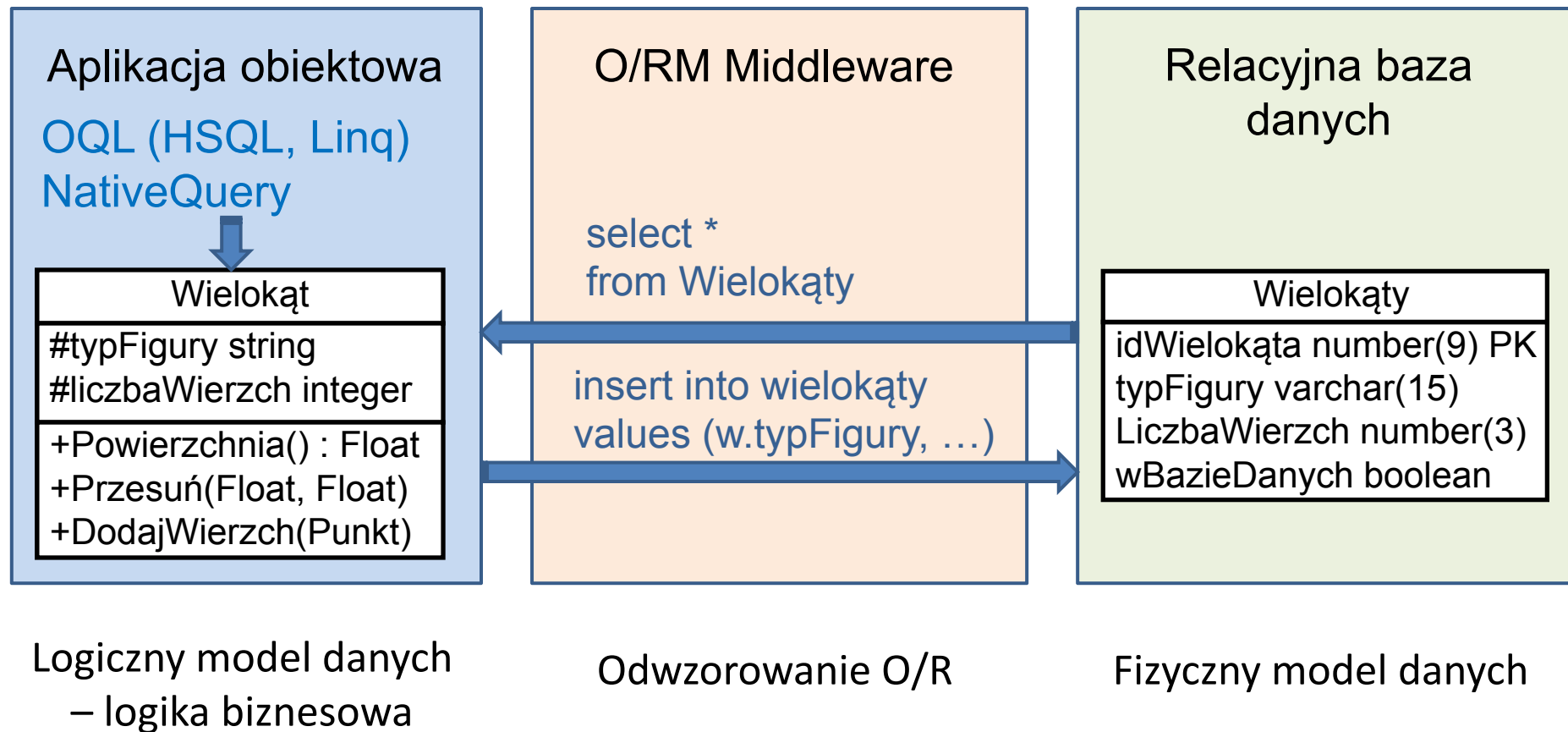


# **Odwzorowanie obiektowo-relacyjne**

Wykład opracował:  
Tomasz Koszlajda

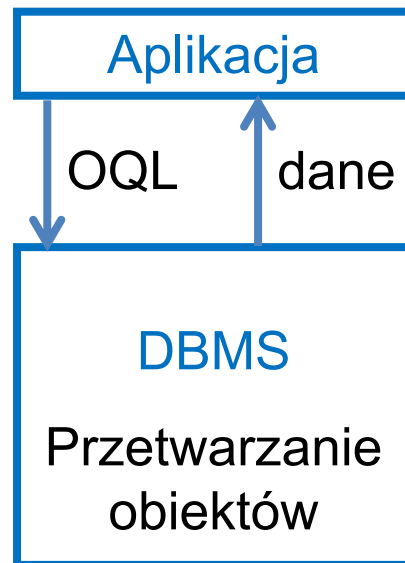
# Odwzorowanie obiektowo-relacyjne

- ▶ Odwzorowanie obiektowej architektury systemu informatycznego w relacyjne struktury danych bazy danych

