SOA w projektowaniu i implementacji oprogramowania

Referat - JPA 2

Rafał Mazur
Dominik Wróbel
Grupa 3a

Spis treści

Adnotacje

@Entity	Adnotacja ta definiuje klasę jako mapowalną do tabeli
@Id	Określa klucz główny często występuje w połączeniu z @GenerateValue
@JoinColumn	Służy do określenia kolumny odpowiedzialnej za połączenie z inną encją
@Table	Pozwala na zmianę nazwy odwzorowanej tabeli(domyślnie jest taka sama jak nazwa klasy)
@Column	Działa podobnie jak @Table z tym że tyczy się ona kolumn
@GenerateValue	Sprawia że kolejne wartości pola będą automatycznie generowane
@ForeignKey	Używane do określenia pola jako klucza obcego

Adnotacje - przykładowe użycie

```
@Entity
@Table(name = "SoldItem")
public class SoldItem {
    @Id
    @Column(name = "SoldItemId")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "StockItemId")
    private StockItem stockItem;
}
```

Relacje

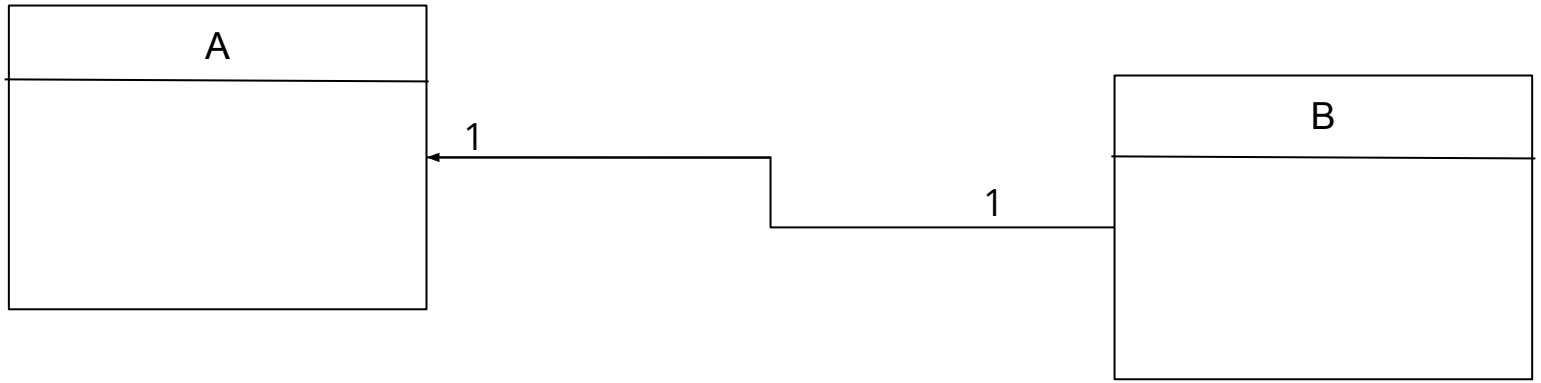
Unidirectional vs bidirectional

W relacji jednokierunkowej (unidirectional) potrafimy dotrzeć tylko od obiektu typu A do obiektu typu B. Jeśli obiekt A jest nadrzędny i obiekt B nigdy nie jest zwracany bez A

W relacji dwukierunkowej (bidirectional) oba obiekty są wzajemnie świadome tego, że są w relacji i z jednego obiektu można przejść do drugiego

@OneToOne - Unidirectional

- Relacja jednokierunkowe jeden do jednego jest modelem związku w którym dostęp do obiektu B może być uzyskany tylko przez obiekt nadrzędny A
- Jako przykład:
  - Dostęp do klasy B może być uzyskany tylko z klasy nadrzędnej A

