

ABTC: Multi-purpose Adaptable Business Tier Components Based on Call Level Interfaces

Óscar Mortágua Pereira, Rui L. Aguiar
Instituto de Telecomunicações, DETI - University of Aveiro, Portugal

Abstract

Call Level Interfaces (CLI) are low level API that play a key role in database applications whenever a fine tune control between application tiers and the host databases is a key requirement. Unfortunately, in spite of this significant advantage, CLI were not designed to address organizational requirements and contextual runtime requirements. Among the examples we emphasize the need to decouple or not to decouple the development process of business tiers from the development process of application tiers and also the need to automatically adapt to new business and/or security needs at runtime. To tackle these CLI drawbacks, and simultaneously keep their advantages, this paper proposes an architecture relying on CLI from which multi-purpose business tiers components are built, herein referred to as Adaptable Business Tier Components (ABTC). This paper presents the reference architecture for those components and a proof of concept based on Java and Java Database Connectivity (an example of CLI).

Keywords: software architecture, components, reuse, access control, information security, call level interfaces.

1. Introduction

This paper is an extended version of the paper presented at ICIS 2015 (IEEE ACIS) [1].

Software systems have increasingly played a key role in small, medium and large organizations by managing the data from which everyday decisions are taken. Data is mostly kept and managed by database management systems. Among the several paradigms, the relational database management systems (RDBMS) continue to be one of the most successful to manage data and, therefore, to build database applications. To be useful, data needs to be inserted, updated, retrieved and processed. In this case Call Level Interfaces (CLI) are effective solutions for building business tiers whenever a fine tune control on the interactions with the host databases is a key requirement [2]. The fine tune control comprises not only the services provided by CLI but also the possibility of using the full expressiveness of the SQL language. In spite of these important advantages, CLI convey some drawbacks, hereafter described.

Problem definition: CLI are general low level API that do not provide any high level assistance to address organizational requirements and runtime requirements. Three examples are provided: 1) in some organizations, business tiers and application tiers are developed by different actors (people playing different roles); 2) in other organizations, the actor is the same for the two tiers and 3) in some database applications, business tiers need to be dynamically adapted, at runtime, to address runtime needs, for example, to address security policies or to address new business needs. These CLI drawbacks are mainly derived from their technical and architectural aspects. Figure 1 presents a typical and simple case based on a CLI, in this case Java Database Connectivity (JDBC) [3]. Hereafter, all examples use Java, JDBC and the Microsoft Northwind database (<http://www.microsoft.com/download/en/details.aspx?id=23654>). Figure 1 depicts a program to retrieve data from a table named *Products* and also to update the attribute *unitPrice* of a list of products. The list of products to be updated is included in *List<Integer> productId* and the new values for *unitPrice* are included in *List<BigDecimal> unitPrice* (see arguments of method *updUnitPrice*). The product list is iterated (line 31), the Select expression is prepared and executed (line 32-35), if a product is found (line 36) some attributes are read (line 37-39) and *unitPrice* is updated (line 40-42). From this example, we can see:

a) Organizational requirements: Source-code of business and application tiers is tangled and, therefore, the roles of programmers cannot be decoupled. Programmers play the business tier developer role: when they need to write Create, Read, Update, Delete (CRUD) expressions (line 32); when they need to create the environment to execute them (line 33-35) and when they are requested to master the database schema (line 37,38,41). They play the application tier developer role when they use the application data and the retrieved data (line 32, 37-41).

b) Runtime requirements: If any modification occurs in the established access control policies leading to maintenance activities at the level of the business logics, there is no other possibility than making them manually and in advance. For example if an attribute of the returned relation is no more authorized to be selected, it will entail a modification on the Select expression and also on the source code.

In reality, CLI were not devised to address any of the presented drawbacks. CLI were mainly devised to tackle the impedance mismatch [4] issue.

```

30 void updUnitPrice(List<Integer> productId, List<BigDecimal> unitPrice) throws SQLException{
31     for (int n=0; n<productId.size(); n++){
32         sql="Select * from Products where productID=" + productId.get(n);
33         st=conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
34                                 ResultSet.CONCUR_UPDATABLE);
35         rs=st.executeQuery(sql);
36         if (rs.next()) {
37             productName=rs.getString("ProductName");
38             supplierID=rs.getInt("SupplierID");
39             // ... more code
40             uPrice=unitPrice.get(n);
41             rs.updateBigDecimal("unitPrice", uPrice);
42             rs.updateRow();
43             // ... more code

```

Figure 1. Typical usage of CLI (JDBC)

Solution: To tackle these CLI drawbacks, a research has been carried out in the context of Component-Based Software Engineering [5], [6]. Component-based development aims at composing software units from other pre-built software units. At the end, a final software system is not built as a unique block but as a composite of software units known as components [7]. A key aspect for the success of any component is its capability of being adapted to be reused [8], which is improved if another key aspect is also considered: the reuse of computation [9]. Thus, this research leverages component-based development to create an reference architecture for building adaptable business tier components (ABTC) aimed at addressing different organizational and contextual runtime requirements. Organizational requirements are addressed by providing different possibilities and combinations to develop and use business tier components. Runtime requirements are addressed through automated tools capable of building and compiling, at runtime, source code from metadata. The services provided by ABTC are closely aligned with CLI to keep their advantages.

This paper is organized as follows: section 2 presents the related work; section 3 introduces the needed background; section 4 presents the reference architecture for ABTC; section 5 presents a proof of concept and, finally, section 6 presents the final conclusion.

2. Related Work

Beyond CLI, such as ODBC [10], JDBC and ADO.NET [11], several tools have been devised to improve the development process of business tiers. From them, Object-to-Relational mapping (O/RM) tools [12], [13], such as LINQ [14], Hibernate [15], [16], Java Persistent API (JPA) [17] and Ruby on Rails [18], have had a significant acceptance in the academic and commercial forums. Other tools, such as embedded SQL [19] and SQLJ [20], have achieved some acceptance in the past. Others were suggested but without any general known acceptance, such as Safe Query Objects [21] and SQL DOM [22]. Next follows a brief discussion about these tools.

O/RM tools were designed to create static representation models of relational database schemas in the object-oriented paradigm. Static models are built in a first stage, eventually by a database administrator, and then programmers start the development process. The basic units of the static representation models are classes (entities), each one representing a database table. Through these entities, programmers read data from tables, update data, insert new data and, finally, delete existing data. To support explicit CRUD expressions, O/RM tools provide language extensions and proprietary SQL languages. Despite these advantages, O/RM present four main drawbacks: 1) they induce an additional overhead when compared to CLI; 2) they were not devised to address the power and the expressiveness of the SQL language and 3) they provide a set of extended functionalities (support for native and proprietary SQL languages, and language extensions), which promote the tangling of source-code of business tiers with source-code of application tiers and 4) they are not geared to address organizational or runtime needs. In spite of these disadvantages, O/RM are powerful tools when the key aspect are typed-objects perspectives of database tables' schemas.