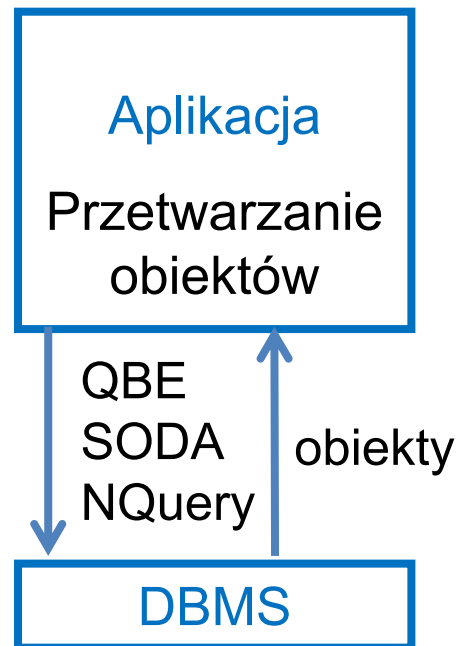


# Architektura przetwarzania O/RM

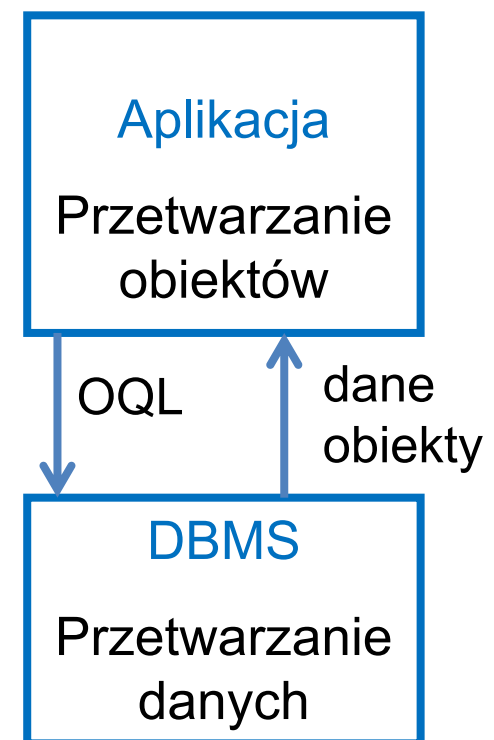
O-R DB



OO DB



O/RM

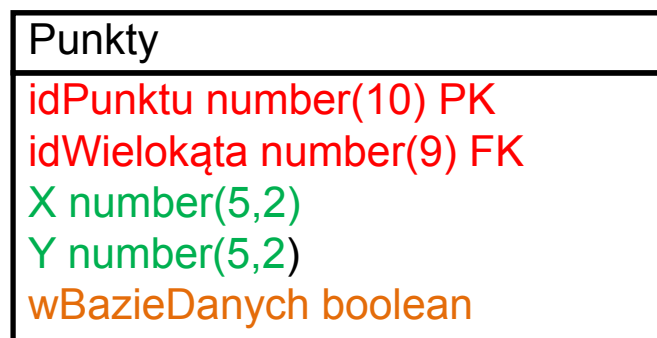
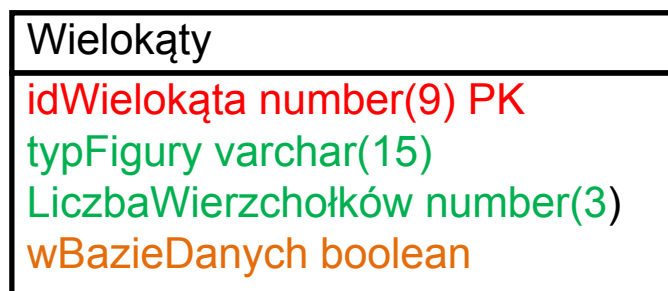
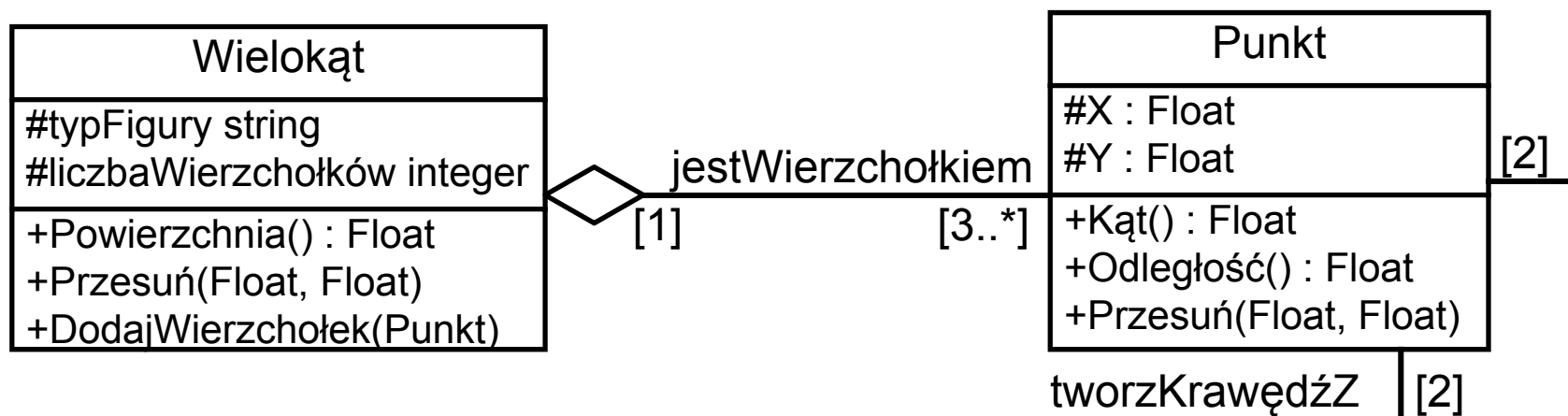


# Rozwiązania O/RM

- ▶ Odwzorowanie modelu obiektowego w relacyjny
  - ▶ Odwzorowanie prostych atrybutów
  - ▶ Odwzorowanie OID
  - ▶ Odwzorowanie sieci dziedziczenia klas
  - ▶ Odwzorowanie związków między klasami
  - ▶ Odwzorowanie atrybutów klas
- ▶ Zarządzanie transakcjami
- ▶ Zarządzanie obiektami

# Dodatkowe atrybuty implementacyjne

- ▶ Oprócz odwzorowania atrybutów informacyjnych w O/RM niezbędne jest utrzymywanie dodatkowych atrybutów implementacyjnych



- ▶ Klucze podstawowe i obce
- ▶ Zapewnienie trwałości danych
- ▶ Znaczniki czasowe

Fizyczny model danych

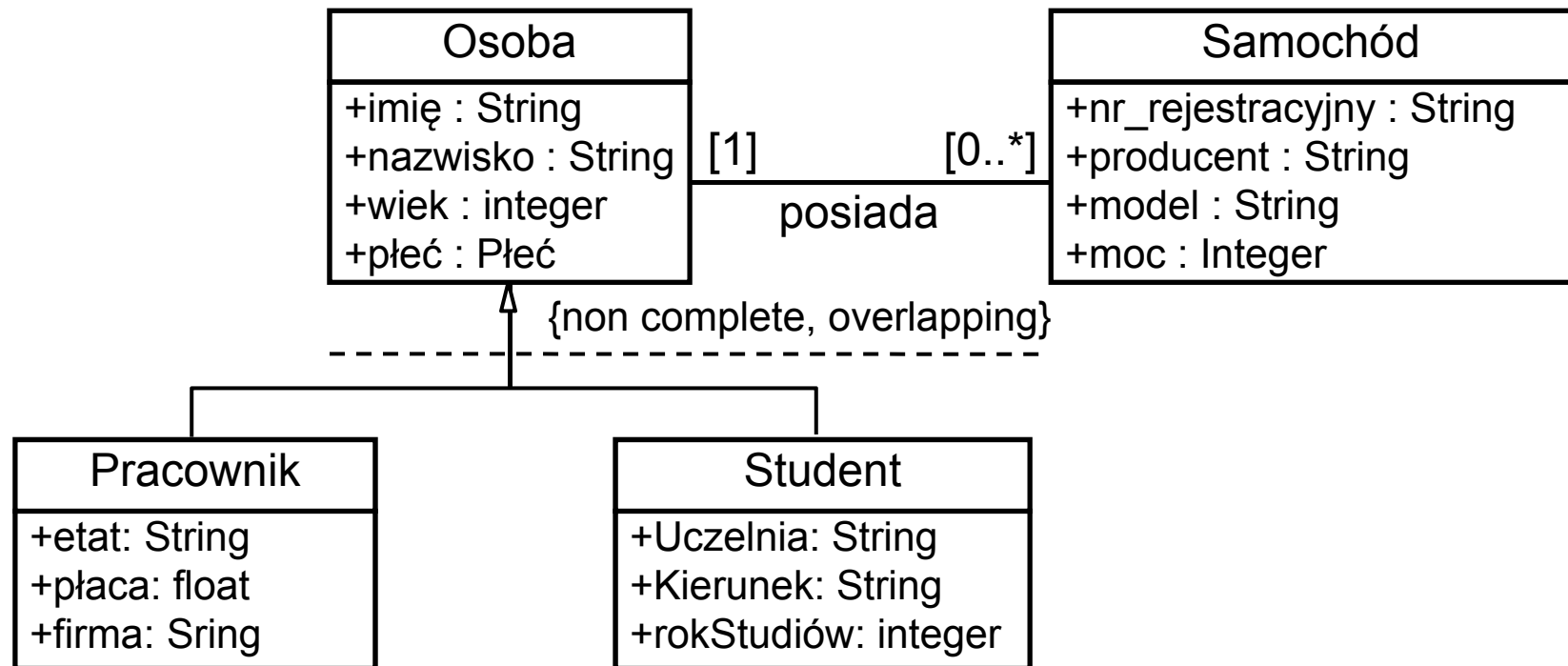
# Dodatkowe atrybuty implementacyjne

- ▶ Narzędzia O/RM wspierają korzystanie z atrybutów implementacyjnych. Na przykład w JPA przewidziano np. definiowane identyfikatorów obiektów, wersji obiektów lub atrybutów informujących o klasie obiektu:

```
@Entity
public class Osoba implements Serializable {
    ...
    @Version
    @Column(name="OPTIMISTIC_LOCK")
    public Integer getVersion() { ... }
    @Id
    @Column(name="id_osoby")
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
        generator="SEQ_STORE")
}
```

# Przykładowa definicja klas

- ▶ Dany model klas przetwarzanych przez aplikacje bazy danych



# Odwzorowanie prostych atrybutów

Przykład odwzorowania klas w schemat relacyjnej bazy danych za pomocą adnotacji języka Java w O/RM EJB3/JPA.

```
@Entity
@Table(name="t_Osoby")
public class Osoba implements Serializable {
    public String imię;
    @Column(name="t_imię", nullable = false, length = 25);
    public String nazwisko;
    @Column(name="t_nazwisko");
    ...
}
```



# Odwzorowanie OID

W JPA możliwe określanie OID obiektów (@id) oraz sposobu generowania jego wartości @GeneratedValue. Dostępne są następujące sposoby generowania wartości OID:

- ▶ TABLE - relacja dedykowana do generowania OID
- ▶ IDENTITY – atrybut relacji przechowujący obiekty
- ▶ SEQUENCE – systemowy *sekwenser*
- ▶ AUTO – automatycznie jedno z powyższych

```
@Entity
```

```
@Table(name="t_Osoby")
```

```
public class Osoba implements Serializable {
```

```
    private Long id;
```

```
    @Id
```

```
    @Column(name="id_osoby")
```

```
    @GeneratedValue(strategy=GenerationType.SEQUENCE,  
                    generator="SEQ_STORE")
```

```
    public Integer getId() { ... }
```

Cecha Id nie posiada wszystkich cech OID.

