howest
hogeschool

Object Oriented Architectures and Secure Development

Java's N-Tier: More Tiers, Less Tears!

Arne Debou
Mattias De Wael
Frédéric Vlummens

Divide et impera

Locale

HTML/CSS

JDBC

Authentication

Middleware

Perpared Statement

User Interface

Data Persistence

Business Logic

MySQL

Service

FXML

howest hogeschool

Impera

?	?	?
Locale	Authentication	JDBC
HTML/CSS	Middleware	Prepared Statement
User Interface	Business Logic	Data Persistence
FXML	Service	MySQL

Developing an actual program using database storage

- In a real-life situation, we need to access the database multiple times.
- List all products, add or update a product based on user input, …
- Two ways of developing the program:
  1. Repeating the database code in each method of our Program class.
     ⇔ Don’t Repeat Yourself!
  2. Isolating the database code in a separate layer and calling it from our Program class.

  This is called multi-tier architecture… (or n tier, or layered architecture, …)

The n-tier architecture

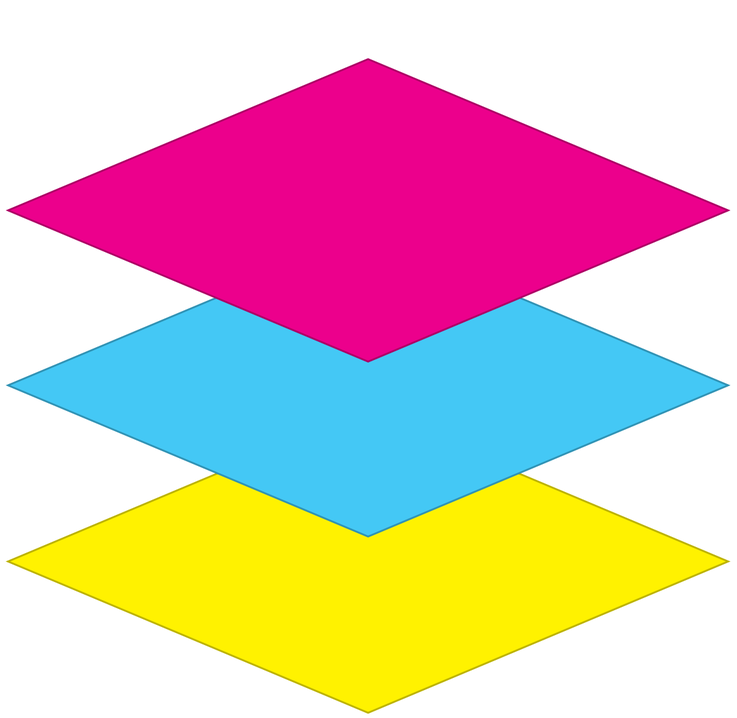
UI

FXML

Services

Data

MySQL



Domain (POJO)

