ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"ΠΛΗΡΟΦΟΡΙΚΗ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Απλούστευση της αλληλεπίδρασης με Τύπους Ορισμένους από τον Χρήστη της SQL με τη χρήση μιας Wrapper βιβλιοθήκης της JDBC

Παναγιώτης Σ. Σταυρόπουλος

Επιβλέπων: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

Αθήνα

Απρίλιος 2022

NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATION TECHNOLOGY AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM
"COMPUTER SCIENCE"

MASTER THESIS

Simplifying the Interaction with User-Defined Types in SQL via a JDBC Wrapper Library

Panagiotis S. Stavropoulos

Supervisor: Alex Delis, Professor NKUA

Athens

April 2022

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Απλούστευση της αλληλεπίδρασης με Τύπους Ορισμένους από τον Χρήστη της SQL με τη χρήση μιας Wrapper βιβλιοθήκης της JDBC

Παναγιώτης Σ. Σταυρόπουλος

Α.Μ.: CS2180015

Επιβλέπων: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

Απρίλιος 2022

MSC Thesis

Simplifying the Interaction with User-Defined Types in SQL via a JDBC Wrapper Library

Panagiotis S. Stavropoulos

S.N.: CS2180015

April 2022

ΠΕΡΙΛΗΨΗ

Η Java Database Connectivity (JDBC) είναι μια διεπαφή προγραμματισμού εφαρμογών (API) για τη γλώσσα προγραμματισμού Java, η οποία ορίζει τον τρόπο με τον οποίο ένας πελάτης μπορεί να έχει πρόσβαση σε μια βάση δεδομένων. Το γεγονός ότι είναι βασικό, το καθιστά φλύαρο και κουραστικό για εκτεταμένη άμεση χρήση και αυτό μεγιστοποιείται κατά την ενασχόληση με πιο σύνθετους User-Defined Types (UDT) στην SQL.

Η παρούσα διπλωματική εργασία περιγράφει τη διαδικασία υλοποίησης μιας βιβλιοθήκης περιτύλιξης διεπαφής JDBC που απλοποιεί την πρόσβαση στη βάση δεδομένων από την απλή εκτέλεση εντολών SQL σε πιο σύνθετες περιπτώσεις χρήσης με τύπους που ορίζονται από τον χρήστη. Για να το επιτύχουμε αυτό αυτοματοποιούμε και κρύβουμε από τον χρήστη διαδικασίες που έως τώρα το βάρος της υλοποίησής τους έπεφτε στον προγραμματιστή. Τέτοιες διαδικασίες είναι η προετοιμασία και η δήλωση των παραμέτρων εισόδου ή εξόδου, η διαδικασία κατασκευής του αποτελέσματος και άλλα.

Τα αποτελέσματα της παραπάνω διαδικασίας ήταν αρκετά ενθαρρυντικά και έδειξαν ότι η βιβλιοθήκη μας καταφέρνει να μειώσει δραματικά τον κώδικα που απαιτείται για τις πράξεις που χειρίζονται τους τύπους που ορίζονται από τον χρήστη, διατηρώντας παράλληλα την χρονική επιβάρυνση σε όχι σημαντικά επίπεδα. Η μείωση της πολυγλωσσίας τείνει να έχει περαιτέρω θετικές επιπτώσεις όπως η μείωση των πιθανοτήτων εμφάνισης σφαλμάτων και η αύξηση της παραγωγικότητας.

ΘΕΜΑΤΙΚΕΣ ΠΕΡΙΟΧΕΣ: Σχεσιακές Βάσεις Δεδομένων, Συνδεσιμότητα βάσης δεδομένων, SQL Τύποι Ορισμένοι από τον Χρήστη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Βάση Δεδομένων, JDBC, Wrapper Βιβλιοθήκη, SQL, UDT

ABSTRACT

Java Database Connectivity (JDBC) is an application programming interface (API) for the Java programming language that defines how a client can access a database. Being basic makes it verbose and tedious for extended direct use. This is intensified while dealing with more complex User-Defined Types (UDT) in SQL database language.

This thesis outlines the process of implementing a lightweight JDBC wrapper library that intents to simplify database access from simple execute of SQL statements to more complex use cases with user-defined types. To achieve this, we decrease verbosity by automating and hiding from the user many processes that until now their implementation burden has fallen on the developer. Such processes are the resource management, the preparation and declaration of input or output parameters, the process of building the result and more.

The results of the above process were quite encouraging and showed that our library manage to dramatically reduce the amount of code (~90%) required for operations handling user-defined types while keeping the time overhead in not considerable levels. Decreasing verbosity tends to have further positive effects such as reducing the chances of errors and increasing productivity.

SUBJECT AREAS: Relational Databases, Database Connectivity, SQL User-Defined Types

KEYWORDS: Database, JDBC, Wrapper Llibrary, SQL, UDT

To my late best friend Paco…

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Alex Delis, for his excellent guidance and support throughout the whole thesis.

To all my colleagues at UOA and at my job: I would also like to thank you for your wonderful cooperation. It was always helpful to discuss ideas with you about my research.

My parents and friends deserve special thanks. Your wise advice and kind words have served me well.

I hope you enjoy reading this thesis!

