Wzorce projektowe

Implementacja pakietu klas realizujących mapowanie O-R dowolnego modelu dziedziny w technologii JavaScript

Dominik Wróbel

Spis treści

1. Opis problemu

Rozważany problem polega na zapewnieniu przyjaznego interfejsu dostępu do relacyjnej bazy danych dla klienta, który wykonuje operacje na relacyjnej bazie danych z poziomu kodu programu.

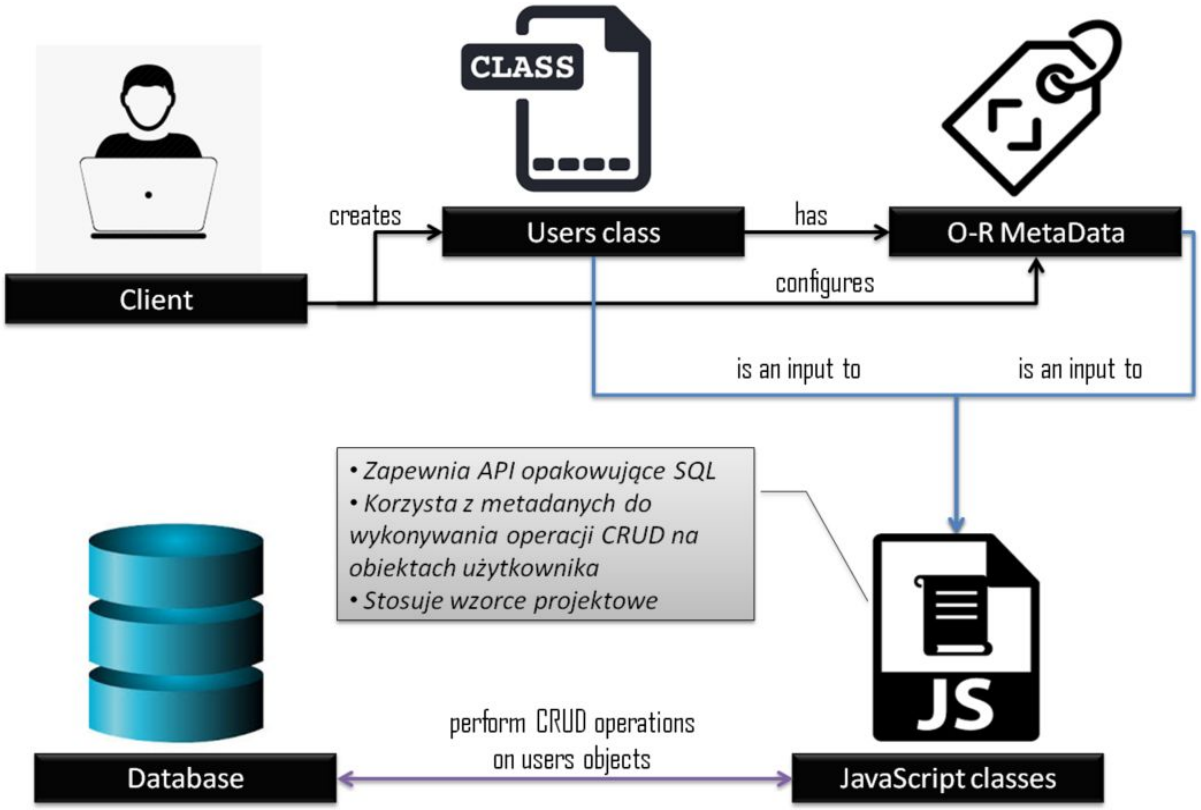
Interfejs taki powinien zapewniać podstawowe funkcje realizowane na bazach danych, a jednocześnie abstrahować od konkretnego modelu danych.

Ponadto interfejs ten powinien agregować w sobie właściwy kod SQL dla zapewnienia wygody i komfortu użytkowania.

Problem ten znany jest powszechnie jako mapowanie obiektowo relacyjne, a w do jego rozwiązania zastosować można szerokie spektrum wzorców projektowych.

2. Zastosowane podejście

W celu rozwiązania problemu zastosowano podejście oparte na metadanych schematycznie przedstawione na Rysunku poniżej.



W podejściu tym użytkownik tworzy własne klasy oraz konfiguruje metadane opisujące jak dana klasa ma być reprezentowana w tabeli bazodanowej.