Safe Query Objects combine object-relational mapping with object-oriented languages to specify queries using strongly-typed objects and methods, relieving programmers from writing traditional CRUD expressions. They rely on Java Data Objects [23] to provide strongly-typed objects and also to provide data persistence. Safe Query Objects are a promising technique to express queries but they also convey the presented CLI drawbacks. The only exception is the need for writing CRUD expressions. Moreover, although joins can be used for filters, the result is always of a single typed object - there is no possibility to project more than one table. This constraint definitely prevents the use of Safe Query Objects in complex database applications.

SQL DOM generates a Dynamic Link Library containing classes that are strongly-typed to a database schema. These classes are used to build dynamic CRUD expressions without manipulating any strings. Similarly to Safe Query

Objects, SQL DOM does not tackle the presented CLI drawbacks and its performance exhibits very poor results.

Aspect-oriented programming community considers persistence as a crosscutting concern [24]. Several works have been presented but none addresses the points here under consideration. The following works are emphasized: [25] is focused on separating scattered and tangled code in advanced transaction management; [24] addresses persistence relying on AspectJ; [26] presents AO4Sql as an aspect-oriented extension for SQL aimed at addressing logging, profiling and runtime schema evolution. It would be interesting to see an aspect-oriented approach for the points herein under discussion.

The authors of this paper addressed two issues regarding the use of CLI. In [27]–[29] the work is focused on defining different architectures for devising reusable business tier components. In [30], [31] the work is focused on defining architectures to enforce access control policies on business tier components. The research herein presented leverages previous researches to address a new research challenge: how to devise multi-functional business tier components aimed at coping with different organizational and contextual runtime requirements.

3. Background

In this document, the term CLI is used with a wider scope than the one defined by ISO/IEC [32]. Herein, CLI concept is used to refer to any API/standard with identical features and characteristics to the standard emanated from ISO/IEC. In this context, other related API have also been devised, such as JDBC. Other tools/frameworks have also been devised to ease the development process of business tiers, which use CLI as the underlying middleware to interact with RDBMS. Some of those examples are: ADO.NET, JPA and Hibernate. Next follows a brief description about the main features of CLI that are relevant for this research.

Local Memory Structures: Local memory structures (LMS) are instantiated at runtime to manage the data retrieved by Select expressions. Figure 2 presents a general LMS containing 5 rows (1 to 5) and 6 attributes (A, B, C, D, E, F). This LMS could have been instantiated to manage the data returned by the following CRUD expression: *Select A, B, C, D, E, F from Table....* In this case, the CRUD expression has returned 5 rows and the current selected row is row number 3. Two representatives of LMS are ResultSet [33] for JDBC and RecordSet [34] for ODBC. The access to LMS attributes is accomplished by selecting a row and then, through an index or through a label (usually the attribute name), by selecting one attribute at a time. For example, to execute an action *action* (read, insert or update) on attribute C of row 3 it is necessary to: a) Select row 3; b) Execute *action* (index of attribute C) or *action* (label of attribute C).

	A	B	C	D	E	F
1						
2						
3						
4						
5						

← Selected row

Figure 2. LMS with 5 rows (tuples) and 6 attributes (A till F).

CLI provide protocols to allow applications to scroll on LMS, to read their contents and to alter (insert, update, delete – only for updatable LMS) their internal content, see Figure 1. Other services are also available but they are not relevant at this point.

Functionalities: Only functionalities of CLI directly related to the execution of CRUD expressions will be addressed in this section. Services such as those for managing connections to host databases are not addressed in this paper. Main services of CLI are organized in four main categories: execution, scrollability, updatability and transactions.

- **Execution:** Execution comprises services related to the execution of native CRUD expressions. CLI deal differently with Select expressions from the other three types of CRUD expressions. Select expressions instantiate LMS, while the other types do not. These latter types (Insert, Update and Delete) generate a value indicating the number of affected rows in the database.
- **Scrollability:** Scrollability comprises services related to the scrolling process on LMS. There are several different implementations but two are emphasized. They are mutual-exclusive and are herein known as: 1) *forward-only* – in this case it is only possible to move forward one row at a time; 2) *scrollable* – in this case it is possible to move in any direction and jump several rows at a time.
- **Updatability:** Updatability comprises services organized in protocols to interact with data contained in LMS. There are several implementations but two are herein emphasized. They are mutual-exclusive and are known as: 1) *read-only* – the content of the LMS is read-only and no modifications are allowed; 2) *updatable* – changes can be performed on LMS (insert new rows, update rows and delete rows), which are replicated in the host database after being committed.
- **Transactions:** transactions comprise a set of services to manage database transactions.

Access Modes of CLI: From descriptions previously presented, we see that CLI provide two different modes to access data, which are herein referred to as the Direct Access Mode and the Indirect Access Mode. The Direct Access Mode is used when programmers use the native SQL

language to write CRUD expressions encoded inside strings, while the Indirect Access Mode is only available through the interaction with data contained on LMS. CLI provide other access modes, such as the execution of CRUD expressions in batch, which will be considered in a future work.

4. ABTC Presentation

This section presents the reference architecture. We start by presenting some fundamental concepts and then we present the reference architecture.

4.1 Fundamental Concepts

CRUD expressions are the basic entities from which business tiers are built. They represent the formalization process used by information systems to interact with data residing in relational databases. Beyond the need to write CRUD expressions, the need for mastering their schema is a key issue to manage their execution cycle. Two examples are given: first, CRUD expressions can use runtime values for parameters and, second, Select expressions return relations. As such, a survey was conducted to define a standard schema, herein known as CRUD Schema, to formalize the management process for the execution of CRUD expressions. Four features were identified to characterize any CRUD expression: the type of CRUD expressions (Select or Update, Insert and Delete), their list of conditions (runtime values for clause conditions), their column list (runtime values for column list of Insert and Update) and, finally, the LMS schema (only for Select expressions). LMS schemas comprise the schema of the returned relation and the protocols to interact with the data they keep. The relevancy of CRUD Schema concept is not restricted to be a formalization method. Another relevant aspect derives from the fact that the relationship between CRUD Schemas and CRUD expressions is 1 to many. An indeterminate number of CRUD expressions can share the same CRUD Schema. Listing 1 shows an example of two CRUD expressions: both are Select expressions, both share the same select list (schema of the returned relation) and both have no values defined at runtime. CRUD expressions sharing the same CRUD Schema are herein known as sibling CRUD expressions. The concept of CRUD schema confines the scope of CRUD expressions to sibling CRUD expressions only. This restriction is acceptable and adequate when a tight biding between CRUD expressions and CRUD Schemas is a requirement. In situations in which this tight biding is not a requirement, the CRUD Schema concept is too restrictive preventing the grouping of CRUD expressions that are not sibling. To overcome this situation, the concept of Business Schema is introduced. A Business Schema formalizes the service to manage a group of CRUD expressions. It is up to the programmer to define the scope of each Business Schema in terms of CRUD schemas to be supported. If required, this concept can be used to minimize

