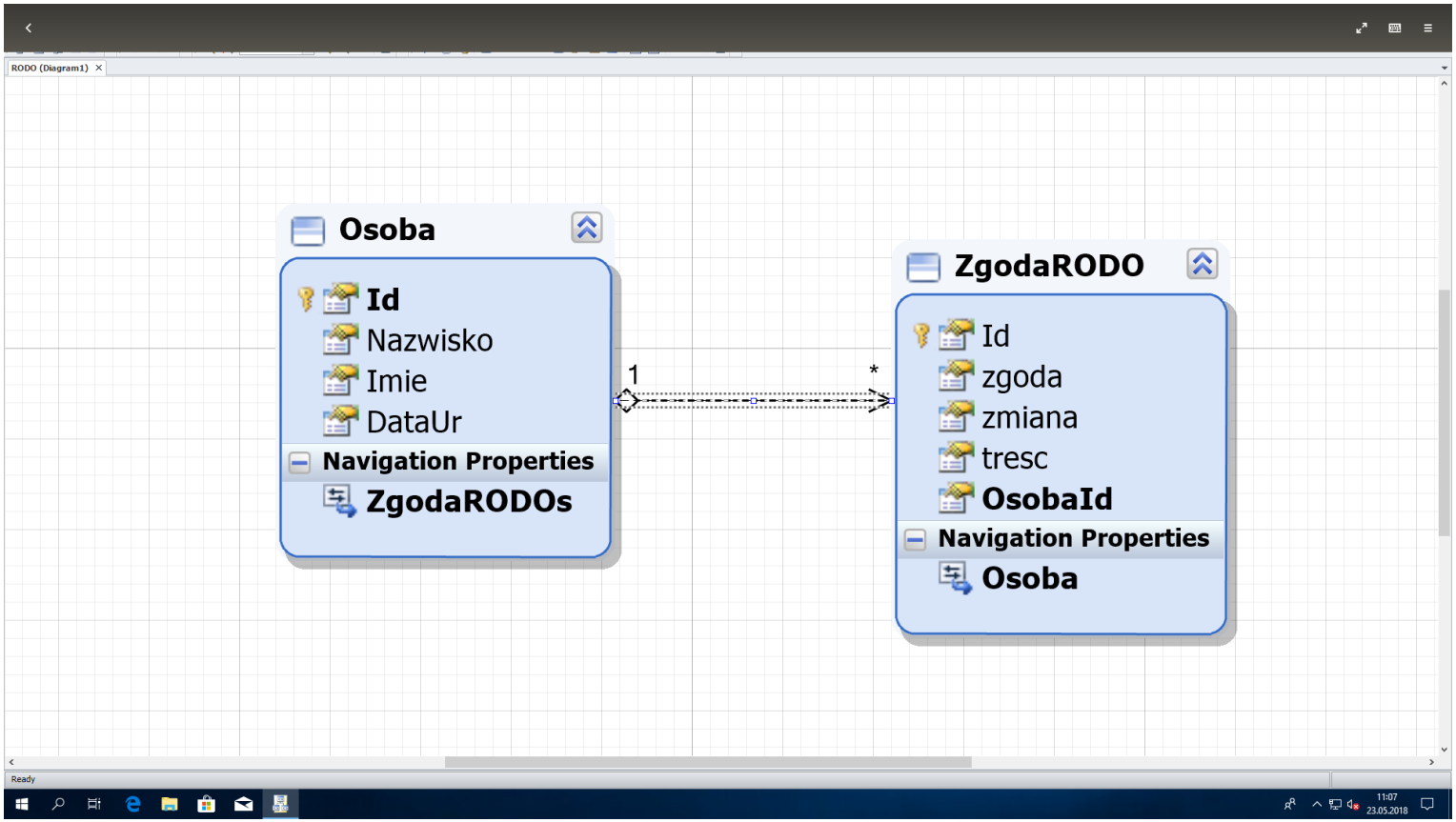
LINQ: Language Integrated Query

- Projekt Microsoftu
- Dostęp na poziomie języka programowania do danych w BD za pomocą składni SQL-podobnej

Model danych w UML (ponownie)



Entity Developer



RODO.Designer.cs

- plik wygenerowany Entity Developerem;
- 553 wiersze.

Przykład LINQ

```
Zgody db = new Zgody(connectionString);
var q = from o in db.Osoba, z in db.ZgodaRODOs
    where o.Id == z.OsobaId and z.zgoda and o.Nazwisko =
    select new {z.zmiana, c.tresc};
foreach (var t in q) { ... }
```

RUBY

ActiveRecords

- każdej klasie będącej podklasą klasy Base odpowiada tabela w bd;
- narzędzia do tzw. migracji;
- obiektowy dostęp do danych

```
Osoba.find(:all,
        :conditions => "nazwisko = 'Młotkowski'")
```

Rozszerzenia RBD na przykładzie Oracle

- PL/SQL – rozszerzenie SQL o procedury
- Składnia zapożyczona z Ady
- Możliwość definiowania własnych programów wykonywanych na serwerze
- Możliwość deklarowania własnych typów danych (klas)
- Rozszerzenie SQL o odwołania do obiektów

Serwer STONE

- Przechowuje obiekty
- Obiekty nie są kopiowane do aplikacji klienta, tylko zostają w serwerze
- Metody są wykonywane na serwerze

Rozwiązanie w Smalltalku

Przechowywanie stanu aplikacji w jednym pliku (obrazie).

