Przygotowanie do zajęć:

- Uruchom maszynę wirtualną
- Uruchom bazę danych skryptami z katalogu opt/devel/scripts
- Ściągnij z githuba projekt https://github.com/PJMPR/MPR_repodemo.git
- Zaimportuj go do eclipsa jako projekt maven'owy

1. Przypatrz się klasom CarRepository oraz PersonRepository -powinieneś zauważyć, że kod w obu tych klasach jest niemalże identyczny – i w niektórych miejscach nagina zasady SOLID:
    a. jest w nim umieszczona logika do tworzenia połączenia z bazą danych – dla każdego obiektu repozytorium będzie tworzone odrębne połączenie z bazą danych – Czy nie da się zrobić jednego połączenia dla wielu repozytoriów?
    b. Każde repozytorium odpowiada za mapowanie iteratora ResultSet na obiekty domeny podczas metod wykonujących operację 'SELECT' na bazie – co jeśli logika mapowania odpowiedzi z bazy danych nie jest trywialna? Czy nie lepiej jej wynieść do odrębnej klasy ?
    c. Brak jakiejkolwiek warstwy abstrakcji dla repozytoriów – są to zwykłe klasy, które mają identyczny interfejs, który można wydzielić na zewnątrz – może to się przydac przy testach jednostkowych.
    d. Co się stanie jeśli wystąpi wyjątek podczas tworzenia obiektu klasy repozytoriów? Czy obiekt który otrzymamy będzie dobrze wykonywać swoje zadania? Czy potrzebny jest obiekt który np. Nie ma zainicjowanego połączenia z bazą danych?

2. Refaktoryzacja pól w klasie. Zacznijmy od wydzielenia wspólnych pól w klasie, które niczym nie różnią się od siebie w obu klasach repozytriów, do klasy bazowej 'RepositoryBase'.

```
private Connection connection;
private Statement createTable;
private PreparedStatement insert;
private PreparedStatement selectAll;
private PreparedStatement update;
private PreparedStatement delete;
```

różnica będzie tylko taka, że w klasie bazowej, te pola będą oznaczone słówkiem 'protected'.

```
package jdbcdemo.dao;

import java.sql.PreparedStatement;
import java.sql.Statement;

public abstract class RepositoryBase {

	protected Connection connection;
	protected Statement createTable;
	protected PreparedStatement insert;
	protected PreparedStatement selectAll;
	protected PreparedStatement update;
	protected PreparedStatement delete;
}
```

Każmy teraz naszym repozytoriom dziedziczyć te pola klasie RepositoryBase – teraz możemy usunąć te pola z naszych klas.

```
public class PersonRepository extends RepositoryBase {

	Connection connection;

	private String createTableSql = "CREATE TABLE person("
			+ "id INT GENERATED BY DEFAULT AS IDENTITY,"
			+ "name VARCHAR(20),"
			+ "surname VARCHAR(50),"
			+ "age INT"
			+ ")";

	private String insertSql = "INSERT INTO person(name,surname,age) VALUES
(?,?,?)";
	private String updateSql = "UPDATE person SET (name, surname, age) = (?,?,?)
WHERE id=?";
	private String deleteSql = "DELETE FROM person WHERE id=?";
	private String selectAllSql = "SELECT * FROM person";

	//---> tutaj były te pola

	public PersonRepository(){

		try {
			connection =
DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/workdb");
			createTable = connection.createStatement();
			insert = connection.prepareStatement(insertSql);
			update = connection.prepareStatement(updateSql);
			delete = connection.prepareStatement(deleteSql);
			selectAll = connection.prepareStatement(selectAllSql);
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}
```

3. Refaktoryzacja konstruktora. Zauważ, że kod konstruktorów obu klas jest identyczny – problem jest jednak postaci zapytań SQL - w obu klasach te zapytania się nieznacznie różnią. Zrobimy tak, że skopiujemy ciało konstruktorów do klasy bazowej, ale zamiast stringów damy metody abstrakcyjne, które będą musiały być zaimplementowane w klasach podrzędnych.

```
	protected RepositoryBase() {
		try {
			connection =
DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/workdb");
			createTable = connection.createStatement();
			insert = connection.prepareStatement(insertSql());
			update = connection.prepareStatement(updateSql());
			delete = connection.prepareStatement(deleteSql());
			selectAll = connection.prepareStatement(selectAllSql());
		} catch (SQLException e) {
			e.printStackTrace();
		}
```

```
    }

    protected abstract String selectAllSql();
    protected abstract String deleteSql();
    protected abstract String updateSql();
```
```
    protected abstract String insertSql();
```

