

Principles of Software Design

Objects and databases

Robert Lukotka

lukotka@dcs.fmph.uniba.sk

www.dcs.fmph.uniba.sk/~lukotka

M-255

Objects and databases

We receive an event and we want to handle it. How would it look in an application that runs in-memory only?

- An object receives the event.
- To make the required calculation and/or state changes
 - several objects may be called
 - some objects may be created/deleted.

But what if we have a database where the state is shared by multiple clients?

Objects and databases

We have many problems.

- Objects exist in memory, database is on hard disk.
- If we access an object we may need to check the database for its attribute values.
 - We do not want to read each row separately as it makes too much calls.
 - On the other hand, we do not want to read way more than we need as it slows everything down.
- We would like ACID transactions.
 - If we read too much, the scope of the locks we have to obtain is big.
 - We need to decide what the transactions are.
- ...

Objects and databases

OCP: Classes should be closed for modification but open for extension.

- In memory, if we use OOP, business objects deal with business rules.
- Persistence is a separate concern.
- An ideal object-oriented design solution should extend business logic objects to have database storing capabilities.
- Unfortunately, storing stuff into database is not an easy task, thus some concerns regarding persistence must be considered even while designing the business logic.

Simple solutions

- One request, one transaction.
- We modify the getters so that they read value from the database.
- We modify the setters so that they write values to the database.

Problems include:

- way too many sequential database calls.
- the locks are requested in a random order, how to e.g. avoid deadlocks?

Simple solutions

- One request, one transaction.
- We read the whole state of the relevant part of the system affected by the call.
- We track what has changed (Unit of work pattern).
- In the end, we update what has changed.

Problems include:

- the “relevant part” may be too big (locks scope, data volume)

Simple solutions

- One request, one transaction.
- When an object is first accessed its state is read from the database (Proxy pattern).
- We track what has changed (Unit of work pattern).
- In the end, we update what has changed.

Problems include:

- maybe too many sequential database calls.
- the locks are requested in a random order, how to e.g. avoid deadlocks?

Simple solutions

- One request, one transaction.
- When an attribute is accessed it gets a new symbol.
- When an attribute is modified the modification is done in a symbolic manner, e.g. if the attribute has variable a and it is increased by 10 the resulting value is an expression $a + 10$, thus we do not read a .
- At the end we read the variables we need to evaluate the expressions.

Problems include:

- handling conditionals (one could use something like $?:$ operator in some situations, but not e.g, when the conditional is in the loop).

How to actually do that

If one of the simple approaches work for you, you are fine.
Otherwise you combine various approaches.

- One request, one transaction - often a good idea.
- You need to balance what you read at once (locks scope, data volume) and how many read requests you make (sequential reads = time, potential deadlocks).
 - Ideally you read exactly what you need, but to know what you need you often need to handle a part of the request.
 - You can sort out a lot of these issues by proper design decision (unfortunately, you have to consider the persistence requirement at least partially while designing the business logic objects).

How hard is to implement these solutions?

- Some of these solution are easier, some are harder to implement.
- Once you deal with a certain situation the task becomes repetitive as the situation reappears.
- After dealing with a certain amount of situations you rarely encounter a new one, you may simply constraint your design so only situation you already handled appear.
- When an task with these properties applies to a big area within an industry, automated tools to do the task appear.

These are called **ORM (object relational mapping) tools**.

Object-relational impedance mismatch

All the stuff we were dealing with in the previous slides are about problems that are not specific for O-O paradigm:

- Data duplication (memory, database).
- Concurrency issues (lock scope, setting up lock hierarchy).
- ...

But there are more straightforward issues:

- Classes have instances, inheritance, relationships; relational databases have just tables.
- References, pointers.
- Datatype differences.
- Database normal forms make little sense in OOP.
- ...

Object-relational impedance mismatch is a set of difficulties that we encounter when we use relational databases with an OO application.

How to map objects to tables

- Class Student with attributes name, surname.
 - Students(id, name: string, surname:string)
- Class Student has subclass PTSSStudent with additional attribute points.
 - Students(id, name: string, surname:string, type: string)
 - PTSSStudents(id, points: integer)
- Pointer/reference to some other object → id - foreign key
- Many to many relations → relation table.

There are many more situations (and the presented solutions are not the only ones).

Python ORM tools

As there are various approaches how to handle object-relational impedance mismatch, there are many competing Python ORM tools.

- SQLAlchemy
- DjangoORM
- Peewee ORM
- Pony ORM
- SQLAlchemy ORM

We will show some basics on SQLAlchemy.

SQLAlchemy

- Uses Python DBAPI to work with various relational databases.
- SQLAlchemyCore - A set of tools to work with relational databases.
- SQLAlchemyORM - ORM build over SQLAlchemyCore

Python DBAPI

- Implemented by third party libraries / python core.
- Major database systems have more than one implementation of DBAPI.
- Solves stuff like bounding parameters within DB queries.
- Slightly harder to use. It is good idea to have something over it.

SQLAlchemyCore

- Dialect - Engine uses it so it is able to deal with various databases.
- Engine - Something to run the queries.
- ConnectionPool - You typically want something like this.
- SQL Expression Language
- Schema, types

SQLAlchemyCore - Engine

```
engine = create_engine('mysql://jano@localhost/test')

engine.execute("insert into employees (name)
                values :name", name="Jano")

with engine.connect() as conn:
    result = conn.execute("select name from employees")
    for row in result:
        print("name:", row['name'])
```

SQLAlchemyCore - Engine

As you can see.

- Parameters are bounded
- Instead of cursor we get an object that behaves like tuple of dicts.
- ...

This helps quite a bit.

SQL Expression Language

```
s = select([users, addresses]).where(users.c.id ==  
addresses.c.user_id)
```

- Instead of SQL we produce statements in O-O manner.
- Hides differences between the databases but still allows you use database-specific tools if necessary.
- On the other hand it is quite unfortunate to have to learn something else instead of SQL.

SQLAlchemy ORM





This is where the actual object-relational mapping happens.

- Get familiar with the basics of SQLAlchemy in this simple [tutorial](#).

Resources I

- [Wikipedia - Object-relational impedance mismatch](#)
- [Object Relational Tutorial](#)

References |

-  [Wikipedia - Object-relational impedance mismatch](#)
-  [Wikipedia - Object-relational mapping](#)
-  [Full stack Python - Object-relational Mappers](#)
-  [SQLAlchemy documentation](#)