Podejście takie jest szeroko stosowane do rozwiązywania problemu mapowania

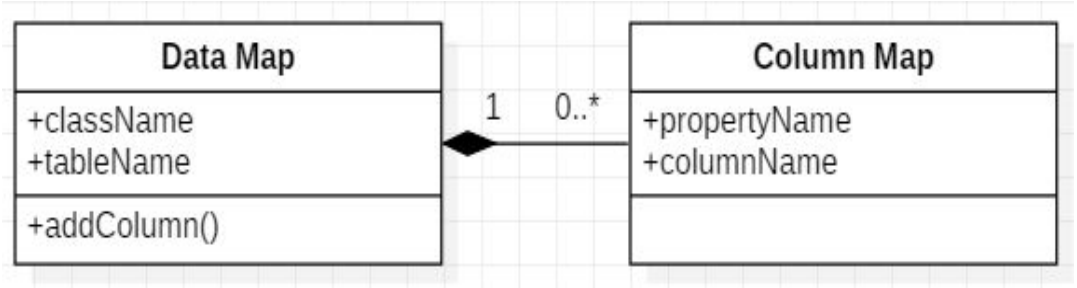
obiektowo-relacyjnego, przykładem jego zastosowania jest biblioteka Hibernate napisana w Javie.

2.1 Metadane

Jednym z kluczowych problemów w stosowaniu takiego podejścia jest podjęcie decyzji dotyczącej formy reprezentacji metadanych. Wśród stosowanych rozwiązań obecne są m.in. konfiguracja metadanych w pliku XML lub przy użyciu adnotacji.

3. Reprezentacja metadanych - wzorzec Metadata Mapping

W projekcie do reprezentacji metadanych zdecydowano się zastosować wzorzec Metadata Mapping. Metadane stanowić będą obiekty zaprojektowane do ich przechowywania.



3.1 Dokumentacja API

- Klasa zdefiniowana przez użytkownika:

```
// User defined class

class Person {
    constructor(id, name, age){
        this.id = id;
        this.name = name;
        this.age = age;
    }
}
```

- Klasy opisujące metadane

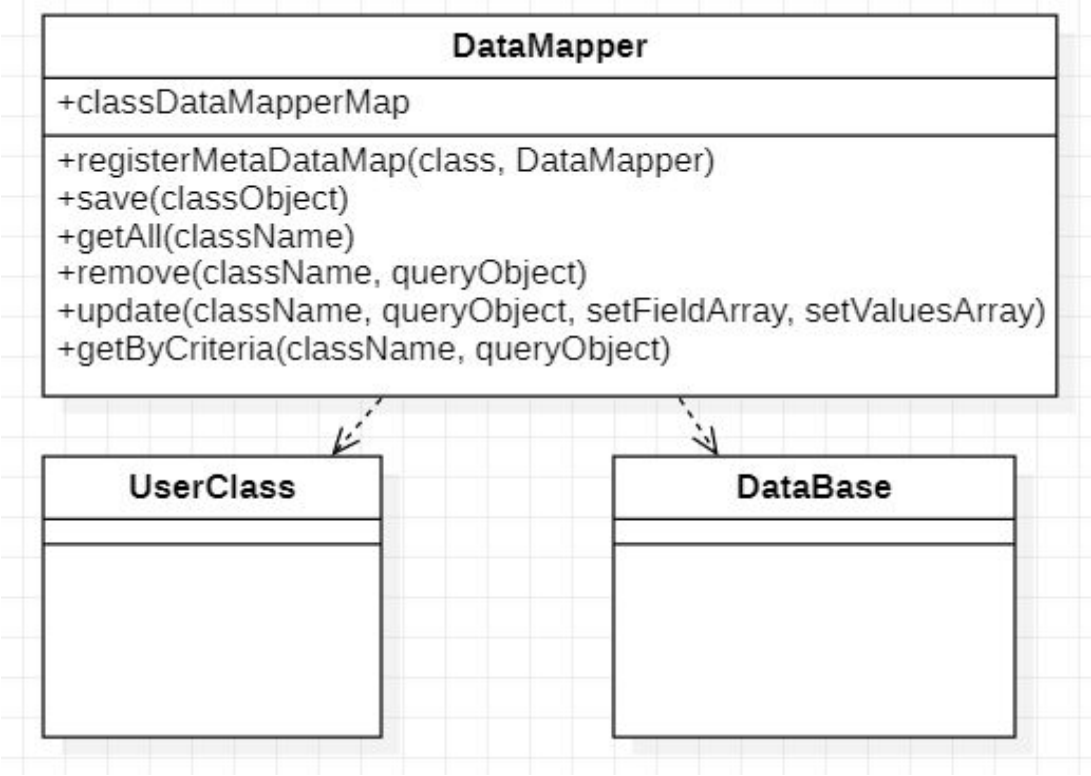
```
// Metadata mapping classes - DataMap
class DataMap {
    constructor(className, tableName){
        this.className = className;
        this.tableName = tableName;
        this.columnsMaps = new Array();
    }

    addColumn(col){
        this.columnsMaps.push(col);
    }
}

// Metadata mapping classes - ColumnMap
class ColumnMap {
    constructor(propertyName, columnName){
        this.propertyName = propertyName;
        this.columnName = columnName
    }
}
```