Zmiany w klasie PersonRepository:

- Konstruktor

```
    public PersonRepository(){
        super();
    }
```

- Zaimplementowane metody z klasy podrzednej

```
    @Override
    protected String selectAllSql() {
        return "SELECT * FROM person";
    }

    @Override
    protected String deleteSql() {
        return "DELETE FROM person WHERE id=?";
    }

    @Override
    protected String updateSql() {
        return "UPDATE person SET (name, surname, age) = (?,?,?) WHERE
id=?";
    }

    @Override
    protected String insertSql() {
        return "INSERT INTO person(name,surname,age) VALUES (?,?,?)";
    }
```

Dotychczasowe pola przechowujące te operacje, insertSql, udateSql, deleteSql, selectAllSql – można śmiało usunąć.

4. Refaktoryzacja metody do tworzenia tabeli. Kolejną metodą zawierającą identyczny kod jest metoda createTable – różni się ona tylko nazwą tabeli którą próbujemy znaleźć – tutaj podobnie jak w przypadku konstruktorów wydzielimy ten kawałek kodu do metody abstrakcyjnej – nazwę tabeli podamy już w konkretnej klasi która będzie dziedziczyć po klasie RepositoryBase
   Przenieśmy metodę createTable do klasy bazowej i miejsce w którym jest podana nazwa tabelki zastąpimy metodą abstrakcyjną, podobnie zrobimy w miejscu gdzie potrzebny jest string z SQL'em tworzącym tabelkę.

```
public void createTable(){
try {
	ResultSet rs = connection.getMetaData().getTables(null, null, null, null);
	boolean tableExists = false;
	while(rs.next()){
	 if(rs.getString("TABLE_NAME").equalsIgnoreCase(tableName())){
				tableExists=true;
				break;
			}
		}
	 if(!tableExists)
		createTable.executeUpdate(createTableSql());
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}
```

```
protected abstract String tableName();
protected abstract String createTableSql();
```

Teraz metodę createTable() można usunąć z klasy PersonRepository, ale musimy doimplementować kolejne dwie metody.

```
	@Override
	protected String tableName() {
		return "person";
	}

	@Override
	protected String createTableSql() {
		return "CREATE TABLE person("
				+ "id INT GENERATED BY DEFAULT AS IDENTITY,"
				+ "name VARCHAR(20),"
				+ "surname VARCHAR(50),"
				+ "age INT"
				+ ")";
	}
```

Pole createTableSql w klasie person można już usunąć.

5. Refaktoryzacja – wprowadzenie parametru typu. Kolejną metodą która wygląda na łatwą do wydzielnia do klasy bazowej jest metoda delete() – aczkolwiek w jednej klase repozytorium przyjmuje ona obiekt typu Person a w drugiej obiekt typu car.

```
	public void delete(Person person) {

		try{
			delete.setInt(1, person.getId());
			delete.executeUpdate();
		}catch(SQLException ex){
			ex.printStackTrace();
		}
	}
```

Zwróćmy uwagę, że w obu przypadkach tak naprawdę jedyną rzeczą którą potzebujemy od klasy Person jak i Car jest metoda getId() – wydzielmy zatem nowy interfejs dla obiektów domeny

IhaveId z jedną metodą – ogólnie wszystkie obiekty których dane będą przechowywane w bazie danych będą posiadały pole id.