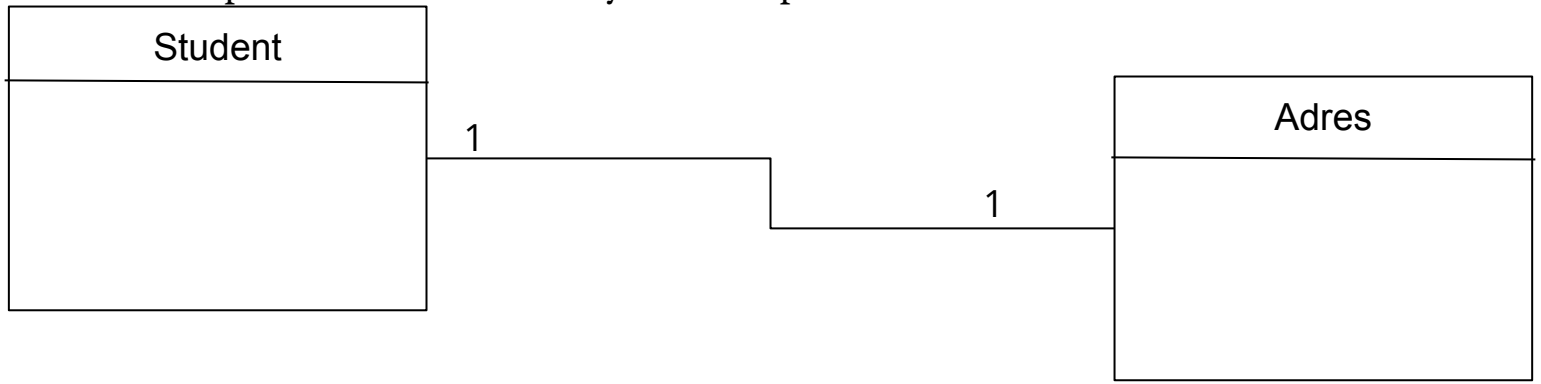


@OneToOne - Unidirectional

```
@Entity
@Table(name="student")
class Student {
    @OneToOne
    @JoinColumn(name="addressId")
    private Adres address;
    // ...
}
@Entity
@Table(name="address")
class Adres {
    @Id
    @Column(name="addressId")
    private long id;
    // ...
}
```

@OneToOne - Bidirectional

- Relacja dwukierunkowa jeden do jednego jest modelem związku w którym obiekt typu A może być powiązany z jednym obiektem typu B, podobnie obiekt typu B może być powiązany z jednym obiektem typu A
- Jako przykład:
  - Student posiada adres pod którym mieszka
  - Adres posiada studenta, który mieszka pod nim

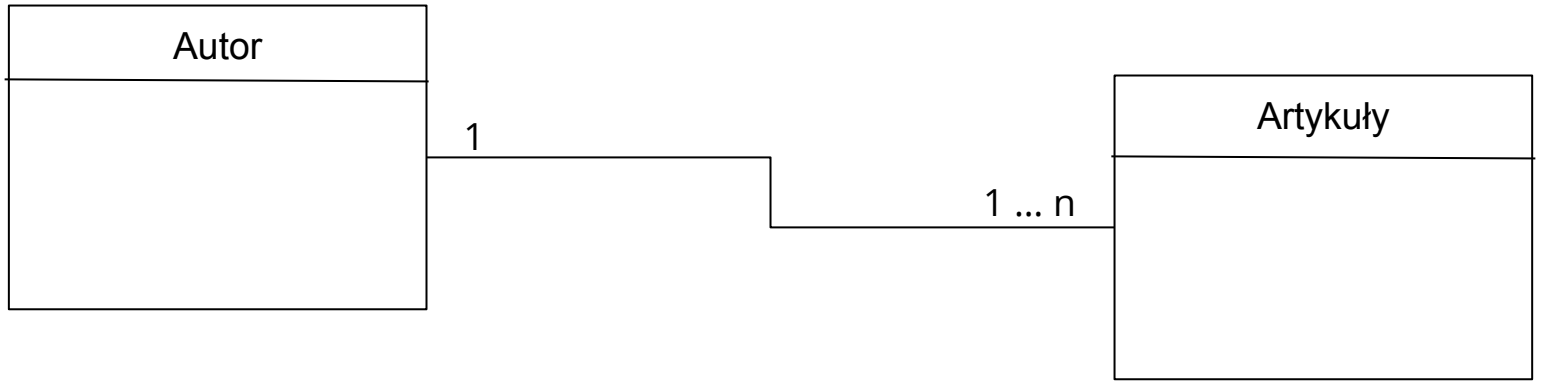


@OneToOne - Bidirectional

```
@Entity
@Table(name="student")
class Student {
    @OneToOne(cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn
    private Adres adres;
    // ...
}
@Entity
@Table(name="adres")
class Adres {
    @OneToOne(mappedBy="student", cascade=CascadeType.ALL)
    private Student student;
    // ...
}
```

