Przygotowanie do zajęć:

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

1. Tym razem spróbujemy oprogramować transakcje na bazie danych. Transakcja bazodanowa to uporządkowany zbiór operacji bazodanowych wykonywanych jedna po drugiej gdzie wszystkie operacje muszą być wykonane, lub żadna -w razie pojawienia się błędu wszystkie wprowadzane zmiany zostają wycofane. Sterownik JDBC dostarcza nam możliwość wysłania do serwera bazy danych zestawu operacji w postaci transakcji – Spróbujemy zaimplementować takie rozwiąanie wykorzystując jedną z wielu implementacji wzorca projektowego 'Unit Of Work' (jednostka pracy).
2. Do projektu dodaj nowy pakiet 'jdbcdemo.dao.uow'. Zacznijmy od wprowadzenia 4 nowych typów do naszego rozwiązania:
    - UnitOfWork – interfejs z metodami do oznaczania stanu encji, oraz zatwierdzaniem/wycofywaniem zmian na bazie danych

```
public interface UnitOfWork {
	public void markAsNew(Entity entity);
	public void markAsDeleted(Entity entity);
	public void markAsChanged(Entity entity);
	public void saveChanges();
	public void rollback();
}
```

    - UnitOfWorkRepository – interfejs z wydzielonymi metodami do wykonowania operacji na danej encji

```
public interface UnitOfWorkRepository {

	public void persistAdd(Entity entity);
	public void persistUpdate(Entity entity);
	public void persistDelete(Entity entity);

}
```

    - EntityState – enum który reprezentuje aktualny stan encji: New, Changed, UnChanged, Deleted, Unknown

```
public enum EntityState {
	New, Deleted, Changed, UnChanged, Unknown
}
```

    - Entity – klasa która będzie przechowywać obiekt oraz stan encji, na której ma być wykonywana operacja bazodanowa oraz repozytorium które będzie odpowiedzialne za wykonanie tejże operacji

```
public class Entity {

	private IHaveId entity;
	private UnitOfWorkRepository repository;
	private EntityState state;

	public IHaveId getEntity() {
		return entity;
	}
	public void setEntity(IHaveId entity) {
		this.entity = entity;
	}
	public UnitOfWorkRepository getRepository() {
		return repository;
	}
	public void setRepository(UnitOfWorkRepository repository) {
		this.repository = repository;
	}
	public EntityState getState() {
		return state;
	}
	public void setState(EntityState state) {
		this.state = state;
	}

}
```

3. Modyfikacja klasy ‘RepositoryBase’. Zaczniemy od implementacji interfejsu UnitOfWorkRepository – ze względu na to, że metody tego interfejsu zostały już prawie zaimplementowane w klasie RepositoryBase. Niech ‘RepositoryBase’ implementuje interfejs ‘UnitOfWorkRepository’

   ```
   public abstract class RepositoryBase<TEntity extends IHaveId> implements
   Repository<TEntity>, UnitOfWorkRepository {…
   ```

   zmień nazwę metody ‘add’ na ‘persistAdd’, oraz zmieńmy typ argumentu Tentity na Entity (tak aby sygnatura metody zgadzała się z sygnaturą metody z interfejsu UnitOfWorkRepository)

```
	public void persistAdd(Entity entity){
		try{
			setupInsert(entity);
			insert.executeUpdate();
		}catch(SQLException ex){
			ex.printStackTrace();
		}
	}
```

Pojawił się problem kompilacji – metoda setupInsert w argumencie przyjmuje typ generyczny, a nie typ encji – poprawimy go wydobywając obiekt encji oraz rzutowaniem go na typ generyczny

```
public void persistAdd(Entity entity){
		try{
			setupInsert((TEntity)entity.getEntity());
			insert.executeUpdate();
		}catch(SQLException ex){
			ex.printStackTrace();
		}
	}
```

Pojawił się teraz kolejny problem – klasa RepositoryBase teraz nie implementuje intefejsu ‘Repository’ – poprawmy to wprowadzajac jeszcze raz metodę ‘add’, ale na razie pozostawimy ją bez implementacji

```
	public void add(TEntity entity) {

	}
```

Do implementacji tej metody wrócimy później.