the number of Business Schemas to be made available to manage all CRUD expressions. Jayapandian and Jagadish [35] have concluded that a large number of CRUD expressions “*can potentially be composed from a given set of related schema elements*”. Business Schemas are not Schema elements [36] but their number can be optimized if their approach is used. Thereby, to support this latter perspective and the one with a tight biding between CRUD expressions and CRUD schemas, two approaches are proposed to devise and use Business Schemas, which are herein known as the closed approach and the open approach.

Closed approach: in the closed approach, Business Schemas are used to manage only sibling CRUD expressions. The relationship between Business Schema and CRUD Schema is one to one, this way conveying a complete schema awareness of each CRUD expression. As a disadvantage, each Business Schema is not flexible to accommodate CRUD expressions with different CRUD Schemas.

Open approach: in the open approach, Business Schemas are driven by the service to be provided and not by a CRUD Schema to be supported. As such, the open approach is intended to create a wide range service with the capacity of managing several CRUD Schemas. The relationship between Business Schema and CRUD Schema is one to many. This approach has the advantage of increasing the possibility of supporting new or evolving CRUD Schemas without needing new Business Schemas. The only requirement is that the new CRUD Schema must be contained in the defined Business Schema scope. As a disadvantage, programmers are required to master CRUD Schemas, but not database schemas, to be able to select the needed service portion.

Each Business Schema, either following the open or the closed approach, is implemented by a service herein known as Business Service. Each Business Service represents a typed object responsible for managing CRUD expressions formalized by CRUD Schemas. Anyway, Business Schemas are not part of ABTC. Business Schemas are designed to address specific business needs for each database application. In our proof of concept we will present a Business Schema aimed at promoting business tier components driven by access control policies.

```

-- a simple CRUD expressions
Select p.ProductID,p.ProductName,p.SupplierID
  From Products p
-- a more complex CRUD expression
select p.ProductID,p.ProductName,p.SupplierID
  From Products p, Suppliers s
  Where p.SupplierID=s.SupplierID and
        p.CategoryId=10;
    
```

Listing 1. Sibling CRUD expressions.

4.2 General Architecture

We need to remember that ABTC is aimed at building

business tiers and simultaneously dealing successfully with organizational and runtime requirements. Regarding organizational requirements, we can understand them as being the policies based on which roles are defined for the development process of business tiers. The development process of business tiers comprises two main exclusive roles: database administration role and developer role. These roles can be played by the same group of persons or, alternatively, they can be played by different persons. Both possibilities are valid and widely used. Thus, ABTC needs to provide the possibility of decoupling or not decoupling the two roles. Regarding runtime requirements, information systems are becoming more and more dynamic, pushing adaptation processes to be carried out at runtime. As an example, critical information systems with dynamic access control policies, which lead to modifications on business logics, cannot wait for manual maintenance activities. Maintenance processes need to be carried out immediately at runtime. We have presented two examples only, but much more could be given. In order to address both types of requirements we devise an architecture based on a component with two facets. One facet is here referred to as the dynamic component, *ABTC_Dynamic*, and the other component is here referred to as the static component, *ABTC_Static*. They share the same fundamental architecture. The key difference is that *ABTC_Dynamic* supports an additional functionality that is not supported by *ABTC_Static*. Basically, *ABTC_Dynamic* supports a functionality capable of generating and compiling source code at runtime from metadata provided by other

components. When combined in different ways, *ABTC_Static* and *ABTC_Dynamic* provide different scenarios each one addressing different requirements. Figure 3 presents the three main different scenarios known as Scenario 1, Scenario 2 and Scenario 3. These scenarios intend to address organizational and runtime requirements. To deepen the understanding, we give some additional information about both facts and the context in which they can be used. *ABTC_Dynamic* is used when there is the need to carry out an adaptation process at runtime. The adaptation process consists in adapting and persisting business logics in accordance with a defined architectural model. Persisted business logics are kept in independent components herein known as Business Logic. We emphasize that there is no imposed architectural model. Architectural models are defined on a case by case basis to address specific database applications needs. *ABTC_Static* is used when the adaptation process takes place at an earlier stage and, therefore, there is no need to be carried out again. Basically, beyond a steady part, *ABTC_Static* uses Business Logic components previously built by *ABTC_Dynamic*.

Figure 3 presents three scenarios which explain the different possibilities to address organizational and runtime requirements.

Scenario 1: This scenario is used when the business tier developer role (ADM) and the application tier developer role (Developer) are played by different actors and there is no need to carry out an adaptation process in a later stage.