@OneToMany

- Relacja jeden-do-wielu jest modelem związku w którym obiekt typu A może być powiązany z wieloma obiektami typu B, a obiekt typu B z tylko jednym typu A
- Jako przykład:
  - Autor mógł napisać wiele artykułów
  - Artykuł może mieć tylko jednego autora

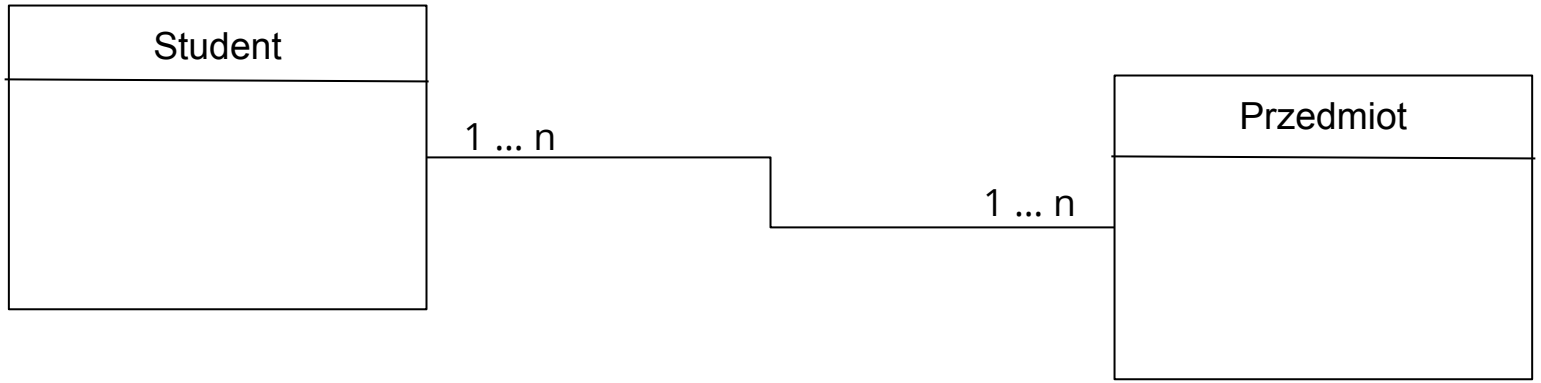


@OneToMany

```
@Entity
@Table(name="author")
class Author {
    @OneToMany
    private List<Article> articles;
    // ...
}
@Entity
@Table(name="article")
class Article {
    @ManyToOne
    @JoinColumn(name=”authorId”)
    private Author author;
    // ...
}
```