Zadanie do samodzielnego wykonania. W podobny sposób zmodyfikuj metody ‘update’ oraz ‘delete’, tak aby klasa RepositoryBase implementowała oba interfejsy

4. Implementacja metody ‘persistChange‘ w klasie ‘Entity’. Klasa Entity posiada wszystkie informacje potrzebne do wykonania operacji bazodanowej – w związku z tym wywołanie odpowiedniej metody przydzielonego repozytorium, w zależnościu od stanu obiektu encji w jakim się znajduje, jest tutaj najprostsze. Napiszemy tutaj metodę która przyda nam się później w implementacji interfejsu UnitOfWork.

```
	public void persistChange() {

		switch(state) {
		case Changed:
			repository.persistUpdate(this);
			break;
		case Deleted:
			repository.persistDelete(this);
			break;
		case New:
			repository.persistDelete(this);
			break;
		default:
			break;
		}
	}
```

5. Implementacja interfejsu UnitOfWork. Dodaj do projektu nową klasę ‘JdbcUnitOfWork’ i zadeklaruj implementowanie interfejsu UnitOfWork

```
package jdbcdemo.dao.uow;

public class JdbcUnitOfWork implements UnitOfWork{

	public void markAsNew(Entity entity) {
		// TODO Auto-generated method stub

	}

	public void markAsDeleted(Entity entity) {
		// TODO Auto-generated method stub

	}

	public void markAsChanged(Entity entity) {
		// TODO Auto-generated method stub

	}

	public void saveChanges() {
		// TODO Auto-generated method stub

	}

	public void rollback() {
		// TODO Auto-generated method stub

	}
}
```

Jdbc dostarczając obiekt połączenia do bazy danych domyślnie wyłącza transakcyjność operacji – można to włączyć korzystając metody setAutoCommit. Nasza implementacja będzie potrzebowała obiektu Connection aby mogła poprawnie wykonywać swój kod – wysyłać zestaw operacji bazodanowych w transakcyjny sposób. Do klasy dodjmy pole typu Connection, a przez konstruktor je ustawimy oraz włączym transakcyjność ustawiając flagę autoCommit na false.

```
	private Connection connection;

	public JdbcUnitOfWork(Connection connection) throws SQLException {
		this.connection =  connection;
		this.connection.setAutoCommit(false);
	}
```

Do klasy dodajmy teraz pole ArrayList'y – która w odpowiedniej kolejności będzie przechowywać obiekty encji.

```
	private Connection connection;
	private List<Entity> entities = new ArrayList<Entity>();

	public JdbcUnitOfWork(Connection connection) throws SQLException {
		this.connection =  connection;
		this.connection.setAutoCommit(false);
	}
```

Teraz zaimplementujmy metody markAsNew, markAsDeleted, markAsChanged - to będą proste implementacje polegające na zmianie stanu encji oraz umieszczeniu jej na liście encji

```
	public void markAsNew(Entity entity) {
		entity.setState(EntityState.New);
		entities.add(entity);
	}

	public void markAsDeleted(Entity entity) {
		entity.setState(EntityState.Deleted);
		entities.add(entity);
	}

	public void markAsChanged(Entity entity) {
		entity.setState(EntityState.Changed);
		entities.add(entity);
	}
```

Teraz zaimplementujmy metodę saveChanges, która przejrzy wszystkie obiekty encji i wywoła z każdej metodę persistChange, potem obiekt connection wyśle wszystkie zadeklarowane operacje w postaci transakcji do bazy danych po czym nastąpi usunięcie elementów z listy encji

```
	public void saveChanges() {

		for(Entity entity : entities)
			entity.persistChange();
		try {
			connection.commit();
			entities.clear();
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}
```

No i na koniec została nam metoda rollback, która jedynie wyczyści kolekcję encji

```
	public void rollback() {
		entities.clear();
	}
```

6. Implementacja metod add, delete, update w klasie RepositoryBase. Wróćmy teraz do implementacji interfejsu Repository – jak się pewnie domyślasz teraz te metody będą wywoływały metody markAsNew, markAsDeleted, markAsChanged z obiektu typu UnitOfWork. Dodajmy nowe pole w klasie RepositoryBase typu UnitOfWork i zainicjujmy je przez konstruktor

```
	ResultSetMapper<TEntity> mapper;
	UnitOfWork uow;
	protected RepositoryBase(Connection connection, ResultSetMapper<TEntity>
mapper, UnitOfWork uow) throws SQLException {
		this.mapper = mapper;
		this.connection = connection;
		this.uow = uow;
		createTable = connection.createStatement();
		insert = connection.prepareStatement(insertSql());
		update = connection.prepareStatement(updateSql());
```

```
            delete = connection.prepareStatement(deleteSql());
            selectAll = connection.prepareStatement(selectAllSql());
      }
```

Zadanie do samodzielnego wykonania. Popraw błędy kompilacji wynikające z dodania nowego argumentu w konstruktorze klasy bazowej 'RepositoryBase'.