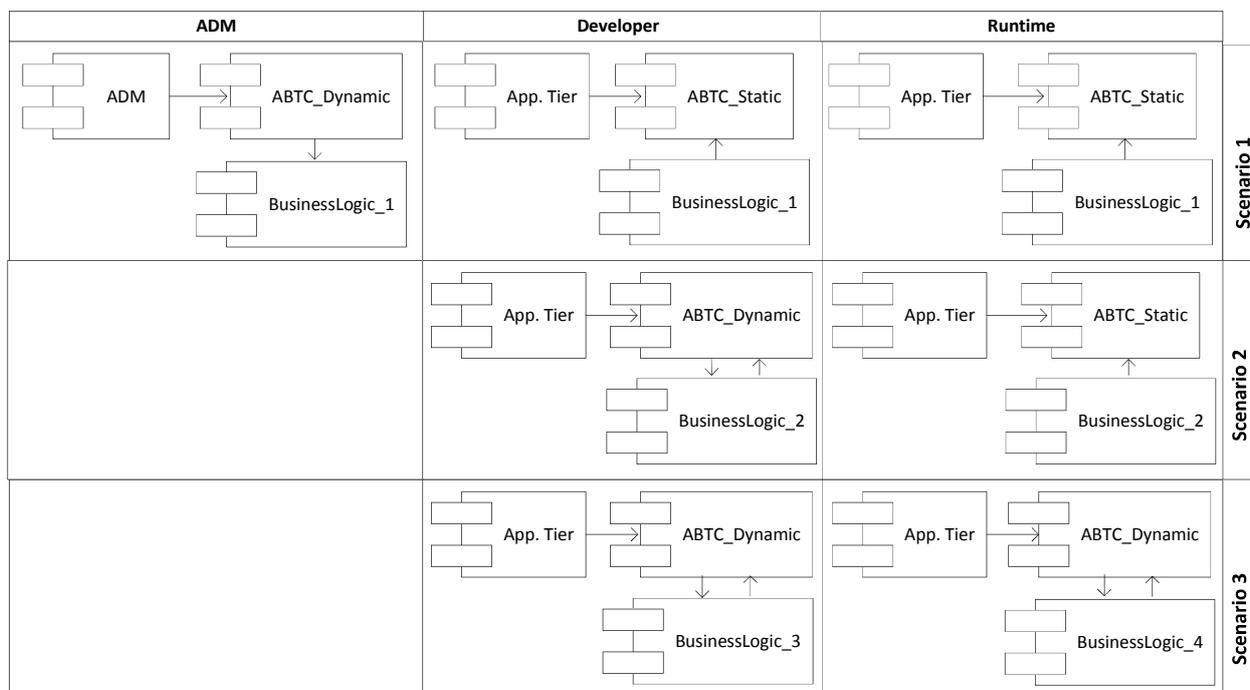


Figure 3. Implemented scenarios to address organizational and runtime needs.

The adaption process is carried out by database administrators (ADM), or someone on their behalf. ABTC_Dynamic is used to build a persisted Business Logic component (BusinessLogic_1), which will be used in the next stages: development process of application tiers and deployment process. Basically, developers of application tiers (Developers) use the persisted Business Logic components (BusinessLogic_1) and ABTC_Static for the development process of application tiers and also for the database applications to be deployed (Deploy).

Scenario 2: This scenario is used when application tier developer role (Developer) and business tier developer role (ADM) are played by the same actor and there is no need to carry out an adaptation process in later stage. The adaptation process takes place during the development process of application tiers (Developer). It uses ABTC_Dynamic to build a persisted Business Logic component (BusinessLogic_2). Then, database applications are deployed with the persisted Business Logic component (BusinessLogic_2) and ABTC_Static.

Scenario 3: This scenario is used, for example, whenever the adaptation process is dynamic and it takes place at runtime after the deployment process. The Business Logic component (BusinessLogic_3) can be built during the development process of application tiers (Developer) but it is not used anymore afterwards. The final adaptation process (BusinessLogic_4) takes place after the deployment process of database applications and at runtime. As an example, the Business Logic can be built in accordance with the profile of the user that is running the database application.

With these scenarios, based on different combinations of ABTC_Dynamic and ABTC_Static, it is possible to cover a wide range of requirements in terms of organizational and runtime needs, this way coping with the goals of this research.

5. Proof of Concept

This section presents the work that has been carried out to prove that ABTC is a reliable architecture to overcome the presented CLI drawbacks. To achieve this goal, we opted for a solution geared to promote business tier components driven by access control policies. We can verify that organizational requirements are addressed by defining who plays which role during development process of business tiers. In other words, by defining who is responsible for implementing the security mechanisms. The runtime needs are addressed by defining security mechanisms based on the running user profiles. Basically, Business Logic components are built and updated at runtime based on user profiles. In our proof of concept, a user profile is defined by a set of Business Schemas and the correspondent CRUD expressions a user is authorized to use. Now we need to design a Business Schema aimed at

representing security mechanisms at the level of business logic.

5.1 Architectural Proposal

This section presents the architecture for our proof of concept, see Figure 4. ABTC comprise several artifacts: *BusinessEngine*, *Business Logic*, *IServiceAllocation*, *IServiceComposition*, *IManager*, *Manager*, *User*, *Session*, *ISession* and, finally, *ITransaction*. The architecture is presented in a single class diagram but it comprises the two fundamental facets: ABTC_Dynamic and ABTC_Static. The only difference between them are the artifacts *BusinessEngine*, *IServiceComposition* and *IServiceAllocation* (used during the adaptation process), which are not part of ABTC_Static. Each artifact is hereafter described:

BusinessEngine: We start by presenting the Business Engine because it is one of the most important entities in our ABTC. It is the component responsible for creating automatically source code (from metadata) for Business Logics components and, therefore, Business Services from Business Schemas. Thus, it is a part of ABTC_Dynamic but not a part of ABTC_Static. The goal and design of Business Engines can change from database application to database application in order to address the specific requirements. In the next sub-section, we will present the Business Schema that is used in the proof of concept.

Business Logic: (at the bottom of the diagram) Business Logic is an independent and persisted container to keep business logics (Business Services and CRUD expressions) built during the adaptation process.

IServiceAllocation: it comprises services to manage the service allocation process which consists in deploying CRUD expressions to be persisted in Business Logic components. We have implemented 3 methods (two for the deployment process and one for removing CRUD expressions) but is up to each system architect to define its own implementation.

IServiceComposition: it comprises services to manage the service composition process, which consists in deploying Business Schemas that are used by Business Engines during the building process of Business Services. We have implemented 3 methods (two for the deployment process and one for removing Business Services) but is up to each system architect to define its own implementation.

IManager: it gathers services to provide one of the two supported versions: dynamic (1) or static (2) versions, ABTC_Dynamic and ABTC_Static, respectively. The dynamic version, beyond extending *IServiceAllocation* and *IServiceComposition*, comprises an additional method to define the repository for the persistent Business Logic.

Manager: it provides two services: 1) a static method (*getInstance*) to create instances of ABTC and 2) implements one of the two versions of *IManager* interface leading to one of the two versions of ABTC:

ABTC_Dynamic and ABTC_Static.

IUser: it provides a service to create sessions. A session is characterized by owning a private connection object to a database and the correspondent transaction management unit (*ITransaction*). Each user opens as many sessions as necessary (*getBusinessSession*).

ISession and Session: they are responsible for managing the instantiation process of Business Services. They provide three methods. The first two (*businessService*) instantiate Business Services: the first one is only for Insert, Update and Delete expressions and the second one is only for Select expressions. They are generics java methods that, among other arguments, accept a Business Schema and return an instance of a Business Service that implements the requested Business Schema. To instantiate Business Services, they need to be loaded into memory at runtime and, then, instances are created using reflection. This process ensures that Business Services may be dynamically created and removed at runtime without raising any runtime

error. The second method opens the possibility to define functionalities of LMS at runtime (read-only or updatable and, forward-only or scrollable). Sessions are released when not needed any more (*releaseBusinessSession*).