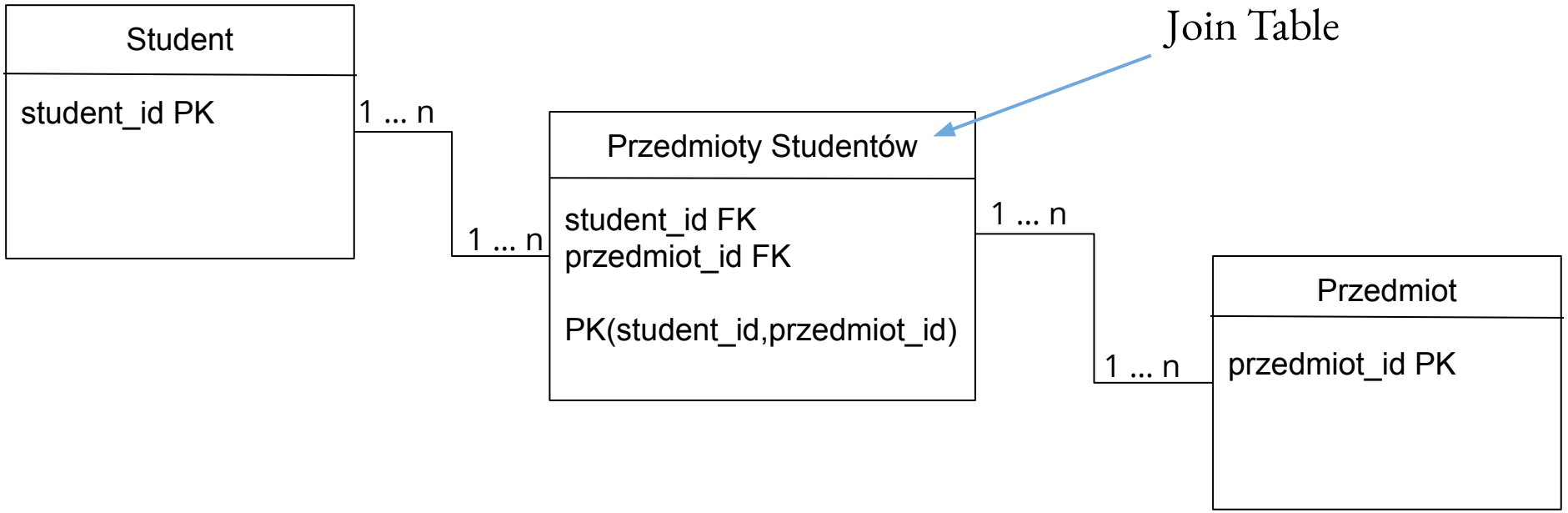
@ManyToMany

- Relacja wiele-do-wielu jest modelem związku w którym obiekt typu A może być powiązany z wieloma obiektami typu B, a obiekt typu B z wieloma obiektami typu A
- Jako przykład:
  - Student ma przypisane wiele przedmiotów na które uczęszcza
  - Przedmiot ma przypisane wielu studentów, którzy nie niego uczęszczają



@JoinTable

- Aby móc utworzyć relacje wiele-do-wielu konieczne jest utworzenie tzw. Join Table, czyli dodatkowej tabeli w bazie danych, która będzie odpowiadała za łączenie obiektów w relacji



Konfiguracja przy użyciu adnotacji - klasa Przedmiot

```
@Entity
@Table(name="przedmioty")
class Przedmiot {
    // ...
    @ManyToMany
    @JoinTable(
            name="przedmioty_studentow",
            joinColumns = @JoinColumn(name="przedmiot_id"),
            inverseJoinColumns = @JoinColumn(name="student_id")
    )
    private List<Student> students;
    // ...
}
```

Konfiguracja przy użyciu adnotacji - klasa Student

```
@Entity
@Table(name="students")
class Student {
    // ...
    @ManyToMany
    @JoinTable(
            name="przedmioty_studentow",
            joinColumns = @JoinColumn(name="student_id"),
            inverseJoinColumns =
@JoinColumn(name="przedmiot_id")
    )
    private List<Przedmiot> przedmioty;
    // ...
}
```

- Dla JoinTable nie tworzymy dodatkowego Entity, tabela generuje się na podstawie konfiguracji

Entity Manager

Entity Manager

- Utworzone przez programistę Entity muszą zostać powiązane z kontekstem, który pozwoli na wykonywanie operacji na bazie danych (Persistence Context).

- W tym kontekście, wszystkie Entity zarządzane są przez komponent nazywany Entity Manager

- Dostęp do komponentu Entity Manager można w aplikacji uzyskać bezpośrednio przez wstrzykiwanie (Dependency Injection):

```
@PersistenceContext
EntityManager entityManager;
```

- A jeśli w projekcie mamy skonfigurowane kilka źródeł danych, to konkretne możemy wybrać dodając:

```
@PersistenceContext(unitName = "primary")
EntityManager entityManager;
```

Database Access Object (DAO)

- W strukturze projektu wydzielamy zazwyczaj dwa oddzielne moduły aby rozdzielić klasy reprezentujące obiekty biznesowe, które będą podlegać mapowaniu do tabel od klas, które realizują konkretne operacje na tych obiektach (zapis, odczyt, itd.)

- Klasy, które realizują konkretne operacje na danych nazywane są Database Access Object (DAO) i w projekcie zazwyczaj konfigurowane są jako bezstanowe EJB. Właśnie do nich wstrzykiwany jest Entity Manager:

```
@Stateless
public class CustomerDAO {

    @PersistenceContext(unitName = "primary")
    EntityManager entityManager;

    public void createCutomer(){
        Customer customer = new Customer();
        customer.setName("Test");
        entityManager.persist(customer);
    }
}
```

Metody Entity Manager

- EntityManager zapewnia kilka bardzo podstawowych metod, które można użyć do wykonania podstawowych operacji na bazie danych, m.in.
  - Zapis - persist(object)

```
entityManager.persist(customer);
```

  - Wyszukiwanie - find(class, primaryKey)

```
Customer c = entityManager.find(Customer.class, 1);
```

  - Usuwanie obiektu - remove(object)

```
entityManager.remove(customer);
```

- Bardziej skomplikowane operacje tworzymy korzystając z jednej z trzech metod omówionych na kolejnych slajdach

Tworzenie zapytań

Jak można wykonywać zapytania ?