Teraz zaimplementujmy w.w. metody

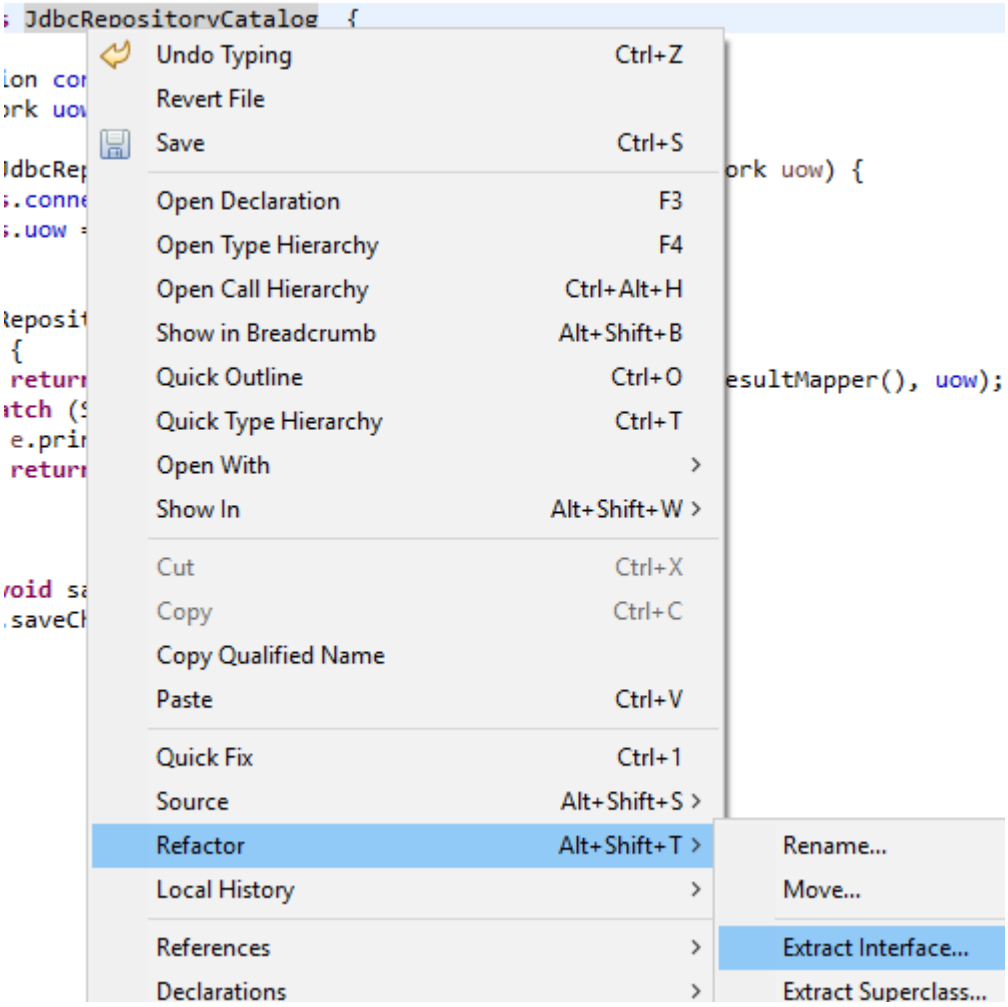
```
      public void add(TEntity entity) {
            Entity ent = new Entity();
            ent.setEntity(entity);
            uow.markAsNew(ent);
      }

      public void delete(TEntity entity) {
            Entity ent = new Entity();
            ent.setEntity(entity);
            uow.markAsDeleted(ent);
      }

      public void update(TEntity entity) {
            Entity ent = new Entity();
            ent.setEntity(entity);
            uow.markAsChanged(ent);
      }
```