ITransaction: it provides all the necessary methods to manage database transactions.

ISession and Session: they are responsible for managing the instantiation process of Business Services. They provide three methods. The first two (*businessService*) instantiate Business Services: the first one is only for Insert, Update and Delete expressions and the second one is only for Select expressions. They are generics java methods that, among other arguments, accept a Business Schema and return an instance of a Business Service that implements the requested Business Schema. released when not needed any more (*releaseBusinessSession*).

ITransaction: it provides all the necessary methods to manage database transactions.

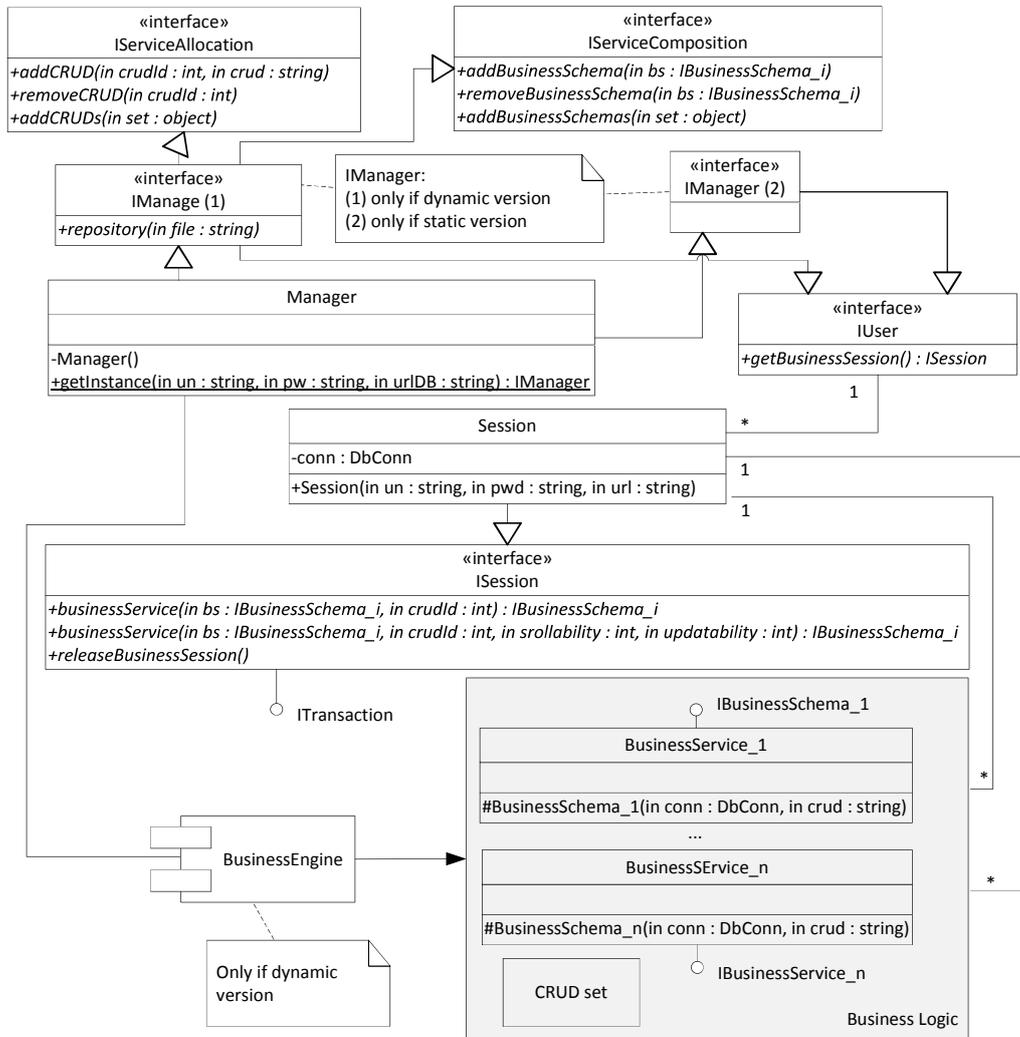


Figure 4. Implemented ABTC class diagram.

5.2 Business Schema and Business Engine Proposals

Now, we need to devise a Business Schema and a Business Engine to be used in our proof of concept. Regarding the Business Schema we leveraged previous researches in this subject [27]–[29]. Then, a Business Engine was also designed and developed. Figure 5 presents the general class diagram for our Business Schemas. It comprises the required entities to completely formalize the key aspects, such as type of CRUD expressions and interactions with LMS. It comprises the following entities:

ILMS: it manages the access to LMS. IRead and IWrite interfaces have getters and setter methods, respectively, to deal with the data contained in LMS. IInsert, IUpdate and IDelete interfaces control the insert, update and delete protocols, respectively. IScroll implements the necessary scrolling policy.

IExecute: it defines one *execute* method with two facets: one for the closed approach and the other for the open approach.

IResult: it is only used with Insert, Update and Delete expressions. It provides a method to retrieve the number of affected rows after the CRUD expression is executed.

IBusinessSchema: it represents the Business Schema to be implemented by a Business Service to manage CRUD expressions. It implements IExecute, ILMS (if the CRUD expression is a Select expression) and IResult (for Insert, Update and Delete expressions only).

BusinessService: it accepts, at instantiation time, an object used to establish a connection to a database server and a CRUD expression to be managed. So, Business Services only at runtime become aware of the CRUD expressions to be managed. Each BusinessService is able to manage several CRUD expressions: sibling CRUD expressions for the closed approach, and the supported CRUD schemas by the Business Schema for the open approach.

One of the biggest challenge was centered on the approach to be followed to formalize Business Schemas. Several approaches were considered, among them XML and standard Java interfaces. In spite of being less expressive than XML, Java interfaces proved to be an efficient and effective approach. The main reasons are the following: programmers do not need to use a different development environment; interfaces are basic entities of any object-oriented programming language and are widely used; interfaces are easily edited and maintained and, finally, Business Schemas have also been defined as interfaces, see IBusinessSchema in Figure 5. These were the fundamental reasons for having opted for Java interfaces in detriment of XML. Thus, BusinessEngine accepts as input, for each Business Service, one interface extending all the necessary interfaces as defined in IBusinessSchema and as shown in Figure 5. Our Business Engines use data formalized by this Business Schema. If the option fell in Business Schemas driven by access control policies, the ones presented in [30], [37] can be used.

