Mission 4 – Database Programming

The aim of this mission is to explore database programming along two directions: 1) how software *operations* complement a data model within a software design and 2) how those abstract operations can be implemented within the database itself (i.e. active databases) or outside of it (i.e. through another programming language and a connectivity layer). Doing so, you will learn about internal database tools such as views, triggers, and stored procedures but also database connectivity tools (e.g. JDBC, ORMs, etc.) when it comes to access your database from code.

From a problem statement, this mission proceeds in four steps:

1. Implement the software exclusively as an active database, favoring stored procedures, views, triggers, etc.

2. Implement a lightweight layer for invoking the services you've created at step 1 using a call-level API (e.g. JDBC in Java)

3. Implement the software a second time, without relying on database tools (no stored procedure, no views, no triggers). Use either a Query builder or Object Relational Mapping (ORM).

4. Step back a little, document your code, then write a short report (two pages max.) that summarizes your experiments and learning.

In the next mission (M5), you will conduct a review of solutions made by other students. An overview of the review focus is provided in the last section of this document.

Check your understanding

At the end of the mission you better understand how data and software operations work hand in hand regarding requirements. You have a good understanding of the tools you have used in your own project, e.g. active databases, call-level such as JDBC, or Object-relational mapping. You understand the advantages and drawbacks of the tools you've used.

You understand how views, rules, triggers may help a software implementation. You use them wisely but surely. Last but not least, you keep the need for transactions in mind when analyzing and implementing software requirements. Whatever your database connectivity tool of choice, you know how to start, commit and rollback transactions in practice so as to meet those requirements.

Submission & Deadline

This mission must be completed by groups of 2 students. The report and documented source code must be submitted *once per group* as a *single ZIP archive* both on *ModdleUCL* and *HotCRP* by the **4 May 2016, 10h45**.

You should have received an email with credentials to access the HotCRP website of the course (http://hotcrp.info.ucl.ac.be/ingi2172).

Application Domain: The Automated Café

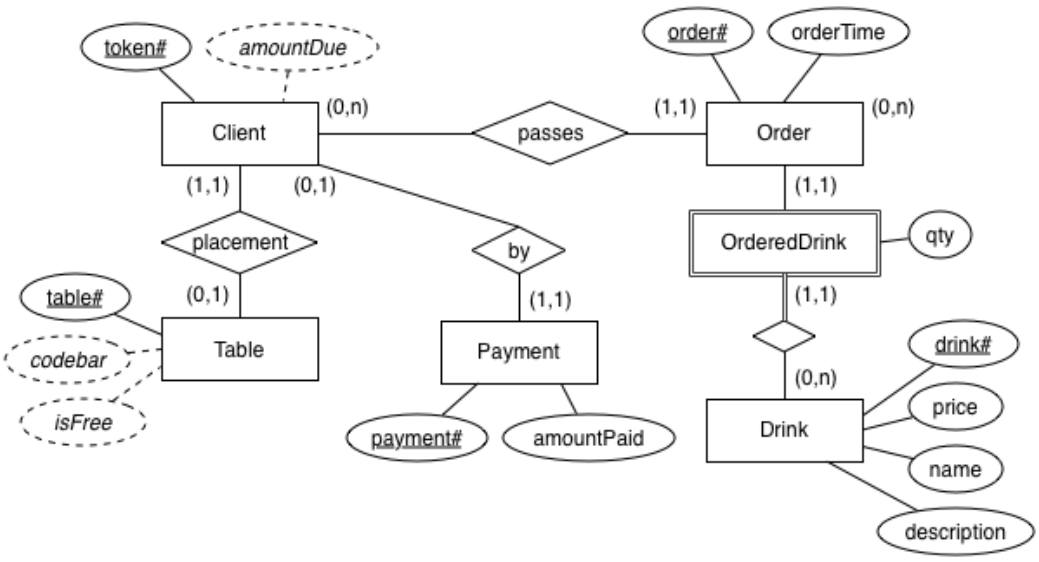
We consider here The Automated Café, a pub where drinks can be ordered at any time from a smartphone (instead of having to wait for the server to come at the table to take the order details). The principle is easy:

- When entering the pub, clients acquire a free table by scanning its bar code sticker with their smartphone. Acquiring a table provides the client's smartphone with a token to be used for identification during ordering.
- When a table is acquired, clients may order drinks at anytime (specifying the quantity for each drink, of course) on the smartphone app.
- When clients leave the pub, they pay all ordered drinks using their smartphone too. Paying releases the table, making it free again for other clients.

In this mission, we only consider the backend API of The Automated Café. We do not consider the smartphone application itself.

Conceptual data model

A conceptual data model covering the domain description is shown below. (This model is given to support our problem description, not as a prescriptive way of designing your database. Feel free to modify/enhance it the way you want to support whatever is needed).