# Odwzorowanie związków

W JAP możliwe jest odwzorowanie związków typu:

- ▶ 1:1 - @OneToOne,
- ▶ 1:N - @ManyToOne, @OneToMany,
- ▶ M:N - @ManyToMany,

Jako związków jedno lub dwukierunkowych.

Do określenia siły powiązania służy opcja Cascade:

- ▶ CascadeType.PERSIST – oznacza, że utrwalenie danego obiektu pociąga za sobą automatyczne utrwalenie obiektu powiązanego.
- ▶ CascadeType.MERGE – oznacza, że dla odłączonego od bazy danych obiektu, modyfikacje obiektu powiązanego będą synchronizowane metodą MERGE razem z obiektem głównym.
- ▶ CascadeType.REMOVE – oznacza, że usunięcie danego obiektu pociągnie za sobą usunięcie obiektu powiązanego.
- ▶ CascadeType.ALL – wszystkie powyższe.

Trwałe związki są implementowane za pomocą współdzielonych kluczy podstawowych (1:1), pary klucz podstawowy, klucz obcy lub za pomocą relacji połączeniowych.

# Odwzorowanie związków

Przykład implementacji związku 1:N w JPA

```
@Entity()  
public class Samochód implements Serializable  
{  
    @ManyToOne(cascade = {CascadeType.REMOVE} )  
    @JoinColumn(name="id_właściciela")  
    public Właściciel GetWłaściciel() {  
        return właściciel;  
    }  
    ...  
}
```

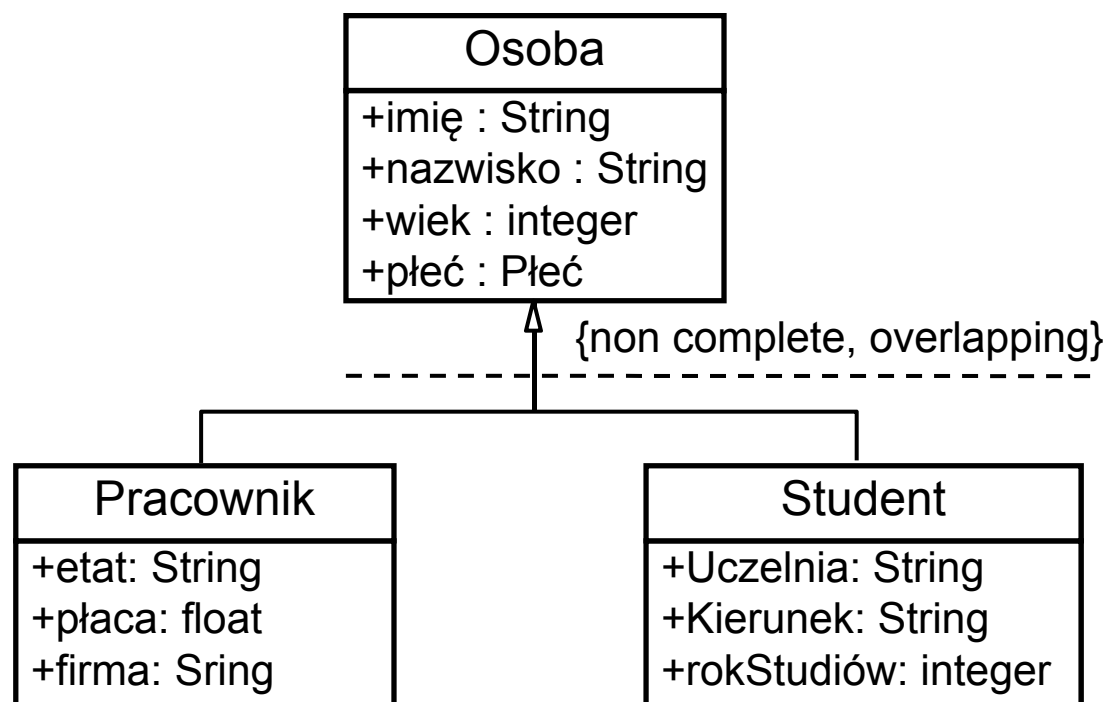
# Atrybuty wielowartościowe

Adnotacja `@OneToMany` służy do implementacji atrybutów wielowartościowych modelowanych w aplikacji jako: `Collection`, `List`, `Set`.

```
@Entity public class Miasto {  
    @OneToMany(mappedBy="miasto")  
    @OrderBy("Ulica")  
    public List<Ulica> getUlice() {  
        ...  
    }  
}  
  
@Entity public class Ulica {  
    @ManyToOne  
    public Miasto getMiasto() {  
        ...  
    }  
}
```

# Odwzorowanie sieci dziedziczenia

Znane są następujące metody odwzorowania:



- ▶ Odwzorowanie hierarchii klas w pojedynczą tablicę
- ▶ Odwzorowanie *konkretnych* klas w osobne tabele
- ▶ Odwzorowanie każdej klasy w osobną tabelę
- ▶ Odwzorowanie klas w uniwersalną strukturę tabel

# Odwzorowanie hierarchii klas w pojedynczą tabelę

Osoby
Imię
Nazwisko
Płeć
Etat
Płaca
Firma
Uczelnia
Kierunek
RokStudiów
jestStudentem
jestPracownikiem

Własności:

- Tabela będzie przechowywała dużo wartości pustych
- Tabela będzie zajmować większą powierzchnię na dysku
- Łatwość przetwarzania polimorficznego – wszystkie dane w jednej tabeli
- Duża wydajność dla przetwarzania polimorficznego
- Mała wydajność dla przetwarzania homogenicznego – tabela zawiera niepotrzebne dane
- Mała wydajność dla modyfikacji struktury pojedynczych klas
- Trudne utrzymywanie przynależności do typów

**Dobre dla prostych i płaskich hierarchii klas**

# Odwzorowanie hierarchii klas w pojedynczą tabelę

JPA wspiera odwzorowanie klas w jedną tabelę. Dla rozłącznych i kompletnych podzbiorów – proste warunki weryfikacji klasy.

```
@Entity
@Table(name="t_Osoby")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="t_płeć")
public class Osoby{ ... }

@Entity
@DiscriminatorValue("K")
public class Kobieta extends Osoba { ... }

@DiscriminatorValue("M")
public class Mężczyzna extends Osoba { ... }

// transformacja zapytań
select k from Kobieta k ->
    select ... from t_Osoby where t_płeć='K'
select o from Osoba o ->
    select ... from t_Osoby
```

# Odwzorowanie hierarchii klas w pojedynczą tabelę

Dla nierozłącznych podzbiorów niezbędne jest stosowanie złożonych warunków weryfikacji klasy.

```
@Entity
@Table(name="t_Osoby")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="jestStudentem")
@DiscriminatorColumn(name="jestPracownikiem")
@DiscriminatorFormula(
    "case when jestStudentem=True then 0
      when jestPracownikiem=True then 1 end")
public class Osoby{ ... }

@Entity
@DiscriminatorValue(0)
public class Student extends Osoba { ... }

@DiscriminatorValue(1)
public class Pracownik extends Osoba { ... }
```



# Odzworowanie konkretnych klas w osobne tabele

- Jeżeli klasa Osoba jest klasą abstrakcyjną
- Zależność między podzbiorami danych typu *complete*

Pracownicy	Studenci
idPracownika	idStudenta
Imię	Imię
Nazwisko	Nazwisko
Płeć	Płeć
Etat	Uczelnia
Płaca	Kierunek
Firma	RokStudiów