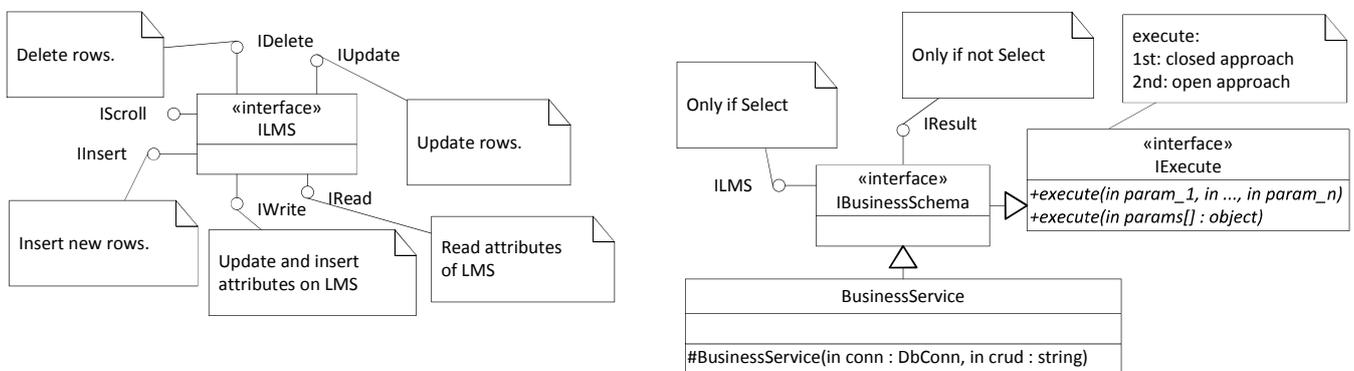


Figure 5. Business Services class diagram for the implemented use case.

5.3 Implementation

In this sub-section we explicitly present how to use the ABTC_Dynamic and also the ABTC_Static. Then they can be combined in order to implement any of the three scenarios, see Figure 3. The use case comprises: 3 users, the set of CRUD expressions and the set of Business Schemas, presented in Table 1 (bottom line presents additional details to understand the table content). The selected set had in mind the main variants to be supported by ABTC. Thus, there are four Business Schemas for the closed approach (IPrd_s, ICat_s, ISup_s and ICat_i) and one Business

Schema for the open approach (IOpen_s). LMS are all of type forward-only and read-only (FR) except one for the CRUD expression ID=4, which is forward-only and updatable (FU). CRUD expressions are all of type Select except the one with ID=8. This way, the different possibilities were defined and used to evaluate the architecture of ABTC. This set is then arranged in 3 smaller sub-sets in order to define 3 different user profiles, which can be dynamically allocated and dislocated to each user.

Now, we present the procedures associated with the adaptation process. We start by presenting the data through structures we have used to formalize the content of Table 1

Table 1. CRUD expressions and Business Schemas for the implemented demos.

ID	CRUD expressions	Business Schemas					LMS
		Closed				Open	
		IPrd_s	ICat_s	ISup_s	ICat_i	IOpen_s	
1	Select * from Products	Y	N	N	Y	Y	FR
2	Select * from Products where ProductID=10	Y	N	N	Y	Y	FR
3	Select * from Products where SupplierID=2	Y	N	N	Y	Y	FR
4	Select * from Categories	N	Y	N	Y	Y	FU
5	Select * from Categories where CategoryID=1	N	Y	N	Y	Y	FR
6	Select * from Suppliers	N	N	Y	Y	Y	FR
7	Select p.*, c.categoryName, c.Description from Products p, Categories c Where p.CategoryID=c.CategoryID	N	N	N	Y	Y	FR
8	Insert into Categories values (?, ?, ?)	N	N	N	Y	N	

ID: CRUD expression identification (1-allFromProducts, 2-fromProducts_productId, 3-fromSuppliers_supplierId, 4-allFromCategories, 5-fromCategories_categoryId, 6-allFromCategories, 6-allFromSuppliers, 7-fromProductsCategories, 8-InsertInCategories)

CRUD expressions: supported CRUD expressions.
Business Schemas: supported Business Schemas.
LMS: - F: forward-only, S:scrollable, R:read-only, U:updatable.

```

11 public interface IRead {
12     public int CategoryID() throws SQLException;
13     public String CategoryName() throws SQLException;
14     public String Description() throws SQLException;
15     public Blob Picture() throws SQLException;
16 }
    
```

Figure 6. IRead interface of ICat_s.

```

10 public interface IExecute {
11     void execute() throws SQLException;
12     void execute(int id) throws SQLException;
13 }
    
```

Figure 7. IExecute interface of ICat_s.

```

8 public interface ICat_s extends IRead, IScroll,
9     IUpdate, IDelete, IWrite, IInsert {
10 }
11 }
    
```

Figure 8. ICat_s definition.

```

14 public class MyInterfaces {
15     private IPrd_s itf1;
16     private ICat_s itf2;
17     private ISup_s itf3;
18     private ICat_i itf4;
19     private ICat_d itf5;
20     private IOpen_s itf6;
21 }
    
```

Figure 9. Class with the authorized Business Schemas.

```

7 public class MyCruds {
8     public static final int allFromProducts=1;
9     private static String crud_1="Select * from Products";
10    public static final int fromProducts_productId=2;
11    private static String crud_2="Select * from Products where ProductID=?";
12    public static final int fromProducts_supplierId=3;
13    private static String crud_3="Select * from Products where SupplierID=?";
14    public static final int allFromCategories=4;
15    private static String crud_4="Select * from Categories";
16 }
    
```

Figure 10. Class with CRUD expressions and their identifications.

```

966 //Instantiate ABTC
967 manager = Manager.getInstance(txt_usr.getText(), txt_password.getText(), url);
968 //Set the name for the Business Logic file.
969 manager.repository("Bes.jar");
970 //Add persistent Business Schemas to the Business Logic
971 manager.addBusinessSchemas(MyInterfaces.class);
972 //Add persistent CRUD expressions to the Business Logic
973 manager.addCRUDs(MyCruds.class);
    
```

Figure 11. Configuration process of ABTC.

and then we show the adaptation process IServiceProvider and IServiceProvider interfaces.