TABLE OF CONTENTS

1. INTRODUCTION 12

2. JAVA DATABASE CONNECTIVITY (JDBC) 13

3. SQL USER-DEFINED TYPES (UDT) 15

3.1 Structured types 15
3.2 Collections 16

4. SQL USER-DEFINED TYPES AND JDBC 18

5. JDBC WRAPER LIBRARY 20

5.1 Objectives of the library 20
5.2 Interaction with UDT 21

6. IMPLEMENTATION 23

6.1 Definition of UDT 23
6.3 Build SQL Statement 26
6.4 Prepare Input Parameters 26
6.5 Prepare Output Parameters 27
6.7 Build Output 29

7. CONCLUSION 30

TERMINOLOGY: ABBREVIATIONS AND ACRONYMS 31

APENDIX 32

SOURCE CODE 32

REFERENCES 33

LIST OF FIGURES

Figure 1: JDBC Architecture 13
Figure 2: Example of Stored Procedure call with UDT parameters via JDBC 19
Figure 3: Example of stored procedure call via JDBC wrapper library 22
Figure 4: Routine call flowchart 25
Figure 5: Prepare Input parameters process flowchart 27
Figure 6: Prepare Output parameters process flowchart 28

LIST OF TABLES

Table 1: UDT support in RDBMS vendors..15
Table 2: Abbreviations and acronyms 31

1. INTRODUCTION

Today all Java applications use the JDBC API for database connectivity. JDBC provides basic interfaces for the interaction with database. Its use can be either direct or behind the scenes with higher level of abstractions like JPA. Frameworks, such as Hibernate, hide any complexity that may arise from direct use of JDBC. Although, fully grown ORM frameworks may be heavyweight. On the other hand, JDBC being basic, makes it verbose and tedious for extended direct use, which is intensified while dealing with more complex scenarios like the interaction with User-Defined Types (UDT) in SQL database language.

SQL's UDT are custom types that extend the built-in types and, as we'll see later, can be quite useful in some situations. Although their use can introduce new functionality and benefits at the database layer, it increases the complexity of implementing the data access layer. Interacting with the UDT via direct JDBC is verbose, tedious, and requires plenty of boilerplate code for simple procedures such as a function call. In addition, we could not identify any framework or library that supports them, and JDBC support for UDT is not well documented. Therefore, simplifying UDT interaction via JDBC has become a critical issue that yet is technically challenging.

Our main goal is to create a lightweight JDBC wrapper library that supports and simplifies the use of SQL UDT. With this library, we aspire to bridge the gap between low-level JDBC access and ORM frameworks while keeping overhead at negligible levels. Our library should be easy to learn and use and achieve LOC reduction compared to direct use of JDBC. Finally, we try to introduce an additional layer of error prevention in the implementation of the data access layer. To achieve our goals, we will automate and hide from the user many of the processes required to interact with the UDT.

2. JAVA DATABASE CONNECTIVITY (JDBC)

Java Database Connectivity (**JDBC**) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation. It provides methods to query and update data in a database and is oriented toward relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment. [1]

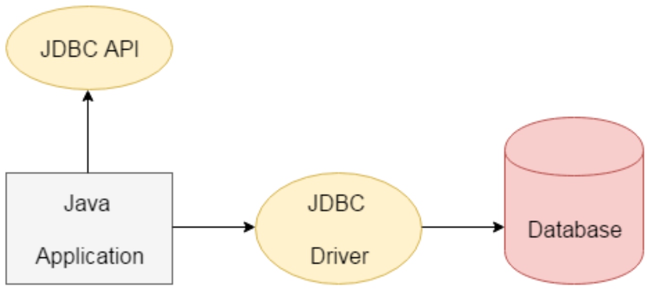


Figure 1: JDBC Architecture

Bellow we present an example Java program that uses the JDBC interface. The code illustrates how connections are opened, how statements are executed and results processed, and how connections are closed. [2]

```
public static void JDBCexample(String userid, String passwd)
 {
    try
    {
       Class.forName ("oracle.jdbc.driver.OracleDriver");
       Connection conn = DriverManager.getConnection(
             "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
             userid, passwd);
       Statement stmt = conn.createStatement();
       try {
          stmt.executeUpdate(
                "insert into instructor values(’77987’, ’Kim’, ’Physics’, 98000)");
       } catch (SQLException sqle)
       {
          System.out.println("Could not insert tuple. " + sqle);
       }
       ResultSet rset = stmt.executeQuery(
             "select dept name, avg (salary) "+
                   " from instructor "+
                   " group by dept name");
       while (rset.next()) {
          System.out.println(rset.getString("dept name") + " " +
                    rset.getFloat(2));
            }
            stmt.close();
            conn.close();
        }
        catch (Exception sqle)
        {
            System.out.println("Exception : " + sqle);
        }
    }
```

The word JDBC was originally an abbreviation for Java Database Connectivity, but the full form is no longer used. The Java program must import java.sql.*, which contains the interface definitions for the functionality provided by JDBC. [2]

We use JDBC as the core of our library to automate processes and eliminate the boilerplate code that used to be a burden of the developers. We expose interfaces for the main functions we will cover hiding the complexity from the end user.

3. SQL USER-DEFINED TYPES (UDT)

User-defined types (UDT) were introduced in the SQL 2000 standard. They are custom complex types defined by the user. Many RDBMS vendors support their use (definition, creation, pass as parameters, use as a column type, etc.) with direct SQL commands, while other vendors support them partially, by which we mean that the user may need to write code in a vendor-specific programming language such as SQL-CLR and .NET for Microsoft SQL Server or like C in Postgres. Finally, there are RDBMS that are not yet supported by.