Specification

We consider here the specification of the backend API, to which the smartphones connect on behalf of real users. We specify four abstract operations that support the ideal scenario (acquire a table ->

order drinks one or multiple times -> see the ticket -> pay and release the table). IN and OUT correspond to the operation parameters and returned value, respectively. PRE and POST are conditions to be met before and after the operation is invoked.

To keep things simple, many details are not considered (e.g. authentication, payment security, etc.). Also, we require operations to be implemented in a robust way, i.e. they must throw an error with an understandable message when a precondition is violated.

AcquireTable

DESC: invoked by the smartphone app when scanning a table code bar
IN: a table bar code
OUT: a client token
PRE: the table is free
POST: the table is no longer free
POST: issued token can be used for ordering drinks

OrderDrinks

DESC: invoked when the user presses the "order" button in the ordering screen
IN: a client token
IN: a list of (drink, qty) taken from the screen form
OUT: the unique number of the created order
PRE: the client token is valid and corresponds to an occupied table
POST: the order is created, its number is the one returned

IssueTicket

DESC: invoked when the user asks for looking at the table summary and due amount
IN: a client token
OUT: the ticket to be paid, with a summary of orders (which drinks in which quantities) and total amount to pay.
PRE: the client token is valid and corresponds to an occupied table
POST: issued ticket corresponds to all (and only) ordered drinks at that table

PayTable

DESC: invoked by the smartphone on confirmation from the payment gateway (we ignore security on purpose here; a real app would never expose such an API, of course).
IN: a client token
IN: an amount paid
OUT:
PRE: the client token is valid and corresponds to an occupied table
PRE: the input amount is greater or equal to the amount due for that table
POST: the table is released
POST: the client token can no longer be used for ordering

Mission details

This mission proceeds in four logical steps, by groups of two students. Feel free to organize your work the way you want though, e.g. by parallelizing the steps instead of making them sequentially.

Step 1: Active database

Implement the backend API with a PostgreSQL database.

1. Implement the data model as SQL tables
2. Implement the core operations with stored prodedures. Use views , triggers, rules, temporary tables, or whatever makes sense to help with the implementation of those procedures.

Output of this step:

- commented SQL source code with your entire solution, to be reviewed by other students .
- a separated script that executes the following scenario: a client acquires a table; he orders a sparkling water, look at the bill, then orders another sparkling water. The client then pays and releases the table.

Step 2: Call-level API

Connect to the database created at Step 1 using a call-level API. You can use the programming language of your choice (e.g. Java, Ruby, Python).

1. Figure out the way to send a query and iterate the results (e.g. to present a ticket in HTML).
2. Figure out the way to invoke a stored procedure, passing value arguments as needed.

Output of this step:

- commented source code that executes the sparkling water scenario from Step 1.

Step 3: Higher-level database connectivity

Revisit your implementation approach, by considering the database as pure storage (no views, no stored procedure, no triggers, etc.).

Implement the core features in the programming language used in Step 2. Use a higher-level approach to database connectivity, either through a Query builder or Object Relational Mapping. Higher-level frameworks are welcome too.

At this step, you may (be forced to) revisit the database entirely. For instance, some ORMs won't be able to use the same schema you designed in Step 1, or will even generate the schema automatically from an object model.

Output of this step:

- commented source code for the entire solution.
- script or test that executes the sparkling water scenario.

Step 4: One page report

Write a very short report that summarizes your work and what you've learned. See this report as a guide for your reviewers, to enter your commented source code and understand it. No need for very long explanations if your approach and code are clean.

Hint: the sparkling water scenario and its various implementations should serve as a self-contained guide through your code already. Take it as a starting point and make higher level comments about interesting database features along the code walk it naturally describes.

Output of this step: .pdf report, max 2 pages.

Review focus

This section provides examples of questions that students will answer when reviewing the solutions of other groups. The precise question list will be available in M5 only, but this section gives you a good overview of the focus.

[In Step 1] Are database tools wisely used? E.g. are views and/or triggers used to help reducing code redundancy, or to help with requirements evolution? If yes, cite an example of such use + traceability to source code.

[Step 1 and Step 3] Are preconditions enforced? If yes, does that lead to code duplication? If yes, how could it be avoided?

[Step 1 and Step 3] In each case, evaluate the cost of the following requirement evolution: we would like to allow multiple payments per table and separate the release of the table from the payment itself (through a new operation ReleaseTable). How much code must change, roughly? Could database tools be wisely used to shield from such hurting changes?

[All Steps] Are transactions correctly used? For instance, is the implementation of PayTable guaranteed to be atomic?

[All Steps] Are transaction boundaries explicit or implicit (e.g. through database magic connection handling, or the ORM/connectivity tool)? If implicit, are the transaction boundaries understood and explained in the report?