## Własności:

- Trudna implementacja *overlapping* - konieczność implementacji wspólnej dziedziny dla kluczy relacji
- Mała wydajność dla przetwarzania polimorficznego – dane w różnych tabelach
- Duża wydajność dla przetwarzania homogenicznego
- Modyfikacja klasy z podklasami wymaga modyfikacji schematów wielu tabel

**Dobre dla rozłącznych podzbiorów rozszerzeń klas**

# Odwzorowanie konkretnych klas

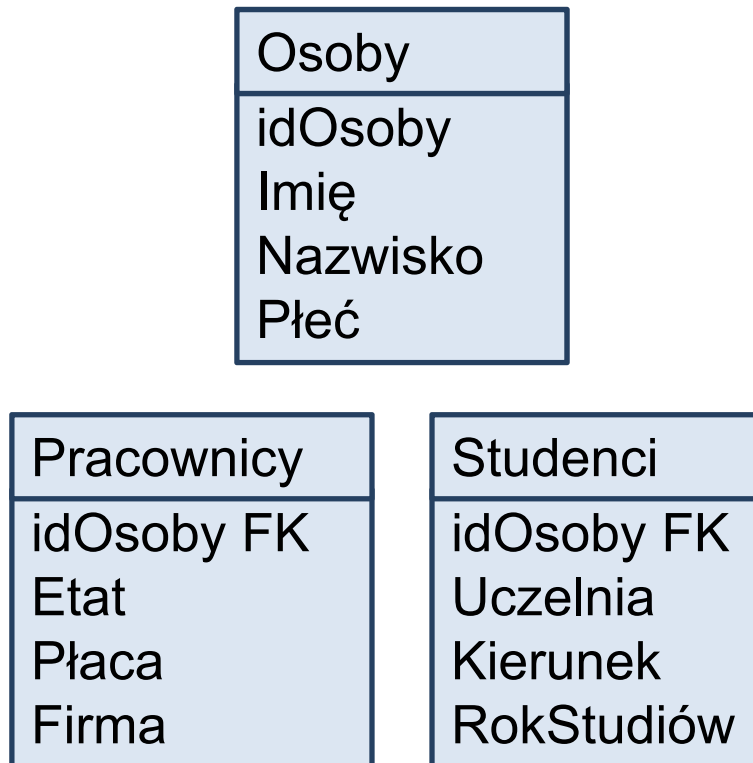
JPA wspiera odwzorowanie konkretnych klas w tabelę za pomocą strategii „Table per class” (Hibernate – „Union class”).

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Osoby{ ... }
@Entity
@Table(name="t_Kobiety")
public class Kobieta extends Osoba { ... }
@Table(name="t_Mężczyźni")
public class Mężczyzna extends Osoba { ... }
```

## // transformacja zapytań

```
select o from Kobieta k ->
    select * from t_Kobiety
select o from Osoba o ->
    select ... from t_Kobiety
    union
    select ... from t_Mężczyźni
```

# Odwzorowanie każdej klasy w osobną tabelę



## Własności:

- Proste odwzorowanie 1:1
- Łatwość przetwarzania polimorficznego
- Łatwe utrzymywanie dla modyfikacji nadklas i dodawania podklas
- Rozmiar struktur proporcjonalny do liczby danych
- Duża liczba tablic
- Mała wydajność przetwarzania homogenicznego – konieczność łączenia tabel

**Dobre dla pokrywających się podzbiorów rozszerzeń klas i często zmieniających się definicji klas**

# Odwzorowanie każdej klasy w osobną tabelę

JPA wspiera odwzorowanie klasy w osobną tabelę za pomocą strategii „Joined subclass”.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="t_Osoby")
public class Osoby{ ... }

@Entity
@Table(name="t_Kobiety")
@PrimaryKeyJoinColumn(name="id_osoby")
public class Kobieta extends Osoba { ... }
```

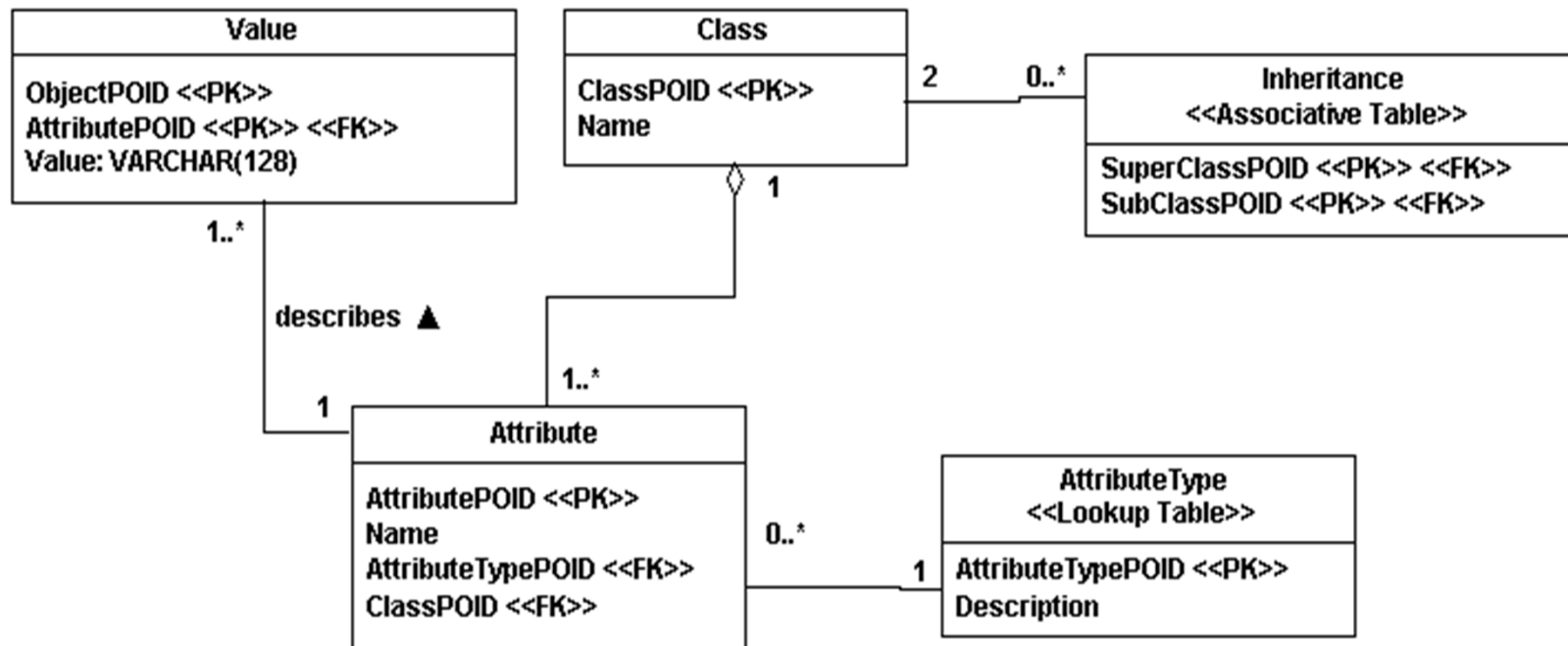
**// transformacja zapytań**

```
select o from Kobieta k ->
    select o.*, k.*
    from t_Osoby o JOIN t_Kobiety k
        ON o.id_osoby = k.id_osoby
```

# Odwzorowanie klas w uniwersalną strukturę tabel

Własności:

- ▶ Łatwość utrzymania dla zmian schematu
- ▶ Trudne i mało wydajne przetwarzanie



Copyright 2002-2006 Scott W. Ambler

**Dobre dla złożonych aplikacji przetwarzających  
niezbyt duże zbiory danych**

# Odzworowanie cech klas

## Osoby

+imię : String  
+nazwisko : String  
+płeć : Płeć  
+liczbaOsób : Integer  
+procentKobiet: Integer