In the table below we present some of the most popular RDBMS vendors and the level of UDT support they provide.

Table 1: UDT support in RDBMS vendors

Vendor	Fully Supported (SQL)	Extension-based supported	Not yet supported
Oracle Db	✓		
IBM DB2	✓		
Microsoft SQL Server		✓	
Postgres		✓	
MySQL			✓
MariaDB			✓

Using UDTs we can extend the built-in types and we can enforce standards. Standards can improve the readability of code. In addition, with UDTs we can introduce Object-Oriented concepts at the database level.

There are different types of UDTs and several times we find differences depending on the RDBMS vendor. We in the context of the thesis will deal with the two main types as defined by the SQL standard, the Structured types and the Collections.

3.1 Structured types

Structured types allow composite attributes of E-R designs to be represented directly. Those types are ccomparable to classes in object-oriented languages. Such types are called user-defined types in SQL. [2]

For instance, we can define the following structured type to represent a composite attribute name with component attribute firstname and lastname:

```
create type Name as (
firstname varchar(20),
lastname varchar(20)) ;
```

Similarly, the following structured type can be used to represent a composite attribute address:

```
create type Address as
(street varchar(20),
city varchar(20),
zipcode varchar(9));
```

Such types are called user-defined types in SQL. The first form of CREATE TYPE creates a composite type. The composite type is specified by a list of attribute names and data types. An attribute's collation can be specified too if its data type is collatable. A composite type is essentially the same as the row type of a table but using CREATE TYPE avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful, for example, as the argument or return type of a function.

3.2 Collections

Collections are comparable to arrays in other programming languages. We can now use these types to create composite attributes in a relation, by simply declaring an attribute to be of one of these types. For example, we could create a table person as follows:

```
create table person (
name Name,
address Address,
dateOfBirth date
);
```

The components of a composite attribute can be accessed using a “dot” notation; for instance name.firstname returns the firstname component of the name attribute. An access to attribute name would return a value of the structured type Name. We can also create a table whose rows are of a user-defined type.

For example, we could define a type PersonType and create the table person as follows:

```
create type PersonType as (
name Name,
address Address,
dateOfBirth date);

create table person of PersonType;
```

A structured type can have methods defined on it. We declare methods as part of the type definition of a structured type.

4. SQL USER-DEFINED TYPES AND JDBC

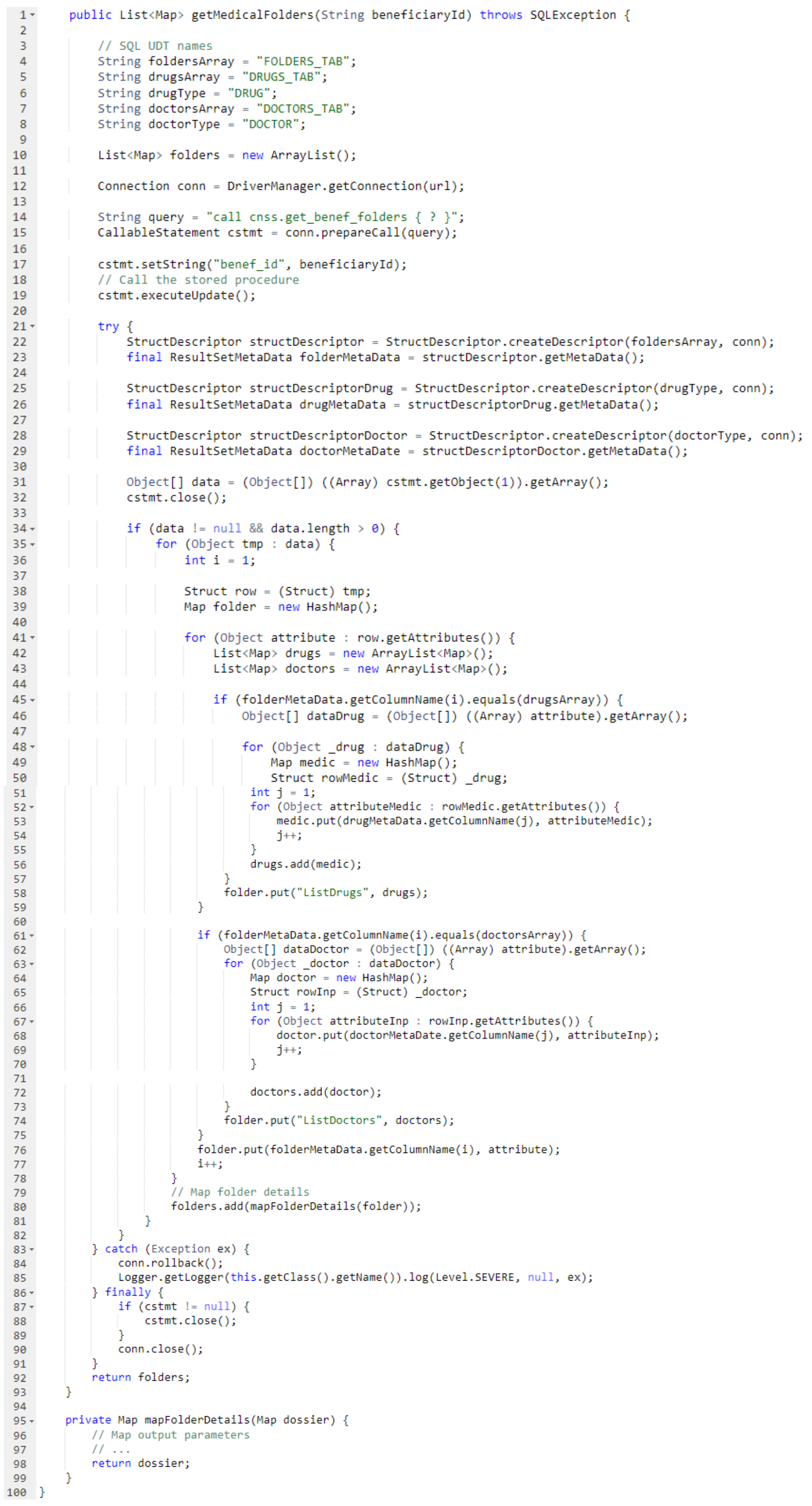
JDBC, being basic, has some disadvantages for extensive direct use. It is quite verbose and tedious. This is multiplied when we need to interact with complex UDTs via functions and procedures. In addition, this area is not well documented and there is a lack of libraries that simplify interaction with the UDT via JDBC.