POJO = Plain Old Java Object

howest hogeschool

The n-tier architecture

UI

FXML

Application

Services

Data

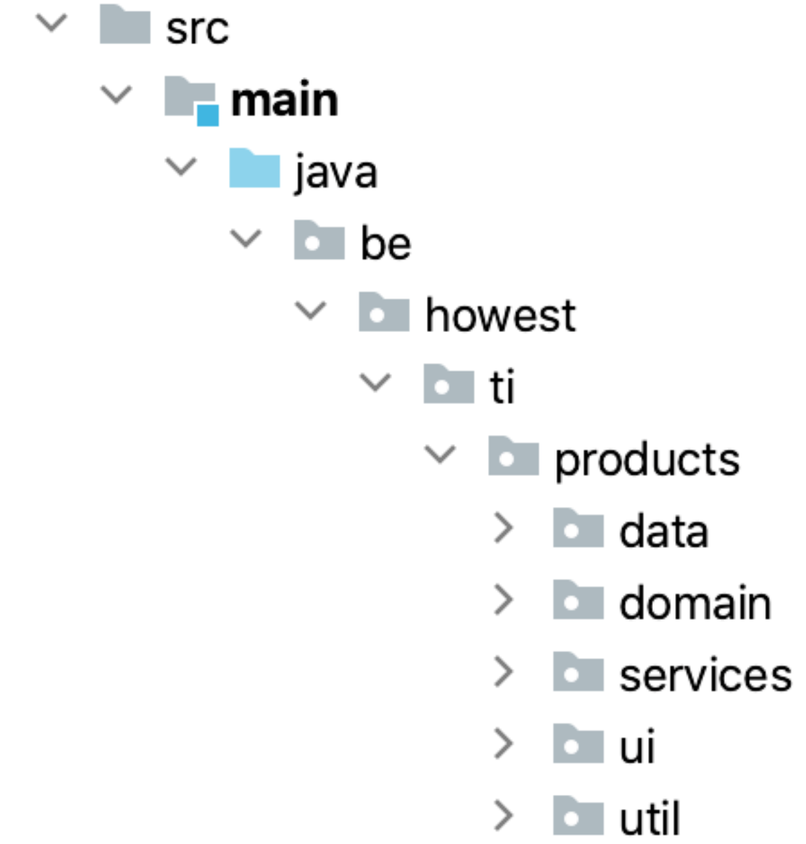
MySQL

POJO = Plain Old Java Object

howest hogeschool

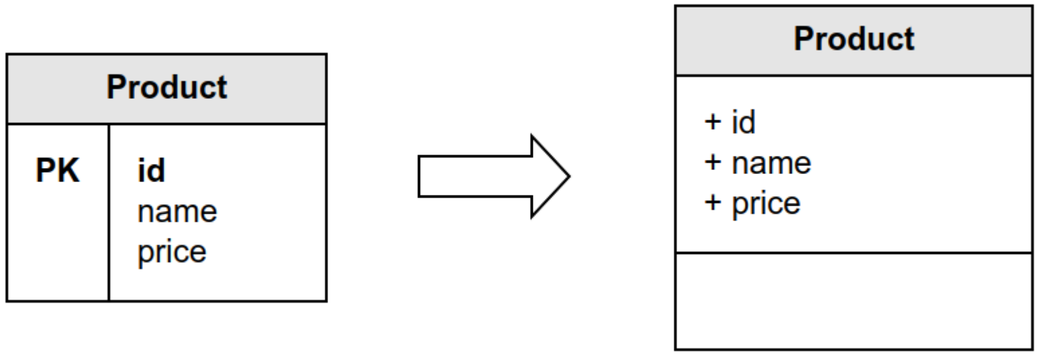
Implementing the n-tier architecture in our application.

- The data layer…
  - is reponsible for communication with the data source (here: MySQL)
- The domain/model layer…
  - is made-up of POJOs, representing entities from datasource
- The services layer…
  - contains non-UI logic
  - sits between the data layer and the UI
- The user interface layer…
  - is responsible for showing information on screen and capturing user input for further processing



The domain/model layer (1)

- The domain or model layer is used to represents the objects manipulated throughout the program.
- Usually, we define classes that map to our underlying datastore.
- For example, a database table **product** with columns **id**, **name** and **price** results in a class **Product** with fields **id**, **name** and **price**.



The domain/model layer (2)

```
public class Product {

    private int id;
    private String name;
    private double price;

    public Product(String name, double price) {
        this(0, name, price);
    }

    public Product(int id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

// ...
```


howest hogeschool

The data layer (1)

- A repository is responsible for handling the communication with the data source.
- Performing CRUD* operations using the domain/model objects.
- We will define an interface, specifying all methods our repository needs to provide:

```
public interface ProductRepository {

    List<Product> getProducts();
    void addProduct(Product product);

}
```

CRUD = Create, Read, Update, Delete

The data layer (2)

- Next, we will provide an **actual implementation** of the repository interface.
- For example, a **MySqlProductRepository**, communicating with the MySQL data source:

```
public class MySqlProductRepository implements ProductRepository {

    private static final String SQL_GET_PRODUCTS = "select * from product";
    private static final String SQL_ADD_PRODUCT = "insert into product(name, price) values(?, ?)";

    private final Logger logger = Logger.getLogger(getClass().getName());

    @Override
    public List<Product> getProducts() {
        try (Connection connection = MySqlConnection.getConnection();
             PreparedStatement stmt = connection.prepareStatement(SQL_GET_PRODUCTS);
             ResultSet rs = stmt.executeQuery()) {

// ...
```

- Why split the interface and the implementation?

The data layer (3)

- Our MySqlProductRepository implements the ProductRepository interface.
- Should our company switch from MySQL to e.g. JSON, we can easily switch implementations, without affecting the calling code.
- Indeed, the calling code only takes into account the interface, not the actual implementation.
- SQL code is nicely hidden in the implementation, our calling code doesn’t even know that SQL is being used.
- Our model/domain layer is not cluttered with SQL statements.

The services layer

- Sits between the data layer and the UI
- Contains logic that isn’t relevant to the UI
- “Intermediate person”
- When switching from one UI (e.g. FXML) to another (HTML, …) non-UI logic stays nicely encapsulated in services layer

The UI layer

- The UI layer talks to the services layer.
- Indeed, because the UI doesn’t need to know anything about the services/database layer’s actual implementation, we can switch UI without affecting database layer code.

The n-tier architecture

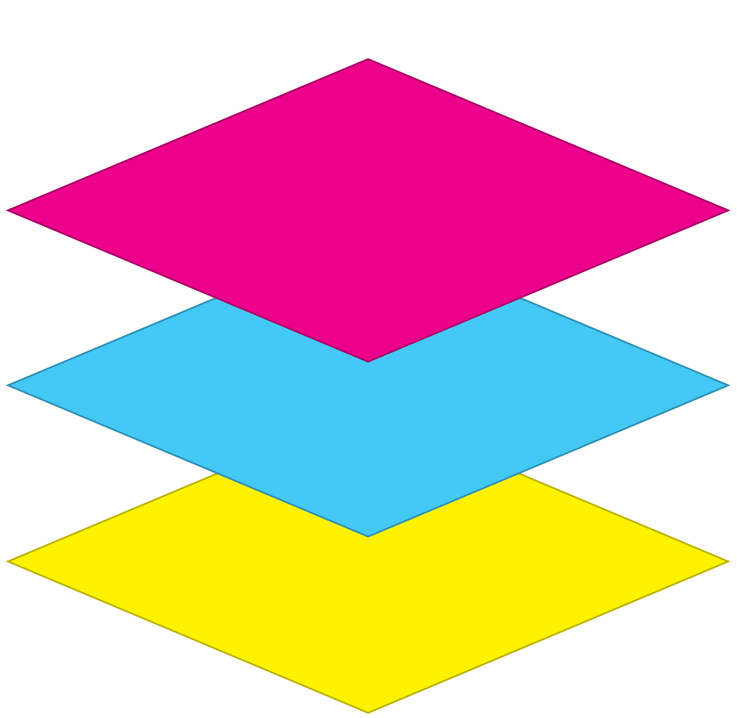
UI

FXML

Services

Data

MySQL



Domain (POJO)

POJO = Plain Old Java Object

howest hogeschool