Wady rozwiązania

- Kłopoty z przetwarzaniem dużych porcji danych
- Kłopoty z szybkim wyszukiwaniem danych
- Problem wielodostępu do danych

Plan wykładu

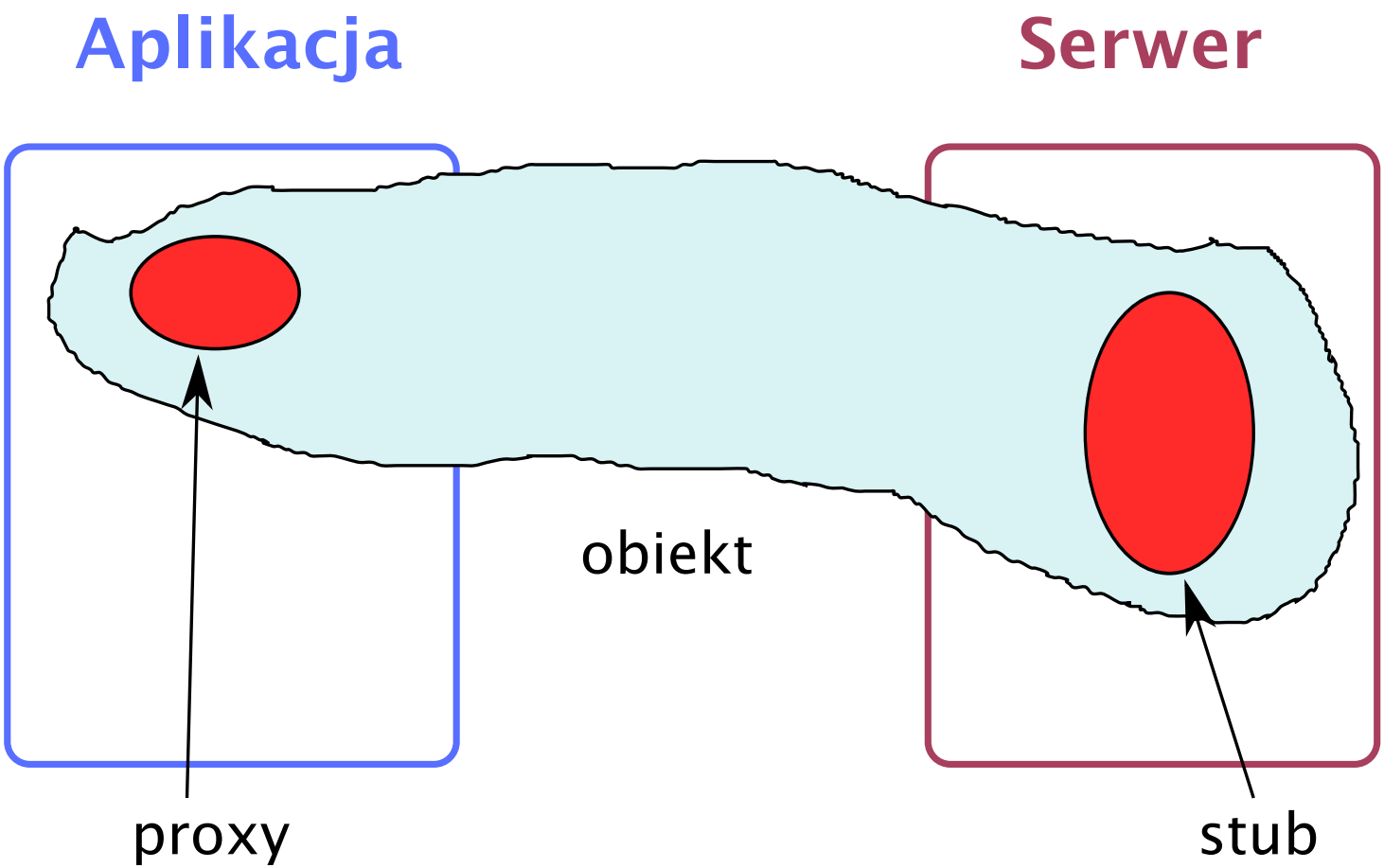
Obiekty trwałe: inna koncepcja

- Obiekt (jego stan) oraz implementacja metod jest pamiętana na serwerze
- Wykonanie metody powoduje odwołanie się do zdalnego obiektu, wykonanie metody na serwerze i zwrócenie klientowi wyniku
- Serwer dba o spójność danych, wielodostęp, transakcyjność etc