Code LOC and complexity increase in proportion to the complexity of the UDTs. As a result, the implementation time spent increases and programmers must write a lot of boilerplate code to handle these types. In addition, the user must handle low-level concepts such as JDBC resource handling. In case of mismanagement this can lead to critical errors.

These features usually lead developers to use higher level abstractions such as JPA. Although fully developed ORM frameworks can be heavyweight something we may want to avoid in various situations.

The following snippet (Snippet 1) shows an example of a stored procedure call via JDBC. This procedure takes as input parameter a simple string and returns as output parameter a Collection UDT with two nested Structured UDTs.

This example could become even longer and more complex if the input parameters were also UDTs or if the structure of these UDTs were even more complex with more fields and nested types.



```java
    public List<Map> getMedicalFolders(String beneficiaryId) throws SQLException {

        // SQL UDT names
        String foldersArray = "FOLDERS_TAB";
        String drugsArray = "DRUGS_TAB";
        String drugType = "DRUG";
        String doctorsArray = "DOCTORS_TAB";
        String doctorType = "DOCTOR";

        List<Map> folders = new ArrayList();

        Connection conn = DriverManager.getConnection(url);

        String query = "call cnss.get_benef_folders { ? }";
        CallableStatement cstmt = conn.prepareCall(query);

        cstmt.setString("benef_id", beneficiaryId);
        // Call the stored procedure
        cstmt.executeUpdate();

        try {
            StructDescriptor structDescriptor = StructDescriptor.createDescriptor(foldersArray, conn);
            final ResultSetMetaData folderMetaData = structDescriptor.getMetaData();

            StructDescriptor structDescriptorDrug = StructDescriptor.createDescriptor(drugType, conn);
            final ResultSetMetaData drugMetaData = structDescriptorDrug.getMetaData();

            StructDescriptor structDescriptorDoctor = StructDescriptor.createDescriptor(doctorType, conn);
            final ResultSetMetaData doctorMetaDate = structDescriptorDoctor.getMetaData();

            Object[] data = (Object[]) ((Array) cstmt.getObject(1)).getArray();
            cstmt.close();

            if (data != null && data.length > 0) {
                for (Object tmp : data) {
                    int i = 1;

                    Struct row = (Struct) tmp;
                    Map folder = new HashMap();

                    for (Object attribute : row.getAttributes()) {
                        List<Map> drugs = new ArrayList<Map>();
                        List<Map> doctors = new ArrayList<Map>();

                        if (folderMetaData.getColumnName(i).equals(drugsArray)) {
                            Object[] dataDrug = (Object[]) ((Array) attribute).getArray();

                            for (Object _drug : dataDrug) {
                                Map medic = new HashMap();
                                Struct rowMedic = (Struct) _drug;
                                int j = 1;
                                for (Object attributeMedic : rowMedic.getAttributes()) {
                                    medic.put(drugMetaData.getColumnName(j), attributeMedic);
                                    j++;
                                }
                                drugs.add(medic);
                            }
                            folder.put("ListDrugs", drugs);
                        }

                        if (folderMetaData.getColumnName(i).equals(doctorsArray)) {
                            Object[] dataDoctor = (Object[]) ((Array) attribute).getArray();
                            for (Object _doctor : dataDoctor) {
                                Map doctor = new HashMap();
                                Struct rowInp = (Struct) _doctor;
                                int j = 1;
                                for (Object attributeInp : rowInp.getAttributes()) {
                                    doctor.put(doctorMetaDate.getColumnName(j), attributeInp);
                                    j++;
                                }

                                doctors.add(doctor);
                            }
                            folder.put("ListDoctors", doctors);
                        }
                        folder.put(folderMetaData.getColumnName(i), attribute);
                        i++;
                    }
                    // Map folder details
                    folders.add(mapFolderDetails(folder));
                }
            }
        } catch (Exception ex) {
            conn.rollback();
            Logger.getLogger(this.getClass().getName()).log(Level.SEVERE, null, ex);
        } finally {
            if (cstmt != null) {
                cstmt.close();
            }
            conn.close();
        }
        return folders;
    }

    private Map mapFolderDetails(Map dossier) {
        // Map output parameters
        // ...
        return dossier;
    }
}
```

Figure 2: Example of Stored Procedure call with UDT parameters via JDBC

5. JDBC WRAPER LIBRARY

In the context of this thesis, we developed a JDBC wrapper library in java to simplify interaction with a database via JDBC. Our library supports most of the quite used operations for accessing and modifying data as well as operations for developing and updating the database schema. Although our main goal is to focus on interaction with UDTs and to simplify their use.