Do pakiety domain dodaj interfejs IhaveId

```
package jdbcdemo.domain;

public interface IHaveId {
	public int getId();
}
```

A teraz zaznaczmy, że klasy domeny będą go implementować

```
public class Person implements IHaveId{

public class Car implements IHaveId{
```

Teraz w klasie bazowej możemy wprowadzić parametr typu (jeśli jest to pierwszy raz jak spotykasz sie z tym pojęciem podpytaj prowadzącego zajęcia szczegóły)

Zmień sygnaturę klasy bazowej na

```
public abstract class RepositoryBase<TEntity extends IHaveId> {
```

I popraw sygnaturę w klasach pochodnych, wprowadzając typ encji jak parametr do klasy RepositoryBase

```
public class PersonRepository extends RepositoryBase<Person>
```

Teraz spokojnie możemy spokojnie metodę delete() przenieść do klasy bazowej, parametryzując jej argumenty parametrem typu

```
public abstract class RepositoryBase<TEntity extends IHaveId> {

	public void delete(TEntity entity) {

		try{
			delete.setInt(1, entity.getId());
			delete.executeUpdate();
		}catch(SQLException ex){
			ex.printStackTrace();
		}
	}
```

Zadanie do samodzielnego wykonania
Spróbuj teraz samemu wynieść metody add() oraz update() do klasy bazowej stosując parametr typu oraz metody abstrakcyjne, zastępując kod który różni się w klasach PersonRepository oraz CarRepository.

6. Refaktoryzacja – Dependency Inversion. Ostatnią metodą do wyrzucenia jest metoda getAll(), jednakże posiada ona logikę do mapowania wyniku z baazy danych na obiekt domeny, który chcielibyśmy wynieść do odrębnej klasy, której obiekty będą dostarczać potrzebną nam logikę do mapowania obiektów.
   Dodajmy nowy pakiet 'jdbcdemo.dao.mappers', a w nim interfejs ResultSetMapper

```
package jdbcdemo.dao.mappers;

import java.sql.ResultSet;
import java.sql.SQLException;

public interface ResultSetMapper<TEntity> {

	public TEntity map(ResultSet rs) throws SQLException;
}
```

   Zauważ, że ten interfejs też zależy od parametru typu – jest typem uogólnionym. Teraz zaimplementujmy mappera dla klasy Person

```
package jdbcdemo.dao.mappers;

import java.sql.ResultSet;
import java.sql.SQLException;

import jdbcdemo.domain.Person;

public class PersonResultMapper implements ResultSetMapper<Person>{

	public Person map(ResultSet rs) throws SQLException {
		Person p = new Person();
		p.setId(rs.getInt("id"));
		p.setName(rs.getString("name"));
		p.setSurname(rs.getString("surname"));
		p.setAge(rs.getInt("age"));
		return p;
	}
}
```

Wprowadźmy go jako zależność do klasy RepositoryBase – ten obiekt będzie dostarczany przez konstruktor

```
	ResultSetMapper<TEntity> mapper;

	protected RepositoryBase(ResultSetMapper<TEntity> mapper) {
		this.mapper = mapper;
		try {
			connection =
DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/workdb");
			createTable = connection.createStatement();
			insert = connection.prepareStatement(insertSql());
			update = connection.prepareStatement(updateSql());
			delete = connection.prepareStatement(deleteSql());
```

```
                selectAll = connection.prepareStatement(selectAllSql());
            } catch (SQLException e) {
                e.printStackTrace();
            }
            }
```

Poprawmy teraz konstruktory klas potomnych

```
    public PersonRepository(ResultSetMapper<Person> mapper){
        super(mapper);
        }
```

Przenieśmy metodę getAll() do klasy bazowej i w miejscach gdzie mapowany jest ResultSet na obiekt encjii, wykorzystajmy mappera.

```
    public List<TEntity> getAll(){
        List<TEntity> result = new ArrayList<TEntity>();
        try {
            ResultSet rs = selectAll.executeQuery();
            while(rs.next()){
                result.add(mapper.map(rs));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result;
    }
```

W ten sposób pozbyliśmy się całego kodu który był powtarzany z klasach repozytoriów, poprzez przeniesienie go do klasy bazowej.