- Klient konfiguruje metadane dla swojej klasy

```
// class: Person --- maps to database table ---> people
// property: id --- maps to database column ----> id
// property: name --- maps to database column ----> person_name
// property: age --- maps to database column ----> person_age
var person1 = new Person(1, 'John', 26);
var personDataMap = new DataMap('Person', 'people');
personDataMap.addColumn(new ColumnMap('id', 'id'));
personDataMap.addColumn(new ColumnMap('name', 'person_name'));
personDataMap.addColumn(new ColumnMap('age', 'person_age'));
```

4. Transfer danych pomiędzy obiektami a bazą danych - wzorzec Data Mapper

Wzorzec Metadata Mapping odpowiada w projekcie za ‘konfigurację’ mapowania danych. Oprócz tego konieczny jest jeszcze mechanizm, który pozwoli na transfer tych danych pomiędzy bazą a obiektami, ta funkcja zrealizowana zostanie przy użyciu wzorca Data Mapper.



4.1 Dokumentacja API

- Klient, tworzy obiekt DataMapper, rejestruje utworzony obiekt reprezentujący metadane, a następnie wykonuje operacje CRUD.

```
// create new DataMapper
var dataMapper = new DataMapper();
// register MetaDataMapping object
dataMapper.registerMetaDataMap("Person", personDataMap);
```

- save

```
// ^*^*^* save object in DB
dataMapper.save(person1);
```

- getAll i getByCriteria

```
// ^*^*^* get all people from DB
allPeople1 = dataMapper.getAll("Person");
console.log(allPeople1);


// ^*^*^* get specific person from DB
var qo1 = new QueryObject();
qo1.setMetaDataMapping(personDataMap);
qo1.addCriteria(new Criteria('=', 'name', 'John'));
qo1.addCriteria(new Criteria('<', 'age', 50));

var specificPeople1 = dataMapper.getByCriteria("Person", qo1);
console.log(specificPeople1);
```

- update

```
// ^*^*^* update
var qo2 = new QueryObject();
qo2.setMetaDataMapping(personDataMap);
qo2.addCriteria(new Criteria('LIKE', 'name', 'J%'));
dataMapper.update("Person", qo2, ['age'], [20]);

var updateResult = dataMapper.getByCriteria("Person", qo2);
console.log(updateResult);
```

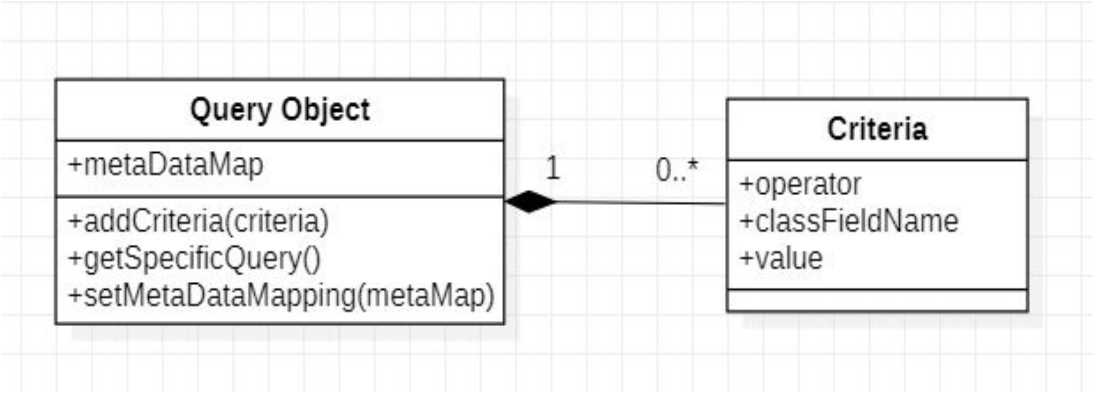
- delete

```
// ^*^*^* delete
var qo3 = new QueryObject();
qo3.setMetaDataMapping(personDataMap);
qo3.addCriteria(new Criteria('LIKE', 'name', 'N%'));
dataMapper.delete("Person", qo3);

var deleteResult = dataMapper.getAll("Person");
console.log(deleteResult);
```