Our library aims to have the following characteristics. To be a lightweight wrapper around the JDBC API using 100% pure JDBC and not any vendor specific code. To make interaction with the Database less verbose (LOC reduction) and less tedious by automating many processes that until now have required a lot of boilerplate code to implement. In this way we aim to simplify complex cases especially when interacting with the UDT. Furthermore, our library aims to be easy to learn and use and finally provide an additional level of error prevention.

5.1 Objectives of the library

The main goal of our library is to simplify interaction with the UDT. However, it supports many other functions for data access, data modification, and more. The objectives of the library are listed below:

- Support for user-defined types
    - Collections
    - Structured types
- Routine calls
    - Stored procedures
    - Functions
- CRUD operations
    - Create
    - Read
    - Update
    - Delete
- Management of JDBC internal resources
    - Connections
    - Callable Statements
    - Prepared Statements
    - Result sets

- Database schema management
    - Create
    - Alter

We focus on the interaction with UDTs when passing them as parameters of stored procedures and functions. Examples of all other functions can be found at the end of this document (APPENDIX A).

At this point we would like to mention that in this thesis we will not deal with issues such as Object Relationship Mapping, query building and security when using the library. Thus, we consider the following as out of scope:

- OR-Mapping
- SQL query builder
- Query generation capabilities
- Generated classes or dynamically generated proxies
- Connection pooling
- Security

To develop our library, we first did a survey of other existing work. We studied two libraries that were close to our main goals:

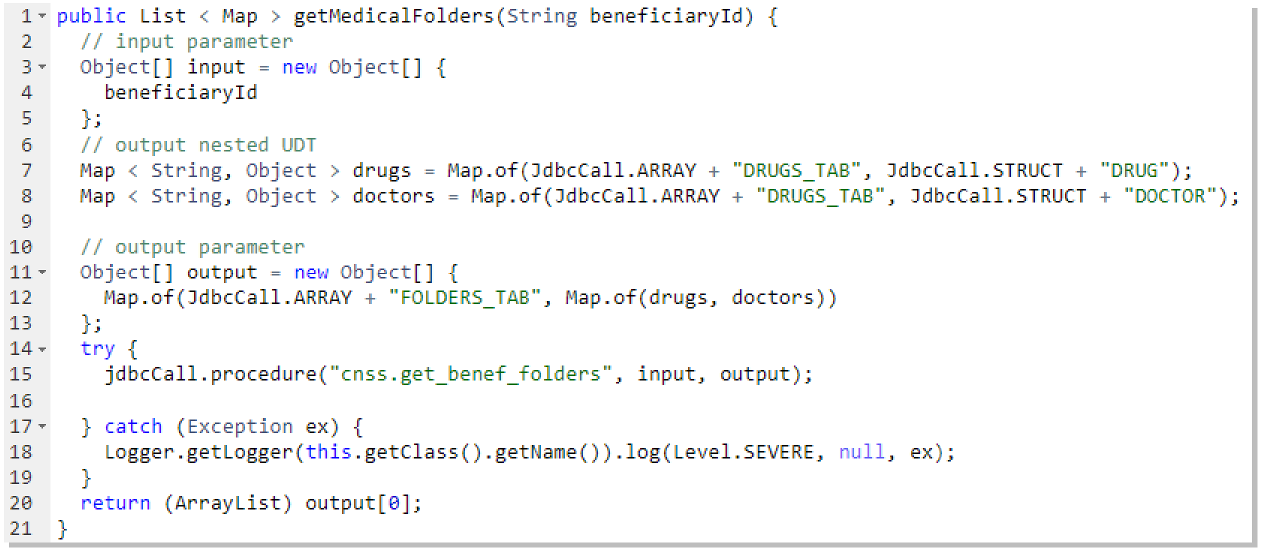
- jOOQ: Excellent library for accessing databases in a secure way [3]. Because of its different scope, it is more than just a thin wrapper around the JDBC API.
- Jdbi: Similar scope, but with a different approach [4].

Although both libraries do not support UDT.

5.2 Interaction with UDT

Our JDBC wrapper library supports interaction with the UDT. This interaction is simplified compared to direct JDBC access. To achieve this, we automate processes that previously required a lot of boilerplate code. Such processes are the definition of the input and output parameters. We present those processes on detail in the next chapter.

The following figure (Figure 3) illustrates a stored procedure call with UDT as output parameters via the JDBC wrapper library.



```java
public List < Map > getMedicalFolders(String beneficiaryId) {
  // input parameter
  Object[] input = new Object[] {
    beneficiaryId
  };
  // output nested UDT
  Map < String, Object > drugs = Map.of(JdbcCall.ARRAY + "DRUGS_TAB", JdbcCall.STRUCT + "DRUG");
  Map < String, Object > doctors = Map.of(JdbcCall.ARRAY + "DRUGS_TAB", JdbcCall.STRUCT + "DOCTOR");

  // output parameter
  Object[] output = new Object[] {
    Map.of(JdbcCall.ARRAY + "FOLDERS_TAB", Map.of(drugs, doctors))
  };
  try {
    jdbcCall.procedure("cnss.get_benef_folders", input, output);

  } catch (Exception ex) {
    Logger.getLogger(this.getClass().getName()).log(Level.SEVERE, null, ex);
  }
  return (ArrayList) output[0];
}
```

Figure 3: Example of stored procedure call via JDBC wrapper library

The above example (Figure 3) depicts the same procedure call that we previously illustrated (Figure 2) using direct JDBC code. It is obvious that the LOCs are dramatically reduced from 100 to 21. We achieve even greater LOC reduction in more complex cases where UDTs exist in both input and output parameters. The benefits also increase while having to interact with more complex types including nested UDTs.