- JPA umożliwia wykonywanie zapytań na trzy różne sposoby:

Native SQL

Użycie natywnego kodu SQL dopasowanego do bazy danych

- Konieczne dopasowanie kodu do silnika bazy danych - słaba przenośność rozwiązania

Java persistence Query Language (JPQL)

Użycie specjalnego języka JPQL podobnego do SQL

- Kod niezależny od konkretnej bazy danych - większa przenośność

Criteria API

Zapytania są formowane przez wywoływanie metod z użyciem odpowiednich obiektów

- Nie zawierają żadnych fragmentów tekstu, co może być zaletą w niektórych sytuacjach

Native SQL - Przykłady

- Zapytanie bez mapowania - zwraca Object[] lub List<Object[]>

```
Query q =  entityManager.createNativeQuery("SELECT * FROM
CUSTOMERS WHERE name IS NOT NULL");
List<Object[]> objects = q.getResultList();
```

- Zapytanie z mapowaniem

```
Query q =  entityManager.createNativeQuery("SELECT * FROM
CUSTOMERS WHERE name IS NOT NULL", Customer.class);
List<Customer> customers = q.getResultList();
```

- Zapytanie z parametrami - pozycja parametru liczona od 1, parametry oznaczamy znakiem ‘?’

```
Query q =  entityManager.createNativeQuery("SELECT *
FROM CUSTOMERS WHERE id = ?", Customer.class);
q.setParameter(1, 10); // pozycja, wartość
List<Customer> customers = q.getResultList();
```

JPQL - Przykłady

- Zapytanie z parametrami

```
Query q =  entityManager.createQuery("SELECT *
FROM CUSTOMERS WHERE name LIKE :custName",
Customer.class)
.setParameter("custName", name);
List<Customer> customers = q.getResultList();
```

- Jak widać to rozwiązanie jest dość podobne do NativeSQL - używamy jednak języka JPQL (niezależnego od konkretnego dialektu SQL) i metody *createQuery*.

- Stosując JPQL oprócz *parametrów pozycyjnych* możemy także stosować *parametry nazywane*, w tym przykładzie mamy nazwę parametru “custName”

JPQL, zapytania nazywane - @Named

- Korzystają z JPQL mamy również możliwość tworzenia zapytań nazywanych związanych z danym Entity

```
@Entity
@NamedQueries({
	@NamedQuery(name = "Customer.findAll", query = "SELECT c FROM Customer c"),
	@NamedQuery(name = "Customer.findByName", query = "SELECT c FROM Customer c
WHERE c.name = :name"),
}) public class Customer implements Serializable {
   // . . .
}
```

- Takie zapytania można następnie używać w taki sposób:

```
Query query =
entityManager.createNamedQuery("Customer.findByName");
query.setParameter("name", name);
Customer customer = (Customer) query.getSingleResult();
```

Criteria API - Przykład SELECT ... FROM ... WHERE

- Reprezentuje nieco inne podejście do tworzenie zapytań - zapytania tworzone są przez wywoływanie metod z Criteria API, takie rozwiązanie daje dużą elastyczność względem tworzenia zapytań w czasie wykonywania programu

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

// pass target entity
CriteriaQuery<Customer> criteriaQuery= builder.createQuery(Customer. class);

// set FROM clause
Root<Customer> c = criteriaQuery.from(Customer. class);

// set SELECT and WHERE clauses
criteriaQuery.select(c).where(builder.equal(c.get( "name"), name));

// finally pass the query to entityManager
Customer customer =
entityManager.createQuery(criteriaQuery).getSingleResult();
```

Criteria API - Przykład UPDATE ... WHERE ...

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaUpdate update = cb.createCriteriaUpdate(Customerclass);

Root e = update.from(Customer.class);

update.set("age", 24);

update.where(cb.lessThan(e.get("age"), age));

Query query = entityManager.createQuery(update);

int rowCount = query.executeUpdate();
```

Criteria API - Przykład DELETE ... WHERE ...

```
int age = 24;

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaDelete delete = cb.createCriteriaDelete(Customerclass);

Root e = delete.from(Customer.class);

delete.where(cb.lessThan(e.get("age"), age));

Query query = entityManager.createQuery(delete);

int rowCount = query.executeUpdate();
```

Dziękujemy za
uwagę

Rafał Mazur
Dominik Wróbel
Grupa 3a