5. Reprezentacja zapytania bazodanowego - wzorzec Query Object

Wzorzec Query Object zastosowano w projekcie w celu ukrycia kodu SQL wykonującego operacje na bazie danych. Pozwala on użytkownikom tworzyć zapytania w prosty i wygodny sposób bez znajomości struktury bazy danych.

Hardcode kodu SQL w klasach nie jest dobrym podejściem ze względu na to, że baza danych może ulec zmianie oraz prowadzi to do duplikacji kodu. Te problemy rozwiązuje wzorzec Query Object.

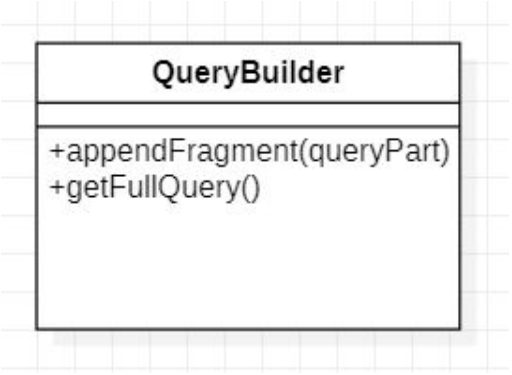


5.1 Dokumentacja API

```
// get all from DB
allPeople1 = dataMapper.getAll("Person");
console.log(allPeople1);

var qo1 = new QueryObject();
qo1.setMetaDataMapping(personDataMap);
qo1.addCriteria(new Criteria('=', 'name', 'John'));
qo1.addCriteria(new Criteria('<', 'age', 50));

// get specific from DB
var specificPeople1 = dataMapper.getByCriteria("Person", qo1);
console.log(specificPeople1);
```

6. Tworzenie zapytania bazodanowego - wzorzec Builder

Wzorzec Builder stosowany jest w projekcie do tworzenia zapytań SQL. Tworzenie zapytania wymaga konkatenacji wielu fragmentów tekstu, co byłoby uciążliwe w przypadku korzystania z z operatora '+'. Lepszym rozwiązaniem jest zastosowanie obiektu, który pozwala na przechowanie i łączeniu wielu fragmentów tekstu, a finalnie zwraca cały 'zbudowany' fragment.