6. IMPLEMENTATION

In this section we present the details of the JDBC wrapper library implementation. Once again, we focus on the interaction with the UDT. We see how we define the structure of UDTs to use them as parameters of stored procedure and function calls. We further present the algorithm implemented to call database routines (functions and procedures) with UDT parameters and explain in detail each separate step of the workflow.

6.1 Definition of UDT

To interact with the UDT, the user must first define the structure of each custom type. In the remainder of this document, we will refer to UDT types that do not contain other nested UDTs as simple UDTs and we will refer to UDT types that contain nested UDT as complex UDT.

Bellow we give an example of a Medical Folder object witch includes prescriptions of drugs and the prescriber doctors defined as complex UDT in PL-SQL:

```
CREATE TYPE folder {
drugs drug_tab,
doctors doctor_tab };
```

We also provide the definitions for the drug collection (drug_tab) and doctor collection (doctor_tab) as well as for their nested custom types:

```
CREATE drug_tab
        IS TABLE OF drug;

CREATE TYPE drug {
drug_code VARCHAR2 (20),
drug_name VARCHAR2 (50) };
```

```
CREATE doctor_tab
        IS TABLE OF doctor;

CREATE TYPE doctor {
doctor_id NUMBER,
doctor_name VARCHAR2 (50) };
```

In order to use the folder UDT object in java with our library, we first need to define its type. To support this process we have defined two constants in our wrapper library. One for collections, **JdbcCall.ARRAY** and one for structured types, **JdbcCall.STRUCT**.

After we define the complex UDT type, we are able to use it as an input parameter or an output parameter during a procedure or a function call. We will discuss this process further on the next section.

Below is an example of defining the complex UDT folder type using the tools provided by the library:

1. We start by defining the inner nested types drug and doctor objects as follows

```
String drug = JdbcCall.STRUCT + "DRUG";
String doctor = JdbcCall.STRUCT + “DOCTOR";
```

2. We define the collection types as a Map with key the type and value the type of the nested object as defined before

```
Map<String, Object> drugs = Map.of(JdbcCall.ARRAY + "DRUG_TAB", drug);
Map<String, Object> doctors = Map.of(JdbcCall.ARRAY + "DOCTOR_TAB",
        doctor);
```

3. Finally we define the folder object as follows

```
Map<String, Object> folder = Map.of(JdbcCall.STRUCT + "FOLDER", Map.of(drugs,
doctors) ) };
```

We do not need to explicitly define the nested types of drug and doctor object as those are simple types in PL-SQL.

6.2 Routine Calls

After defining the structure of the UDT we are able to use them in stored procedures and function calls as input parameters, output parameters or as the return type of a function. In the remaining chapters of section 6 we describe the process of routine calls with UDT as parameters. We use the term routine to describe a stored procedure or a function.

From a high-level view the algorithm implemented for routine calls is modelled in the flowchart of Figure 4:

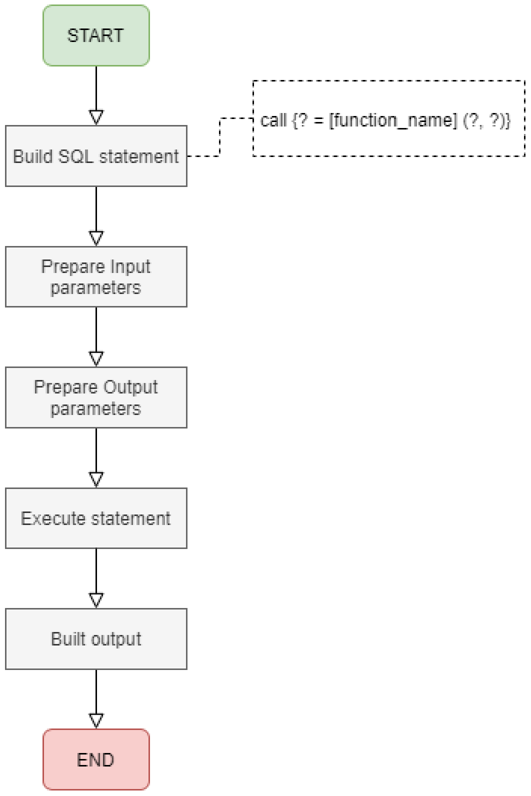


Figure 4: Routine call flowchart

The flowchart is depicting the workflow if the Routine call process as it is implemented internally in our wrapper library. We assume that the user has already defined correctly the input and the output parameters as we described earlier on chapter 6.1.

In order to tigger the process we need to call either the procedure or the function interface of our library like in the example bellow:

```
String protocolNumber = ’’1002446698’’;
jdbcCall.procedure( "get_medical_folder", new Object{protocolNumber}, new
        Object {folder} );
```

In the previous example the parameters are:

1. The procedure name as a string.
2. The input parameters (a simple string in our case) as list of Objects.
3. The output parameters (the complex UDT folder) as list of Objects.