## Osoby\_LiczbaOsób

LiczbOs

## Osoby

LiczbaOsób	ProcentKobiet
------------	---------------

## ZmienneKlasowe

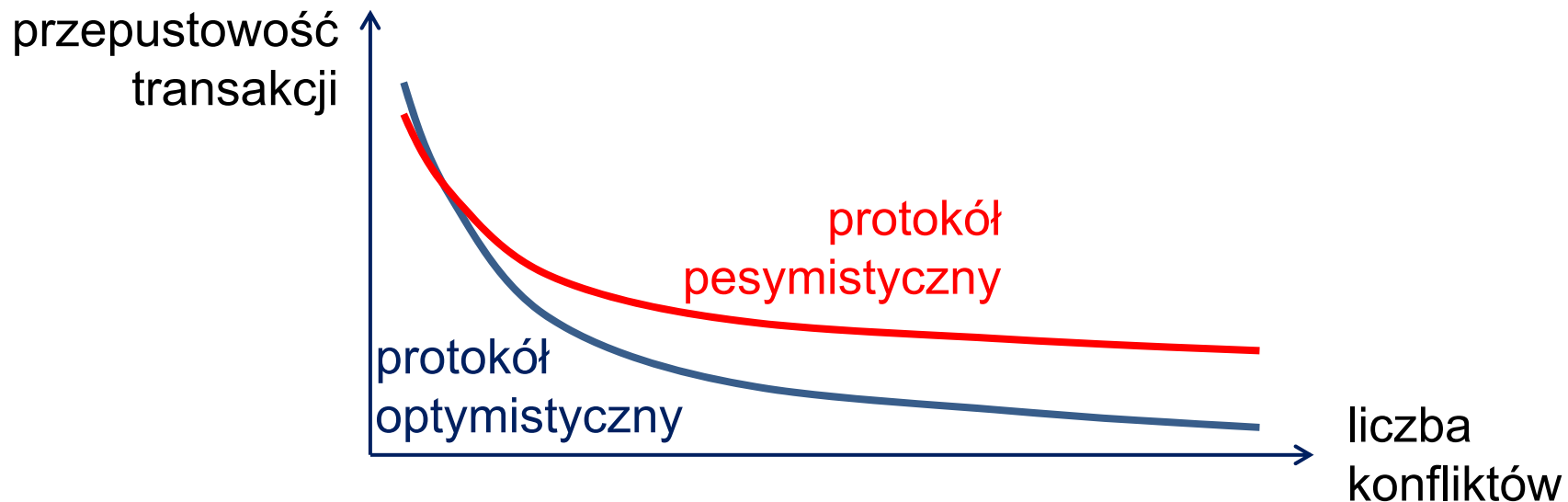
Os_LiczbaOsób	Os_ProcentKobiet	...
---------------	------------------	-----

### ► Możliwe strategie:

- Jednokolumnowa i jednowierszowa tabela dla każdej cechy
- Wielokolumnowe i jednowierszowe tabele dla każdej z klas
- Wielokolumnowa i jednowierszowa tabela wspólna dla wszystkich klas
- Wielowierszowa tabela o generycznym schemacie wspólna dla wszystkich klas

# Optymistyczna synchronizacja transakcji

- ▶ W większości protokołów gwarantujących poprawność współbieżnego przetwarzania transakcji implementowane jest *pesymistyczne* podejście do synchronizacji, polegające na modyfikacji historii konfliktowych transakcji przez czasowe zawieszenie działania jednej z nich.
- ▶ Optymistyczne protokoły synchronizacji transakcji przesuwają wykrywanie potencjalnych konfliktów do czasu akceptacji transakcji. Wykryte konflikty są obsługiwane przez wycofanie jednej z konfliktowych transakcji.



# Optymistyczna synchronizacja transakcji

- ▶ Schemat protokołu optymistycznej synchronizacji współbieżności zakłada przeprowadzenie każdej transakcji przez trzy kolejne fazy:
  - ▶ Faza odczytów i lokalnych modyfikacji – potencjalnie długa i obejmująca interakcje z użytkownikiem transakcji.
  - ▶ Faza walidacji, która zaczyna się w punkcie akceptacji transakcji i służy do weryfikacji możliwości poprawnego zakończenia transakcji.
  - ▶ Synchronizowana transakcyjnie faza zapisu danych buforowanych w PAO do bazy danych.
- ▶ Zaletami optymistycznej synchronizacji transakcji są: niewystępowanie w tym protokole zakleszczeń transakcji oraz mniejsze obciążenie zasobów DBMS (brak blokad).
- ▶ Optymistyczna synchronizacja transakcji nie gwarantuje pełnej uszeregowalności – nie obsługuje anomalii fantomów.



# Optymistyczna synchronizacja transakcji

Algorytm fazy walidacji jest następujący [H.T.Kung, J.T.Robinson – 1981]:

**Validation ( $T_j$ )**

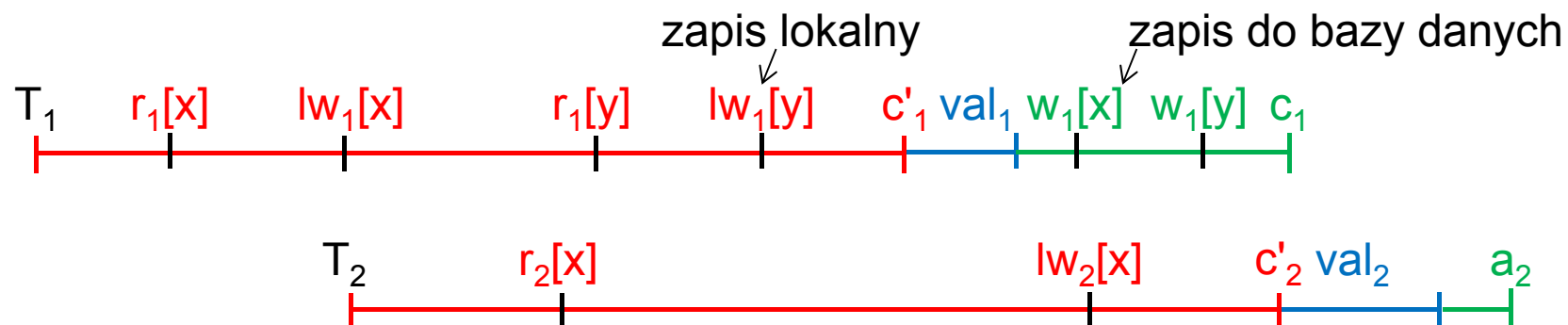
```
<valid := true;    // początek sekcji krytycznej
  for  $TNR_i$  from  $TNR_{start+1}$  to  $TNR_{finish}$  do
    if  $RS_j \cap WS_i \neq \emptyset$  then valid:=false;
  if valid then
  begin
    (write);
     $TNR_j := TNC$ ;
     $TNC := TNC+1$ 
  end> // koniec sekcji krytycznej
  if not valid then (rollback);
```

gdzie:

$TNR_{start}$  jest największym znacznikiem transakcji w momencie startu transakcji  $T_j$ ,  $TNR_{finish}$  jest największym znacznikiem transakcji w momencie rozpoczęcia fazy walidacji, a RS i WS są zbiorami czytanych i modyfikowanych danych.

# Działanie algorytmu

Przykład synchronizacji dwóch transakcji  $T_1$  i  $T_2$ .



$$TNR(T_1) = 10$$

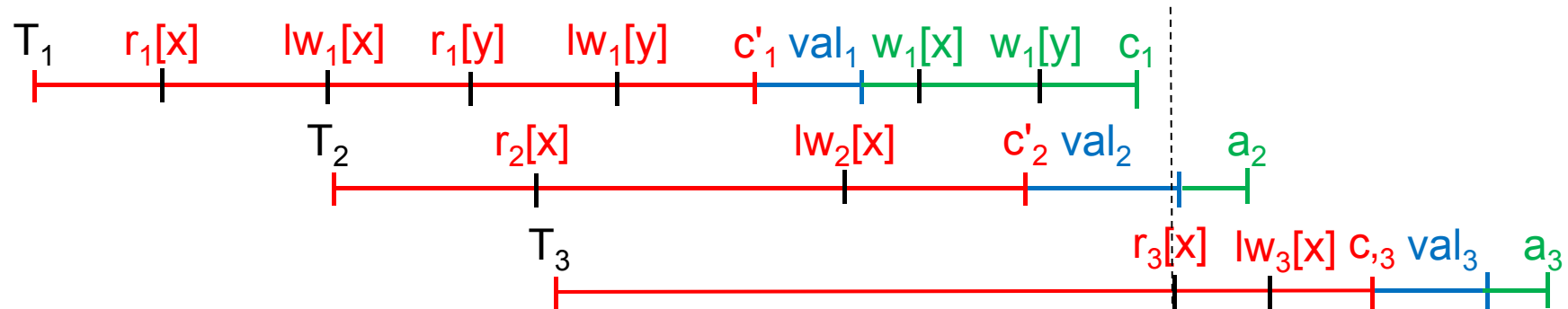
Walidacja transakcji  $T_2$ :

- ▶  $TNR_{start+1} = 10$ ,  $TNR_{finish} = 10$
- ▶  $RS(T_2) = \{x\}$ ,  $WS(T_1) = \{x, y\}$ ,
- ▶ skąd:  $RS(T_2) \cap WS(T_1) = \{x\} \neq \emptyset$

Transakcja  $T_2$  musi zostać wycofana.