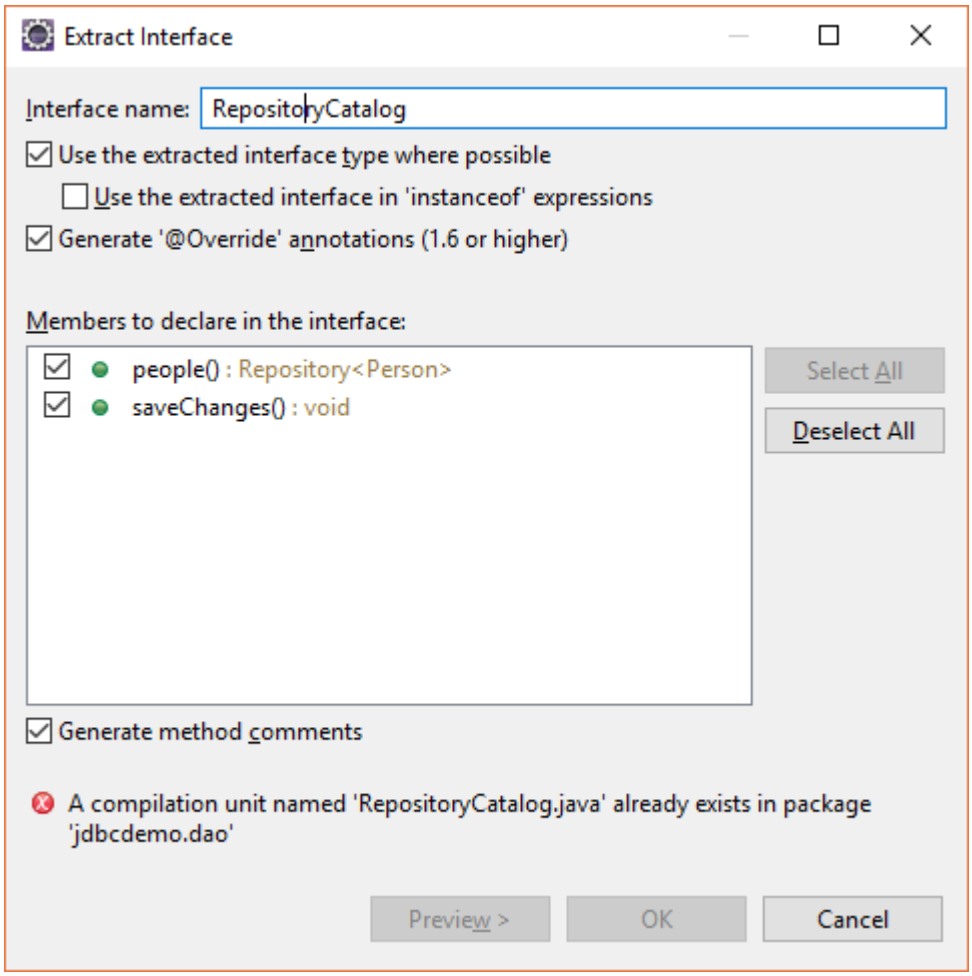
7. Ostatnie kosmetyczne poprawki. Zmieńmy nazwę klasy RepositoryCatalog na JdbcRepositoryCatalog (shift+alt+r) i dodajmy metodę saveChanges, dzięki której będziemy mogli „comitować" zmianyna bazie.

```
public class JdbcRepositoryCatalog  {

      Connection connection;
      UnitOfWork uow;

      public JdbcRepositoryCatalog(Connection connection, UnitOfWork uow) {
            this.connection = connection;
            this.uow = uow;
      }

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

      public void saveChanges() {
            uow.saveChanges();
      }
```

```
}
```

Teraz wydzielmy z klasy interfejs RepositoryCataog






```
public interface RepositoryCatalog {

	Repository<Person> people();

	void saveChanges();
}
```

Ponieważ dostarczenie poprawnie zbudowanego obiektu katalogu repozytoriów, a tym somamym obiektów repozytoriów staje się coraz bardziej skomplikowaną operację (podanie urla do bazy, przygotowanie połączenia, przygotowanie unit of work, przygotowanie katalogu...), skorzystamy ze prostej implementacji wzorca Fabryka aby ukryć tę skomplikowaną logikę, a tym samym otrzymać dobrze zbudowany obiekt do operacji CRUD.

Dodajmy interfejs DbCatalogFactory

```
	package jdbcdemo.dao;

	public interface DbCatalogFactory {

	public RepositoryCatalog HsqlDbWorkDb();
	}
```

Oraz implementację

```
package jdbcdemo.dao;

import java.sql.Connection;
import java.sql.DriverManager;

import jdbcdemo.dao.uow.JdbcUnitOfWork;
import jdbcdemo.dao.uow.UnitOfWork;

public class JdbcCatalogFactory implements DbCatalogFactory{

	public RepositoryCatalog HsqlDbWorkDb() {
		String url = "jdbc:hsqldb:hsql://localhost/workdb";
		try {
		Connection connection = DriverManager.getConnection(url);
		UnitOfWork uow = new JdbcUnitOfWork(connection);
		return new JdbcRepositoryCatalog(connection, uow);
		}
		catch(Exception ex) {
			ex.printStackTrace();
			return null;
		}
	}
}
```

Dzięki temu zabiegowi użycie naszej warstwy bazodanowej w systemie może wyglądać następująco:

```
    public static void main( String[] args )
    {

       Person jan = new Person();
       RepositoryCatalog workdb = new JdbcCatalogFactory().HsqlDbWorkDb();
       workdb.people().add(jan);
    }
```