During the next sections we will describe in detail each step of the Routine call workflow (Figure 4)

6.3 Build SQL Statement

The first step of our workflow is the simplest one. Here we build the statement, that means we construct the SQL statement as a string in order to call the routine. To do so we need to know the routine name and the number of input and output parameters. All this information is easy to be extracted by the procedure method parameters, we saw in the previous chapter 6.2.

6.4 Prepare Input Parameters

We follow a similar recursive process for building and preparing the parameters of each routine. In this step we describe the Prepare Input Parameters workflow.

Bellow we describe each step of the process as it is executed internally in our wrapper library:

1. We initialise a CallableStatement object of JDBC. This object will hold all the information needed in order to call the routine.
2. For each input parameters we check the type of the parameter (input parameters are passed as a list of objects in the procedure method as we saw in chapter 6.2).

   2.1 We register the parameter type in the CallableStatement.

   2.2. If the parameter is a UDT then we repeat step 2 for each nested element
3. The process is finished once all input parameters are registered

The process of preparing input parameters is depicted bellow (Figure 5):

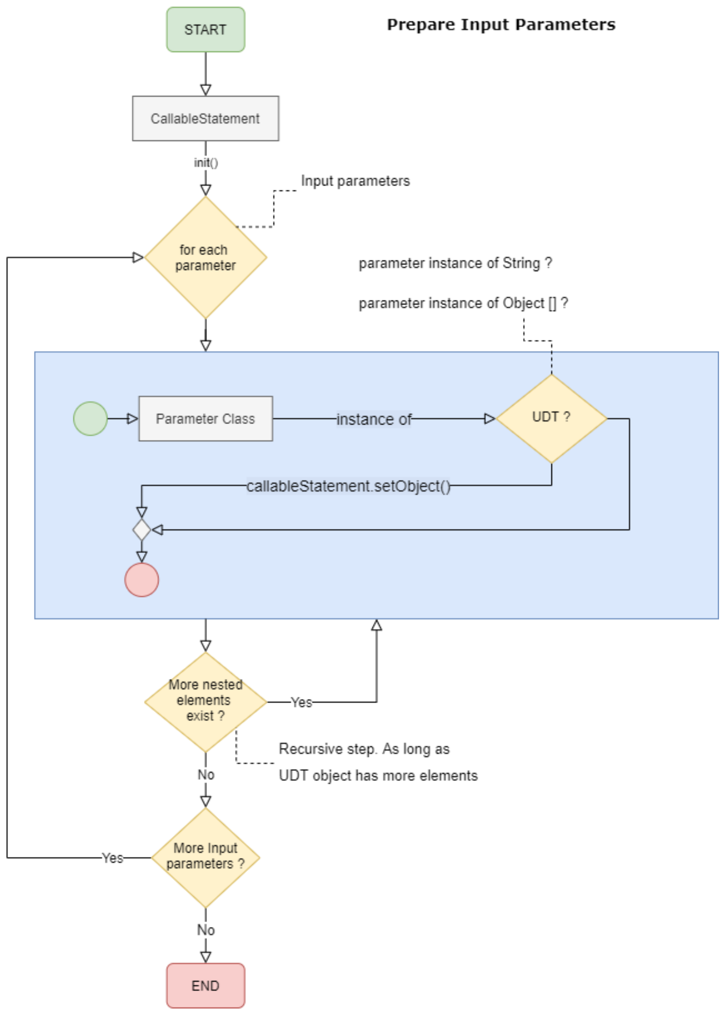

Prepare Input Parameters
START
CallableStatement
init()
Input parameters
for each parameter
parameter instance of String ?
parameter instance of Object [] ?
Parameter Class
instance of
UDT ?
callableStatement.setObject()
More nested elements exist ?
Yes
Recursive step. As long as UDT object has more elements
No
More Input parameters ?
Yes
No
END

Figure 5: Prepare input parameters process flowchart

6.5 Prepare Output Parameters

We follow a similar recursive process for building and preparing the output parameters of each routine.

Bellow we describe each step of the process as it is executed internally in our wrapper library:

1. We use the CallableStatement object we initialised in the Prepare Input Parameters process.
2. For each output parameter we check its type (output parameters are passed as a list of objects in the procedure method as we saw in chapter 6.2).

   2.1 We register the parameter type in the CallableStatement.

2.2. If the parameter is a UDT then we repeat step 2 for each nested element

3. The process is finished once all output parameters are registered

The process for preparing output parameters is depicted bellow (Figure 6):