# Ulepszona wersja algorytmu

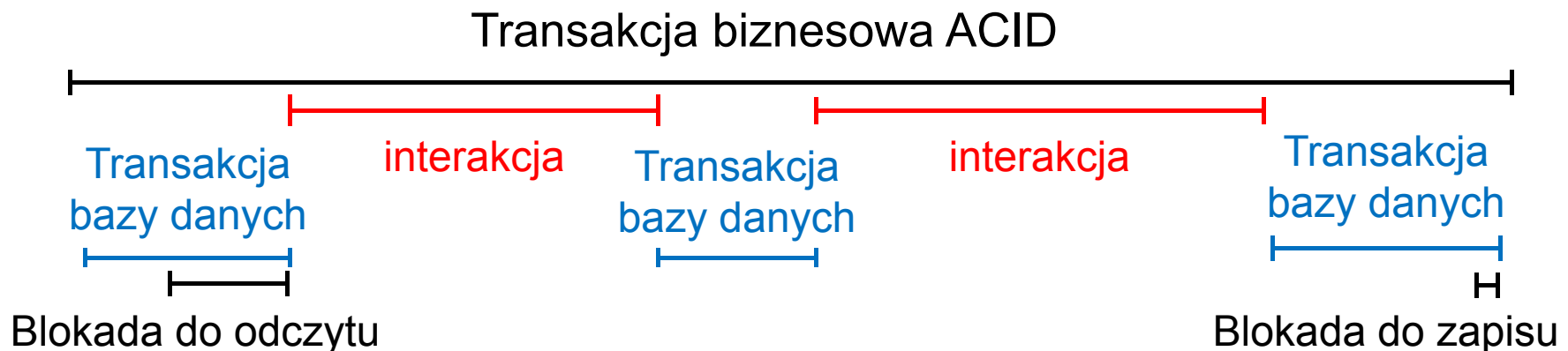
## ► Pomijanie nieistotnych konfliktów



- Transakcja  $T_3$  zostanie wycofana, mimo że dysponuje zatwierdzoną wartością danej  $x$ . Wykryty konflikt jest nieistotny dla poprawności przetwarzania, ale jest przyczyną wycofania transakcji.
- Modyfikacja protokołu OC o rozszerzenie zbioru danych czytanych przez transakcję o znaczniki końców współbieżnych transakcji. Dla transakcji  $T_3$ , zbiór czytanych danych  $RS_3 = \{EOT_1, x\}$ .
- Weryfikowany w fazie walidacji warunek zostanie zmodyfikowany do postaci:  $RS_{ij} \cap W_i \neq \emptyset$ , gdzie  $RS_{ij}$  jest podzbiorem tylko tych danych, które zostały odczytane po zakończeniu transakcji  $T_i$ .  $RS_{13} = \emptyset$ , co znaczy, że w zmodyfikowanym protokole transakcja  $T_3$  nie zostanie wycofana.

# Zarządzanie współbieżnością transakcji

- ▶ W środowiskach O/RM zalecany sposób synchronizacji transakcji są protokoły zarządzania współbieżnymi transakcjami wzorowane na podejściu optymistycznym. Modyfikacje obiektów są wykonywane lokalnie. Ich zapis do bazy danych, realizowany przez systemową operację *flush*, jest przesuwany na koniec transakcji. W związku z tym blokady do zapisu są utrzymywane krótkoterminowo, tylko na czas walidacji transakcji.
- ▶ Dodatkowe rozwiązania mające na celu zwiększenie skalowalności współbieżnego dostępu zakładają, że transakcje systemu bazy danych nie obejmują czasu interakcji z użytkownikiem aplikacji bazy danych. W efekcie również blokady do odczytu zakładane przez system bazy danych są krótkoterminowe. Logiczna transakcja (biznesowa) składa się z kilku transakcji systemu bazy danych.



# Zarządzanie wersjami obiektów

- ▶ W środowiskach O/RM do implementacji protokołów optymistycznej synchronizacji transakcji stosuje się rozproszony algorytm zarządzania. Transakcje biznesowe synchronizują się ze sobą poprzez weryfikację w fazie walidacji stanu bazy danych.
- ▶ Walidacja transakcji odbywa się poprzez kontrolę wersji modyfikowanych danych. Kontrola wersji może być realizowana ręcznie przez aplikację lub wspierana systemowo.
- ▶ Weryfikacja wersji obiektów odbywa się przez kontrolę: znaczników czasowych, identyfikatora wersji obiektu lub przez porównanie wartości obiektu zapamiętanych z fazy odczytu z wartością odczytaną w fazie walidacji.

# Synchronizacja transakcji przez aplikację

Każda interakcja z bazą danych jest osobną sesją. Aplikacją musi załadować/przeładować z bazy danych wszystkie obiekty przetwarzane w ramach danej interakcji oraz kontrolować wersje modyfikowanych danych dla uniknięcia anomalii *lost update*.

// o1 jest wersją obiektu załadowanego przez poprzednią sesję

```
session = factory.openSession();
```

```
Transaction t = session.beginTransaction();
```

```
int oldVersion = o1.getVersion();
```

```
session.load(o1, o1.getKey() ); // ładuj aktualną wersję
```

```
if ( oldVersion != o1.getVersion() )
```

```
    throw new StaleObjectStateException();
```

```
o1.setProperty("J23");
```

```
t.commit(); // wykonanie metody flush – zapis o1 do bazy danych
```