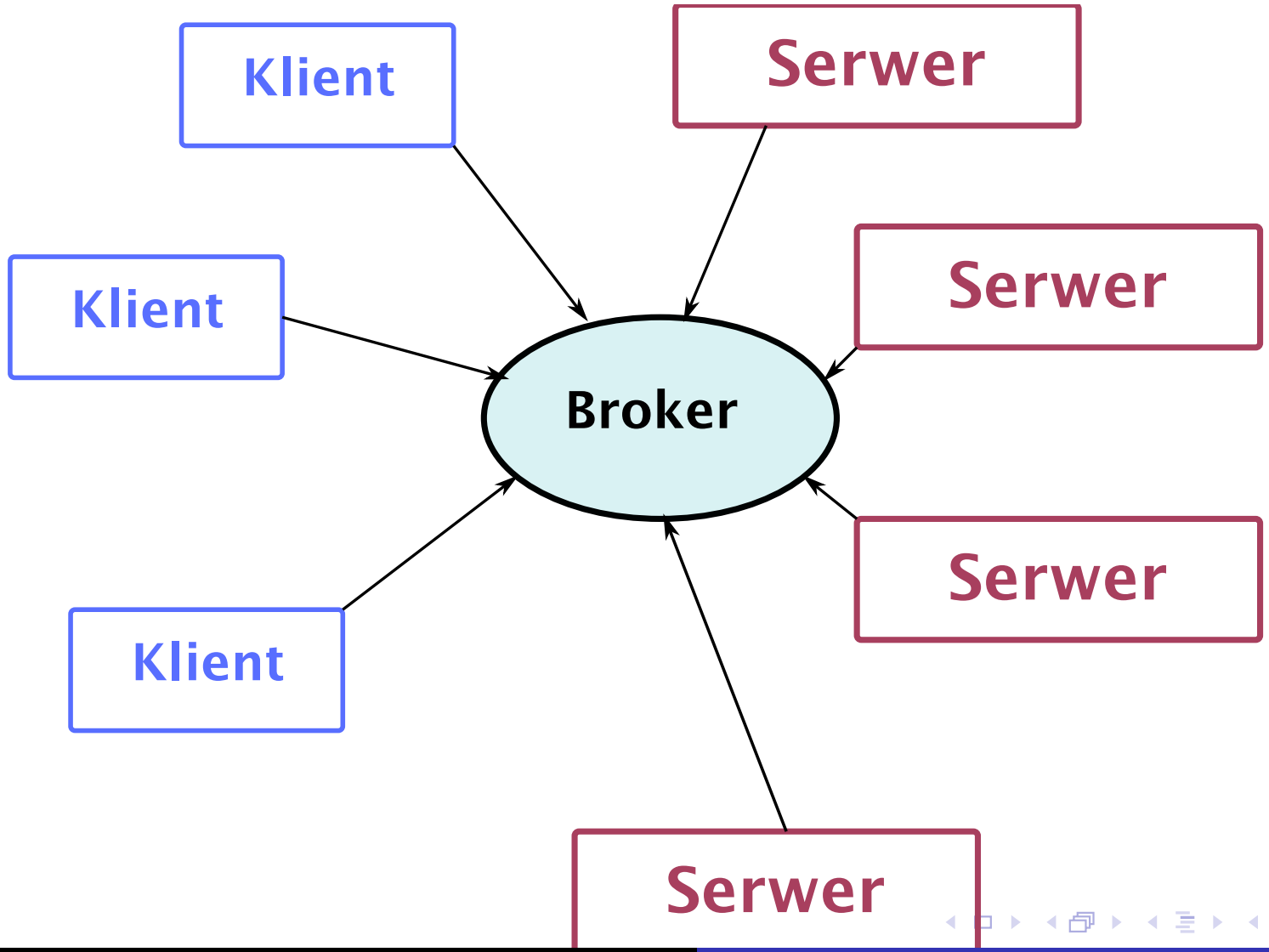
Postulaty dotyczące takiego środowiska

- Jednolity dostęp do obiektów lokalnych i odległych
- Łatwość tworzenia odwołań do odległych obiektów
- Niezależność standardów od platformy

Architektura



Broker



Środowiska obiektów rozproszonych

- Java RMI
- CORBA
- SOAP
- DCOM

Java RMI

- RMI – Remote Method Invocation
- Współpraca tylko między aplikacjami napisanymi w Javie
- java.rmi.*
- Całe środowisko jest częścią SDK

CORBA

- Zbiór standardów opracowanych przez OMG (Object Management Group)
- Niezależność od platformy/sprzętu/języka
- Implementacje: VisiBroker, ORBit
- Bardzo obszerna specyfikacja

SOAP

- Simple Object Access Protocol
- Oparty na XML
- Standard W3C
- Implementacje: .NET Remoting, Apache SOAP
- .NET Remoting: również realizacja binarna

DCOM

- Distributed Component Object Model
- Implementacja tylko na systemy Microsoftu
- Uznany za przestarzały na rzecz .NET Remoting