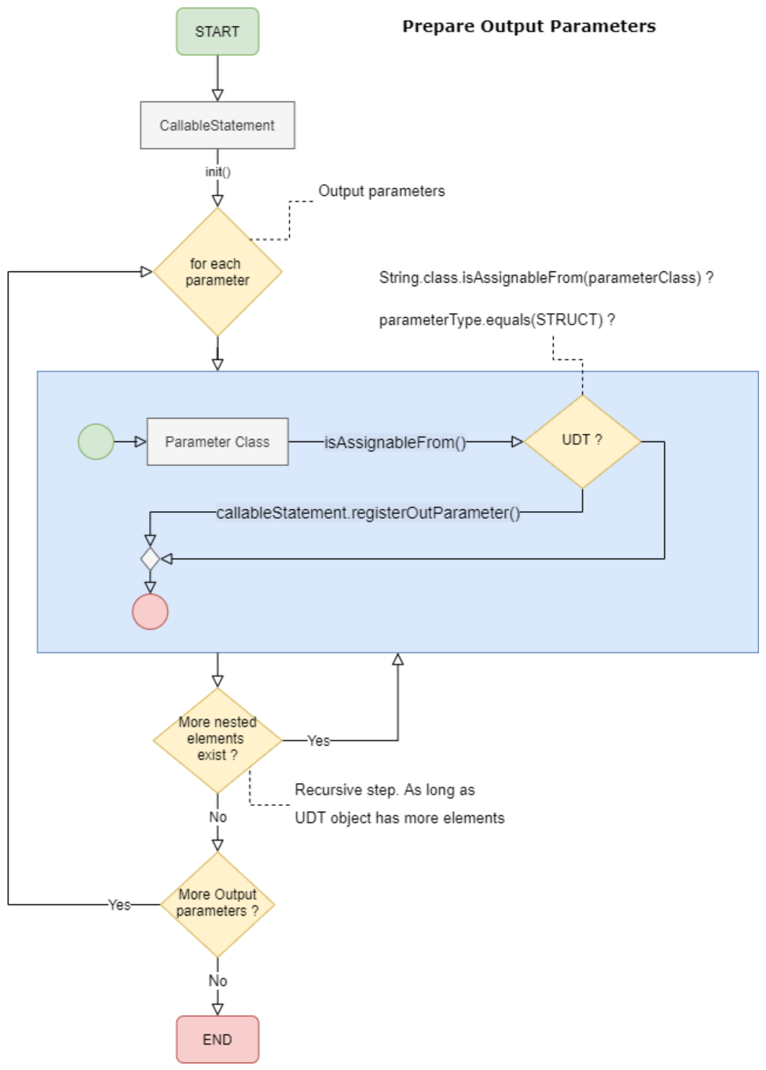

Prepare Output Parameters
START
CallableStatement
init()
Output parameters
for each parameter
String.class.isAssignableFrom(parameterClass) ?
parameterType.equals(STRUCT) ?
Parameter Class
isAssignableFrom()
UDT ?
callableStatement.registerOutParameter()
More nested elements exist ?
Yes
Recursive step. As long as UDT object has more elements
No
More Output parameters ?
Yes
No
END

Figure 6: Prepare Output parameters process flowchart

6.6 Execute Statement

After we have defined all the routine parameters and we have registered them in the CallableStatement object we are ready to call the actual routine. To do so we use the JDBC execute interface of the CallableStatement object.

6.7 Build Output

Building output is another a recursive processes similar to the Prepare input and Prepare output parameters processes.

Bellow we describe each step of the process as it is executed internally in our wrapper library:

1. We depict the call of the CallableStatement.execute method as the first step of the process in order to be clear that the following steps are executed after this.
2. We initialise a list of Objects that will carry the data of the routine output.
3. For each output object we have defined earlier we check the type.

    2.1 If the parameter is a simple type we retrieve the data from the ResultSet and we add its value in the output list of Objects.

    2.2 If the parameter is a UDT then we repeat step 2 for each nested element.

4. The process is finished once all output parameters are extracted and added in the output list of Objects.

5. The output is returned.

7. CONCLUSION

Interacting with User Defined Types of SQL from a Java Application is a troublesome process that leads to verbose and tedious code. So, we created a JDBC wrapper library that simplifies the interaction with those types and achieves high LOC reduction (~90% for complex cases) making the interaction with UDTs much easier.

We focused on the interaction with UDTs through the calls. There we described how we can define UDT in order to pass them as input and output parameters in stored procedure or function calls and finally to retrieve the actual output of the routine.

There are several open issues for improvement and future research. For example, how we could automate other processes which include interaction with UDTs like the creation of a table where a column has a UDT type and also operations like SELECT, UPDATE etc for this kind of tables which include UDTs

TERMINOLOGY: ABBREVIATIONS AND ACRONYMS

Table 2: Abbrevations and acronyms

Abbreviation	**Meaning**
API	Application Programming Interface.
C	C programming is a general-purpose, procedural, imperative computer programming language
E-R	Entity Relationship
JDBC	Java Database Connectivity is an application programming interface (API) for the programming language Java, which defines how a client may access a database.
JPA	Java Persistence API. It's a specification which is part of Java EE and defines an API for object-relational mappings and for managing persistent objects.
LOC	Lines Of Code.
ORM	Object Relational Mapping.
OR-Mapping	Object Relational Mapping is a programming technique for converting data from relational model to object code of object-oriented programming
RDBMS	Relational Database Management System is a system used to maintain relational databases.
SQL	Structured Query Language is used to communicate with a database.
UDT	User-Defined Types.

APENDIX

SOURCE CODE

Repository: https://github.com/StavPanos/JDBCmooth

REFERENCES

[1] JDBC API Specification Version: 4.0. https://www.oracle.com/java/technologies/ [Accessed 10/12/2021]

[2] Database System Concepts Sixth Edition, Abraham Silberschatz, Henry F. Korth, S. Sudarshan pp. 1-352.

[3] JOOQ library: https://www.jooq.org/ [Accessed 10/12/2021]

[4] Jdbi library: https://jdbi.org/ [Accessed 10/12/2021]