Data Structures for the Service Composition: Figure 6, Figure 7, Figure 8 and Figure 9 partially present the data structures required to accomplish the service composition process for the 3 demos. Service composition process takes place in all the three demos/scenarios, see Figure 3, but in different stages. Figure 6, Figure 7 and Figure 8 present the three interfaces that need to be customized to define the Business Schema for ICat_s: IRead, IExecute and ICat_s, respectively. The same procedure need to be followed for each one of the remaining Business Schemas. From these three interfaces and in accordance with the presented Business Schemas, BusinessEngine automatically builds and compiles the source for the correspondent Business Service. This automated process includes the IWrite interface which is automatically inferred from IRead interface. All other interfaces are shared by all Business Schemas relieving programmers from the need of writing them repeatedly. The service composition process of Business Services is aimed at managing Insert, Update and Delete expressions, only the *execute* method of IExecute interface needs to be customized. Figure 9 presents another relevant class that contains all the 6 Business Schemas. This class is used during the service composition process for users with permissions to use the 6 Business Schemas. During the service composition process Business Services are automatically build and persistently kept by their Business Logic.

Data Structures for the Service Allocation: The service allocation process is focused on deploying the authorized CRUD expressions, in accordance with any policy. The service allocation process uses one class only, see Figure 10, containing the authorized CRUD expressions for the running user profile and also their identifications.

How it Works: Finally, Figure 11 presents the main interaction between the application tier and the business tier (ABTC) during the service composition and the service allocation processes. Please remember that these processes only happen if the component being used is the

ABTC_Dynamic one and that it is not dependent on the running scenario. ABTC is instantiated (line 967), the name for the repository file for the Business Logic is defined (line 969), Business Services are deployed (line 971) and, finally, CRUD expressions are also deployed (line 973). An important aspect is that, if the scenario is the scenario 3, then the content of Business Logic can be modified at any time even after the system has been deployed. There is the possibility to remove existent Business Services and CRUD expressions and also the possibility of adding new Business Services and CRUD expressions.

5.4 ABTC_Static Perspective

Next, we present one perspective which takes place after the adaptation process and, simultaneously, there is no need to any additional adaptation process. Basically, it shows the interaction between application tiers and components based on ABTC that have been already adapted. Figure 12 presents the example shown in Figure 1, but now using ABTC and an updatable IPrd_s. The first note to be emphasized is the similarity between the use of ABTC and JDBC. The main difference between them is that business tiers and application tiers are completely decouple.

Now we emphasize additional aspects of the presented architecture: 1) source-code for business logics is automatically built and driven by any policy and 2) Business Services and pools of CRUD expressions maximize the reuse of computation. The counterparts to obtain all these advantages are: 1) the need to write each CRUD expression only once (which is an unavoidable activity); 2) the need to write an IExecute interface (for Insert, Update and Delete CRUD expressions), or the need to write IRead and IExecute interfaces (for each Select expression) for each Business Schema. We have shown only a part of our experiment in order not to overcrowd the paper with figures. Anyway, tests were carried out with the three demos (covering the three scenarios). From the collected results, it is clear that ABTC completely addresses the goals of this research and particularly for the three scenarios.

```

35     private void updUnitPrice(ISession s, List<Integer> productId, List<BigDecimal> unitPrice)
36     {
37         throws SQLException, BTC_Exception {
38
39         IPrd_s p= s.businessService(IPrd_s.class, MyCruds.fromProducts_productId,
40                                 scrollability.forwardOnly, updatability.updatable);
41         for (int i=0; i<productId.size(); i++) {
42             p.execute(productId.get(i));
43             if (p.moveToNext()) {
44                 productName=p.ProductName();
45                 supplierId=p.SupplierID();
46                 // ... more code
47                 uPrice=unitPrice.get(i);
48                 p.beginUpdate();
49                 p.UnitPrice(uPrice);
50                 p.updateRow();
51                 // ... more code

```

Figure 12. Example shown in Figure 1 but using the ABTC.

6. Conclusion

CLI are used to build business tiers whenever a fine tune control in the interaction with host databases is required. In spite of this advantage, they present some important drawbacks, among which the inability to cope with organizational and runtime requirements. To overcome these drawbacks, this paper presents a multi-purpose architecture based on CLI, herein referred to as ABTC. Two facets of ABTC were defined: the ABTC_Dynamic is aimed at building business logics automatically at runtime and the ABTC_Static is aimed at using business logics previously created by ABTC_Dynamic. This approach promotes the definition of different scenarios to address different requirements. We have implemented a use case with three demos to address some organizational (who plays which role) and contextual runtime needs (reusable business tiers automatically built at runtime). Other scenarios could also be implemented as, for example, to address security requirements. Basically, to address security requirements, the deployment process of CRUD Schemas and CRUD expressions should be driven by access control policies.

The proof of concept here presented is based on Java, JDBC and SQL Server 2008. An ABTC has also been built with C#, ADO.NET and SQL Server 2008. The component was manually built. The achieved success proved that the presented architecture is flexible enough to be used with different technologies. From our previous experience with O/RM tools, namely Java Persistence API, it is our belief that this architecture can also be used. However, it is so easy to be used with CLI that it would only bring disadvantages if used with O/RM, namely because of their induced overhead.

As future work, we plan to complement ABTC with an additional tool to ease the process of creating business logic. At this stage, every CRUD expression and Business Schema is manually written. The tool will automate the building process of some default CRUD expressions and Business Schemas, as O/RM tools do. The automated process uses database schemas to create default select, insert, update and delete expressions and the correspondent Business Schemas for each database table. More complex CRUD expressions and more complex CRUD Schemas will also be supported by a tool based on the one presented in [38], [39]. Additionally, ABTC will be extended to support the remaining access modes of CLI.

It is expected that the outcome of this research may lead to open new approaches to improve the development process of business tiers based on CLI to address different organizational and runtime requirements.