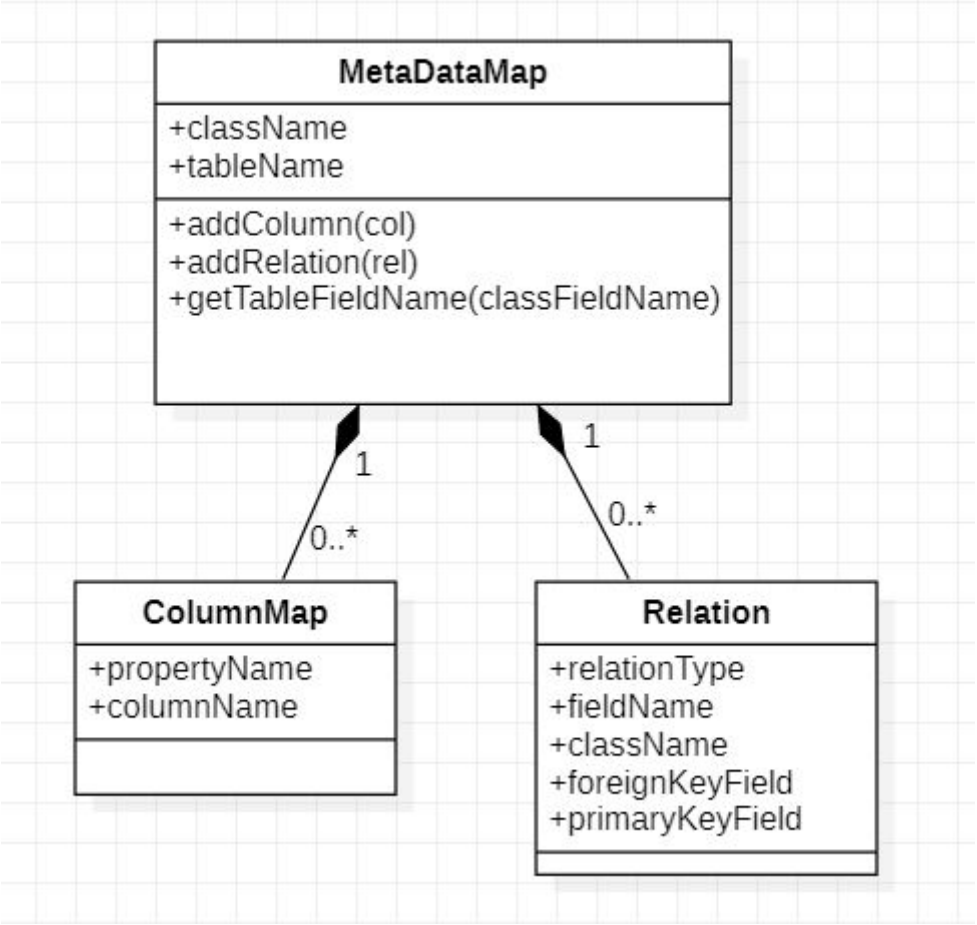


6.1 Dokumentacja API

```
let qb = new QueryBuilder();
qb.appendFragment('INSERT INTO ');
qb.appendFragment(metaDataMapper.tableName);
qb.appendFragment(' (');
for (let i = 0; i < metaDataMapper.columnsMaps.length; i++) {
    qb.appendFragment(metaDataMapper.columnsMaps[i].columnName);
    if(i === metaDataMapper.columnsMaps.length-1){
        qb.appendFragment(')');
    } else {
        qb.appendFragment(', ');
    }
}

qb.appendFragment(' VALUES (');
```

7. Uwzględnienie relacji - OneToOne, OneToMany

Kolejnym krokiem w projekcie systemu jest wprowadzenie obsługi relacji pomiędzy tabelami, ponieważ docelowo chcemy obsługiwać bazy danych SQL składające się z wielu relacji. W tym celu zaczynamy od dodania do metadanych opisu relacji poprzez dodanie nowych składowych typu Relation.



Pozostałe klasy nie ulegają zmianie, wystarczające jest dodanie nowych metadanych opisujących relacje obiektu do innego obiektu.

7.1 Dokumentacja API

- Konfiguracja metadanych, najpierw znana już konfiguracja osobnych klas, a następnie konfiguracja relacji pomiędzy nimi

```
var city7 = new City(4, 7, 'New York', 8000000);
var city8 = new City(4, 8, 'Chicago', 3500000);
var city9 = new City(4, 9, 'San Francisco', 2000000);
var citiesOfUSA = new Array();
citiesOfUSA.push(city7);
citiesOfUSA.push(city8);
citiesOfUSA.push(city9);
var countryUSA = new Country(4, 'USA', citiesOfUSA);

// configure meta data for fields and columns
var countriesDataMap = new MetaDataMap('Country', 'countries');
countriesDataMap.addColumn(new ColumnMap('id', 'country_id'));
countriesDataMap.addColumn(new ColumnMap('name', 'country_name'));


var citiesDataMap = new MetaDataMap('City', 'cities');
citiesDataMap.addColumn(new ColumnMap('countryId', 'country_id'));
citiesDataMap.addColumn(new ColumnMap('cityId', 'city_id'));
citiesDataMap.addColumn(new ColumnMap('name', 'city_name'));
citiesDataMap.addColumn(new ColumnMap('numberOfPeople', 'people'));

// configure meta data for relations
countriesDataMap.addRelation(new Relation('OneToMany', 'cities', 'City', 'countryId', 'id'));
```