7. Refaktoryzacja – wprowadzenie warstwy abstrakcji dla repozytoriów, wykorzystannie ardzo uproszczonego wzroca Factory do dostarczania obiektów repozytoriów. Jako, że w chwili obecnej klasy pochodne nie posiadają żadnych metod publicznych – zostały one wszystkie wydzielone do klasy bazowej – możemy z klasy bazowej wydzielić wspólny interfejs, dla wszystkich repozytoriów

```
package jdbcdemo.dao;

import java.util.List;

import jdbcdemo.domain.IHaveId;

public interface Repository<TEntity extends IHaveId> {

    public void delete(TEntity entity);

    public List<TEntity> getAll();

    public void add(TEntity person);

    public void update(TEntity person);

    public void createTable();
```

```
}
```

I kazać go implementować jedynie klasie bazowej (pozostałe klasy będą go implementować poprzez mechanizm dziedziczenia)

```
public abstract class RepositoryBase<TEntity extends IHaveId> implements
Repository<TEntity>
```

I ostatnią rzeczą która nam pozostała to wykorzystanie interfejsu repository w prostej fabryce repozytoriów.

Dodajmy klasę RepositoryCatalog która poprzez metode people()bedzie dostarczać nam obiekt repozytorium dla osób

```
package jdbcdemo.dao;

import jdbcdemo.dao.mappers.PersonResultMapper;
import jdbcdemo.domain.Person;

public class RepositoryCatalog {

	public Repository<Person> people(){
		return new PersonRepository(new PersonResultMapper());
	}
}
```

Zadanie do samodzielngo wykonania

Wykonaj wszystkie kroki refaktoryzacji dla klasy CarRepository.

8. Refaktoryzacja – poprawnie zdeiniowany obiekt. Ostatnią rzecz na którą warto zwrócić uwagę jest klauzula try-catch w konstrktorze klasy bazowej – jeśli wystąpi tam jakikolwiek wyjątek typu SQLException, to zostanie on przechwycony co zaskutkuje utworzeniem obiektu repozytorium z źle zainicjowanymi polami – a to spowoduje że ten obiekt bedzie bezużyteczny. Lepiej będzie jeśli pozbędziemy się klauzuli try-catch i zadeklarujemy, że w konstruktorze może wstąpić wyjątek, który będzie musiał być obsłużony później.

```
	protected RepositoryBase(ResultSetMapper<TEntity> mapper)
		throws SQLException {
	this.mapper = mapper;
connection = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/workdb");
	createTable = connection.createStatement();
	insert = connection.prepareStatement(insertSql());
	update = connection.prepareStatement(updateSql());
	delete = connection.prepareStatement(deleteSql());
	selectAll = connection.prepareStatement(selectAllSql());
	}
```

Kolejną rzeczą która jest tutaj mankamentem jest to, że dla każdego obiektu repozzytorium, połączenie będzie tworzone za każdym razem – lepiej byłoby gdyby połączenie było przekazywane do obiektu z zewnątrz

```
protected RepositoryBase(Connection connection, ResultSetMapper<TEntity> mapper)
throws SQLException {
		this.mapper = mapper;
		this.connection = connection;
		createTable = connection.createStatement();
		insert = connection.prepareStatement(insertSql());
		update = connection.prepareStatement(updateSql());
		delete = connection.prepareStatement(deleteSql());
		selectAll = connection.prepareStatement(selectAllSql());
	}
```

Trzeba będzie to także poprawić w klasach pochodnych

```
	public PersonRepository(Connection connection, ResultSetMapper<Person>
mapper) throws SQLException{
		super(connection, mapper);
	}
```

Teraz poprawmy kod w klasie RepositoryCatalog, tak aby obsługiwał wyjątek i przekazywał połączenie do nowoutworzonych obiektów repozytoriów.

```
public class RepositoryCatalog {
    Connection connection;

public RepositoryCatalog(Connection connection) {
		this.connection = connection;
	}

public Repository<Person> people(){
		try {
			return new PersonRepository(connection, new
PersonResultMapper());
		} catch (SQLException e) {
			e.printStackTrace();
			return null;
		}
	}
}
```

Zadanie do samodzielnego wkonania

Sprawdź, czy nie ma żadnych błędów kompilacji. Utwórz połączenie w metodzie main. Wykorzystaj obiekt katalogu repozytoriów do operacji CRUD na bazie danych.