References

- [1] Ó. M. Pereira and R. L. Aguiar, "Multi-purpose Adaptable Business Tier Components Based on Call Level Interfaces," in *ICIS 2015 - 14th IEEE/ACIS International Conference on Computer and Information Science*, 2015, p. accepted.
- [2] W. Cook and A. Ibrahim, "Integrating programming languages and databases: what is the problem?," 2005. [Online]. Available: <http://www.odjms.org/experts.aspx#article10>.
- [3] M. Parsian, *JDBC Recipes: A Problem-Solution Approach*. NY, USA: Apress, 2005.
- [4] M. David, "Representing database programs as objects," in *Advances in Database Programming Languages*, F. Bancilhon and P. Buneman, Eds. N.Y.: ACM, 1990, pp. 377–386.
- [5] G. T. Heineman and W. T. Council, "Component-based software engineering: putting the pieces together." Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [6] C. Szyperky, D. Gruntz, and S. Murer, *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 2002.
- [7] L. Kung-Kiu and W. Zheng, "Software Component Models," *IEEE Trans. Soft. Eng.*, vol. 33, no. 10, pp. 709–724, 2007.
- [8] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *J. Syst. Softw.*, vol. 74, no. 1, pp. 45–54, 2005.
- [9] P. V. Elizondo and K.-K. Lau, "A Catalogue of Component Connectors to Support Development with Reuse," *J. Syst. Softw.*, vol. 83, no. 7, pp. 1165–1178, 2010.
- [10] Microsoft, "Microsoft Open Database Connectivity," 1992. [Online]. Available: [http://msdn.microsoft.com/en-us/library/ms710252\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx).
- [11] G. Mead and A. Boehm, *ADO.NET 4 Database Programming with C# 2010*. USA: Mike Murach & Associates, Inc., 2011.
- [12] W. Keller, "Mapping Objects to Tables - A Pattern Language," *European Conference on Pattern Languages of Programming Conference (EuroPLoP)*. Irsse, Germany, pp. 1–26, 1997.
- [13] R. Lammel and E. Meijer, "Mappings Make data Processing Go 'Round: An Inter-paradigmatic Mapping Tutorial," in *Generative and Transformation Techniques in Soft. Eng.*, 2006, vol. 4143, pp. 169–218.
- [14] M. Erik, B. Brian, and B. Gavin, "LINQ: Reconciling Object, Relations and XML in the .NET framework," in *ACM SIGMOD Intl Conf on Management of Data*, 2006, p. 706.
- [15] C. Bauer and G. King, *Java Persistence with Hibernate*. Manning, 2007.

- [16] B. Christian and K. Gavin, *Hibernate in Action*. Manning Publications Co., 2004.
- [17] D. Yang, *Java Persistence with JPA*. Outskirts Press, 2010.
- [18] D. Vohra, "CRUD on Rails - Ruby on Rails for PHP and Java Developers," Springer Berlin Heidelberg, 2007, pp. 71–106.
- [19] J. W. Moore, "The ANSI binding of SQL to ADA," *Ada Lett.*, vol. XI, no. 5, pp. 47–61, 1991.
- [20] A. Eisenberg and J. Melton, "Part 1: SQL Routines using the Java (TM) Programming Language," *American National Standard for Information for Technology Database Languages - SQLJ*. International Committee for Information Technology, 1999.
- [21] R. C. William and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in *27th Int. Conf. on Software Engineering*, 2005, pp. 97–106.
- [22] A. M. Russell and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in *27th Int. Conf. on Software Engineering*, 2005, pp. 88–96.
- [23] Oracle, "Java Data Objects (JDO)," 2011. [Online]. Available: <http://www.oracle.com/technetwork/java/index-jsp-135919.html>.
- [24] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications, 2003.
- [25] J. Fabry and T. D'Hondt, "KALA: Kernel Aspect Language for Advanced Transactions," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1615–1620.
- [26] T. Dinkelaker, "AO4SQL: Towards an Aspect-Oriented Extension for SQL," in *8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11)*, 2011, pp. 1–5.
- [27] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Reusable Business Tier Architecture Driven by a Wide Typed Service," in *ICIS 2013 - 12th IEEE/ACIS International Conference on Computer and Information Science*, 2013, pp. 135–141.
- [28] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "ABC Architecture - A New Approach to Build Reusable and Adaptable Business Tier Components Based on Static Business Interfaces," *Eval. Nov. Approaches to Softw. Eng.*, vol. 275, pp. 114–129, 2013.
- [29] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Reusable Business Tier Components Based on CLI and Driven by a Single Wide Typed Service," *IJSI - Int. J. Softw. Innov.*, vol. 2, no. 1, pp. 37–60, 2014.
- [30] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "ACADA - Access Control-driven Architecture with Dynamic Adaptation," in *SEKE'12 - 24th Intl. Conf. on Software Engineering and Knowledge Engineering*, 2012, pp. 387–393.
- [31] Ó. M. Pereira, R. Aguiar, and M. Santos, "BTA: Architecture for Reusable Business Tier Components with Access Control," in *ICCSA - 12th Int. Conf. on Computer Systems and Applications*, 2012, vol. 7335, pp. 682–697.
- [32] ISO, "ISO/IEC 9075-3:2003," 2003. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=34134.
- [33] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Assessment of a Enhanced ResultSet Component for Accessing Relational Databases," in *ICSTE-Int. Conf. on Software Technology and Engineering*, 2010, pp. V1:194–201.
- [34] Microsoft, "RecordSet (ODBC)," *Real-Time Syst*, 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/5sbfs6f1.aspx>.
- [35] M. Jayapandian and H. V Jagadish, "Automated creation of a forms-based database query interface," *Int. Conf. Very Large Database*, vol. 1, no. 1, pp. 695–709, 2008.
- [36] C. Yu and H. V Jagadish, "Schema summarization," *32nd Intl Conf on Very large data bases. VLDB Endowment*, Seoul, Korea, pp. 319–330, 2006.
- [37] Ó. M. Pereira, R. L. Aguiar, and M. Y. Santos, "Runtime Values Driven by Access Control Policies Statically Enforced at the Level of the Relational Business Tiers," in *SEKE'13 - Intl. Conf. on Software Engineering and Knowledge Engineering*, 2013, pp. 1–7.
- [38] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms," in *ICSEA 2010 - Int. Conf. on Software Engineering and Applications*, 2010, pp. 114–122.
- [39] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "CRUD-DOM: A Model for Bridging the Gap Between the Object-Oriented and the Relational Paradigms - an Enhanced Performance Assessment Based on a case Study," *Int. J. Adv. Softw.*, vol. 4, no. 1&2, pp. 158–180, 2011.



Óscar Mortágua Pereira graduated in 1983 in Electronics and Telecommunications Engineering at the University of Aveiro (UA). He devoted several years to research and development activities in several private companies. In 2013 he completed the doctoral program MAP-i and obtained a PhD degree in computer science. Since then he is an Auxiliary Professor in the Department of Electronics, Telecommunications and Informatics at the UA. He is currently a researcher at the Telecommunications Institute in Aveiro. His main research activities are

focused on Software Engineering, Access Control, Databases, Big Data and IoT. He has published more than 30 papers in international conferences, international journals and book chapters. He is also participating in several Technical Committee Programs of international conferences and international journals.



Rui L. Aguiar is a Full Professor at the University of Aveiro where he received his PhD degree in 2001 in electrical engineering. He has been an adjunct professor at the INI, Carnegie Mellon University and is currently a Visiting Research Fellow at Universidade de Uberlandia. His current research interests are centered on the

implementation of advanced communication systems in the areas of 5G and Future Internet. He has served as Technical and General Chair of several conferences and is Associate Editor of several journals, having more than 400 published papers. He is a member of ACM and a senior member of IEEE.