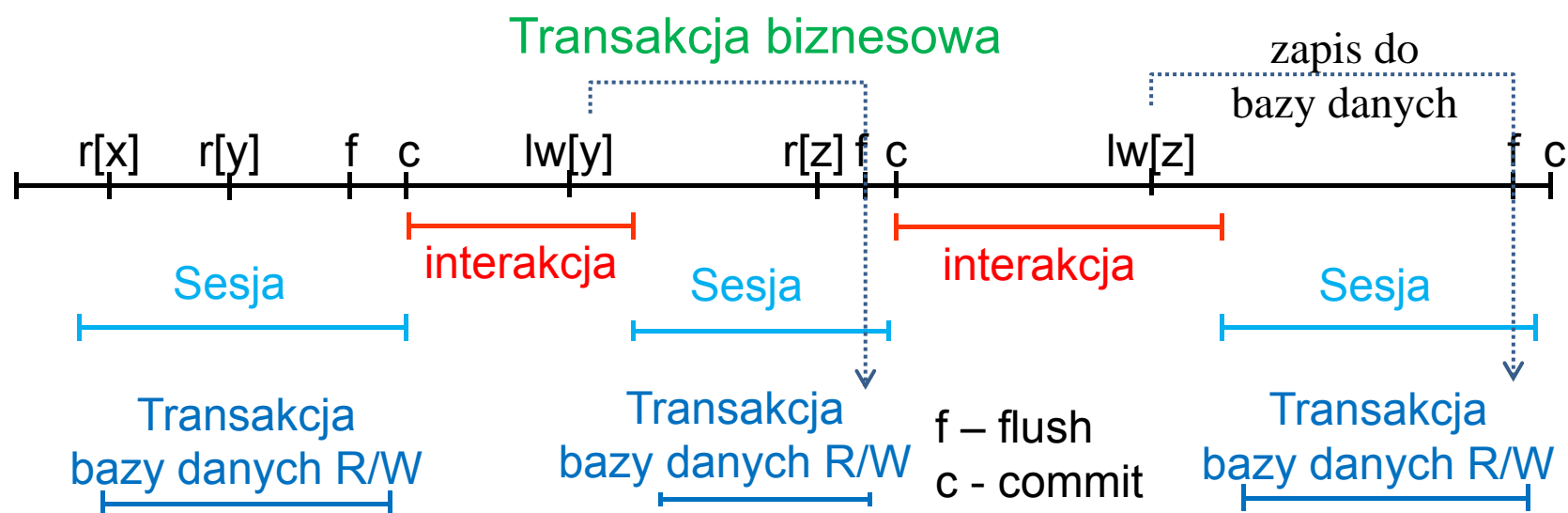
```
session.close();
```

// ciąg dalszy transakcji biznesowej

# Synchronizacja transakcji przez aplikację

Rozwiązanie charakteryzuje się w ogólności niskim poziomem izolacji. Transakcje biznesowe uzewnętrzniają przed zakończeniem wprowadzone modyfikacje danych. Nie wyklucza anomalii: brudny odczyt, brudny zapis, rozmyty odczyt, skrośny zapis i fantomów. Ponadto transakcja biznesowa zarządzana przez logikę aplikacji nie jest jednostką odtwarzania – nie jest atomowa.

Ponadto rozwiązanie to charakteryzuje się dużym obciążeniem systemu bazy danych operacjami otwierania i zamykania sesji. W praktyce powinno być stosowane jedynie dla pojedynczej interakcji z użytkownikiem - jednej transakcji odczytującej i jednej modyfikującej stan bazy danych.



# Systemowa synchronizacja z kontrolą wersji

Aplikacja biznesowa jest implementowana przez pojedynczą tzw. rozszerzoną sesję (*extended session*). Każda interakcja z bazą danych jest osobną transakcją. Ostatnia transakcja sesji jest odpowiedzialna za zapisanie wszystkich modyfikacji sesji. Kontrola poprawności wersji obiektów jest realizowana automatycznie.

// o1 jest wersją obiektu załadowanego przez poprzednią transakcję

// w tej samej sesji

```
Transaction t = session.beginTransaction();
```

// uzyskaj nowe połączenie z bazą danych z puli aktywnych połączeń

```
o1.setProperty("J23");
```

```
t.commit(); // oddanie połączenia z bazą danych
```

// interakcja z użytkownikiem

```
t = session.beginTransaction(); // ostatnia transakcja w sesji
```

```
o2.setProperty("007");
```

```
session.flush(); // rzucenie do bazy danych zapisów z całej sesji
```

```
t.commit();
```

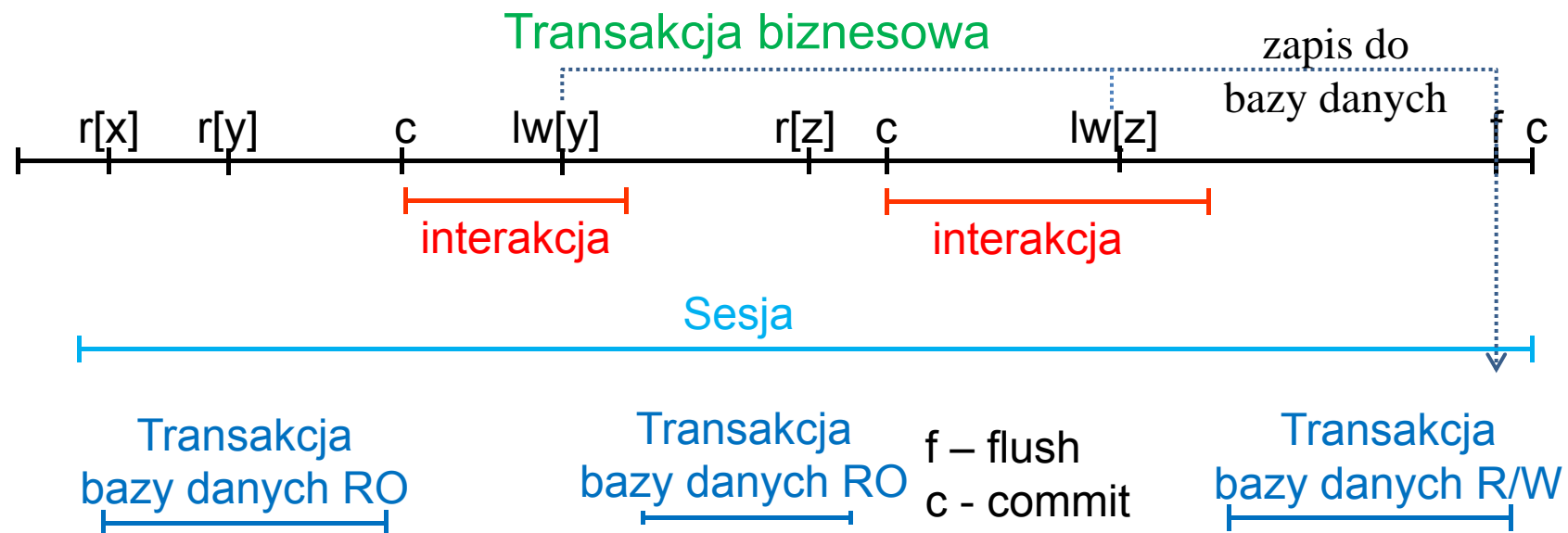
```
session.close();
```



# Systemowa synchronizacja z kontrolą wersji

Rozwiązanie charakteryzuje się niekompletnym poziomem izolacji. Nie wyklucza anomalii: rozmyty odczyt, skróśny zapis i fantomów. Jednak transakcja biznesowa jest atomowa.

Rozwiązanie to charakteryzuje się dużą skalowalnością w dostępie do systemu bazy danych. W danym momencie aktywna jest stała i znacznie mniejsza od liczby współbieżnych użytkowników aplikacji liczba połączeń z bazą danych \*.



\*) W DBMS Oracle maksymalnie  $4 \cdot 10^9$  współbieżnych transakcji

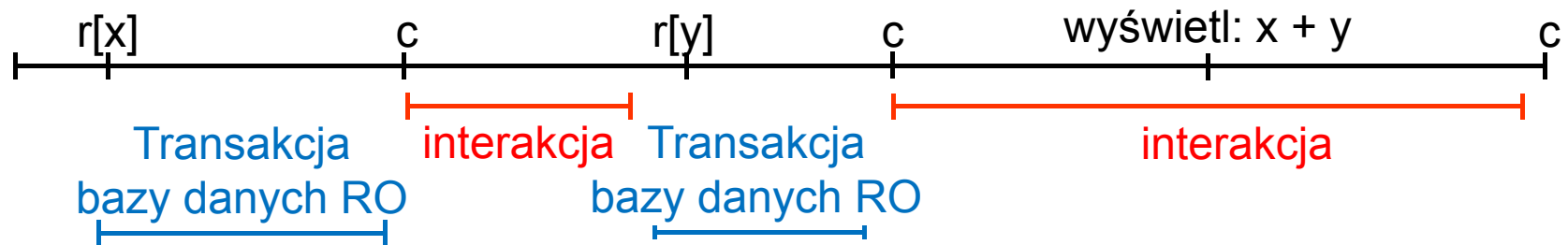
# Własności optymistycznej synchronizacji

Adaptacja optymistycznego protokołu zarządzania współbieżnością transakcji w platformach O/RM zawiera pewne uproszczenia:

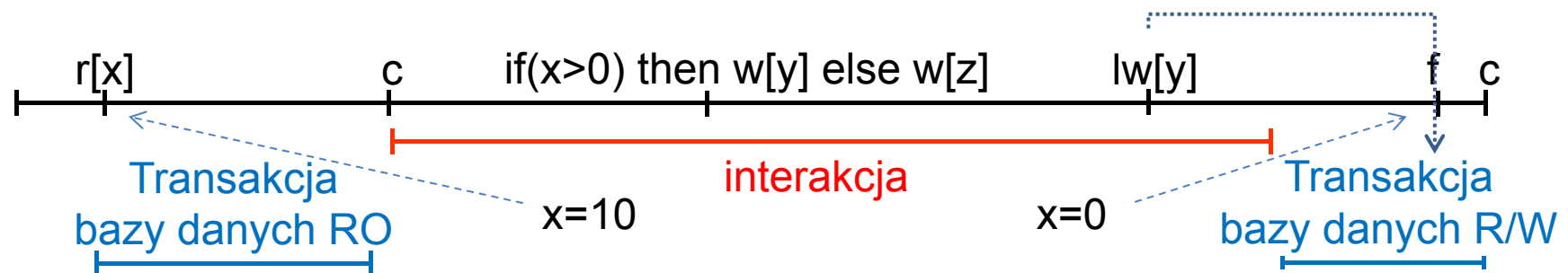
- ▶ Zaimplementowany mechanizm synchronizacji zbudowany jest w oparciu pesymistycznych mechanizmach systemów baz danych. Stąd przebieg transakcji biznesowych wiąże się z zakładaniem blokad do odczytu i zapisu. Blokady do zapisu służą do zapewnienia poprawnej realizacji faz walidacji (działania operacji *flush*) współbieżnych transakcji. Czas utrzymywania blokad jest znacznie krótszy niż w pesymistycznych protokołach synchronizacji. Dzięki temu, ewentualne zawieszenia przetwarzania trwa krócej, a prawdopodobieństwo zakleszczenia jest mniejsze.
- ▶ Kontrola modyfikowanych wersji obiektów ma bardziej ograniczony zasięg niż w protokole Kunga i Robinsona. Warunek weryfikujący możliwość zatwierdzenia transakcji jest zredukowany do sprawdzenia zbioru modyfikowanych, a nie do wszystkich odczytywanych danych: zamiast  $RS_j \cap WS_i \neq \emptyset$ , weryfikowany jest warunek  $WS_j \cap WS_i \neq \emptyset$ . W efekcie protokół dopuszcza większą liczbę anomalii.

# Anomalie współbieżnego wykonania

- ▶ Błędne wyniki przetwarzania wartości odczytanych z bazy danych w dwóch różnych transakcjach.



- ▶ Błędny stan bazy danych będący konsekwencją braku atomowości pary operacji odczytu i zapisu.



- ▶ Skrośny zapis, fantomy

# Wycofanie transakcji

- ▶ W wypadku konfliktów wykrytych w fazie walidacji dla uniknięcia anomalii *lost update* transakcja biznesowa musi zostać wycofana.

```
try {  
    Transaction t = session.beginTransaction();  
    // przetwarzanie danych  
    ...  
    session.flush();  
    t.commit();  
    session.close();  
}  
catch (RuntimeException e) {  
    t.rollback();  
    throw e;  
}  
finally { session.close(); }
```

# Implementacja walidacji transakcji

Do walidacji poprawności wykonania współbieżnych transakcji mogą być używane różne mechanizmy :

- ▶ Znaczniki czasowe/numery wersji – struktura wszystkich danych jest rozszerzona o dodatkowy atrybut systemowy do składowania czasu modyfikacji lub numeru wersji danej zmienianego przez każdą modyfikację. Wadą jest konieczność stosowania tego rozwiązania przez wszystkie transakcje.
- ▶ Weryfikacja wartości danych – polegająca na porównaniu wartości danych z fazy odczytu z wartościami danych w fazie zapisu. Może dotyczyć wszystkich pól obiektu, lub tylko tych które zostały zmienione przez daną transakcję.

# Obiektowe języki zapytań

- ▶ W platformach O/RM stosuje się języki zapytań wzorowane na języku SQL. Są one uboższe od obiektowo relacyjnego języka SQL, np. o polecenia modyfikacji danych lub niemożność wywoływania metod.
- ▶ Przykładowe języki wzorowane na SQL:
  - ▶ H SQL,
  - ▶ EJB SQL,
  - ▶ Linq

## EJB SQL – selekcja obiektów

- ▶ EJB SQL pozwala na selekcję obiektów w bazie danych:

```
select osoba from Osoba as osoba  
where osoba.nazwisko = 'Tarzan'
```

Warunki selekcji nie mogą się odwoływać jedynie do cech strukturalnych, a nie do metod. Ze względu na to, że zapytania są w EJB SQL są zamieniane na zapytania SQL w relacyjnej bazie danych, a te nie mają dostępu do metod składowanych po stronie aplikacji.

## EJB SQL – wyrażenia ścieżkowe

- ▶ W klauzuli WHERE możliwe jest stosowanie wyrażen ścieżkowych nawigujących wzdłuż powiązań między obiektami i w głąb atrybutów złożonych.

```
select o from Osoba as o  
where o.szef.adres.miasto.nazwa = 'Poznań'
```

- ▶ Powyższe zapytanie zostanie zmienione na zbiór zapytań SQL na relacyjnej bazie danych



## EJB SQL – połączenia strukturalne

- ▶ Dostępne są połączenia strukturalne, wewnętrzne i zewnętrzne (jedynie lewostronne), które zwracają kolekcję tablic obiektów.

```
select zespół, pracownik, dzieci  
from Zespół as zespół inner join  
    Pracownik as pracownik left join fetch  
    Dzieci as dzieci
```

- ▶ Opcja połączenia **fetch** powoduje, że dostęp do obiektów połączonych związkiem typu 1:N albo dostęp do obiektów lub wartości składowanych w atrybucie wielowartościowym jest realizowane za pomocą pojedynczego zapytania w bazie danych.

## EJB SQL – połączenia dynamiczne

- ▶ Dostępne są również połączenia dynamiczne konstruowane na podstawie wyrażeń logicznych odwołujących się do prostych wartości obiektów lub do ich tożsamości.

```
select pracownik, szef
from Pracownik as pracownik inner join
    Pracownik as szef
    where pracownik.etat = 'Referent'
// równość wartości
    and pracownik.szef = szef
// tożsamość
```

## EJB SQL – odwołanie do OID

- ▶ W warunkach selekcji możliwe jest odwoływanie się do identyfikatora obiektów:

```
select pracownik  
from Pracownik as pracownik  
where pracownik.etat = 'Referent' and  
pracownik.szef = 11432
```

## EJB SQL – zapytania polimorficzne

- ▶ W EJB SQL umożliwia wykonywanie zapytań polimorficznych na wystąpieniach klas tworzących hierarchię dziedziczenia. Nie ma możliwości korzystania z dynamicznego wiązania.

```
select osoba
```

```
from Osoba as osoba
```

```
// wynik zapytania obejmuje wystąpienia wszystkich
```

```
// specjalizacji klasy Osoba
```

```
select osoba
```

```
from Osoba as osoba.class = Student
```

```
// wynik zapytania obejmuje wystąpienia wszystkich
```

# Podzapytania, sortowanie, grupowanie

- ▶ Grupowanie – wynikiem zapytań zawierających operacje projekcji lub grupowania dla wyznaczania danych statystycznych są wartości lub obiekty nie tworzące logicznego modelu danych. Jednak możliwość przetwarzania dużych zbiorów danych po stronie systemu bazy danych zwiększa radykalnie wydajność przetwarzania.

```
select avg(s.średnia), max(s.średnia), count(s)
from Student s
group by s.rok_studiów
```

# Strojenie wydajności

- ▶ Strojenie przez **wybór wydajnego mapowanie** struktur obiektowych na relacyjne: wybór strategii odwzorowania hierarchii dziedziczenia, związków i cech klas
- ▶ **Opóźnione/natychmiastowe czytanie**: dla dużych obiektów żądanie dostępu do obiektu może być realizowane jako odczyt jedynie podzbioru danych składających się na obiekt. Pozostałe części obiektu będą doczytane w momencie wystąpienia żądania dostępu do nich
- ▶ **Autofetch** – inteligentny wybór momentu odczytu powiązanych danych

# Oferta platform OR/M

## ▶ O/RM dla języka Java

- JDO – Java Data Object
- Enterprise Java Bean
- Java Persistence API – JPA
- Hibernate
- Top Link Oracle

## ▶ O/RM dla platformy .NET

- ADO.NET Entity Framework
- Linq
- Nhibernate
- SODA