# AN ADAPTABLE BUSINESS COMPONENT BASED ON PRE-DEFINED BUSINESS INTERFACES

Oscar M. Pereira, Rui L. Aguiar
*Instituto de Telecomunicações, University of Aveiro, Aveiro, Portugal*

Maribel Yasmina Santos
*Algoritmi Research Center, University of Minho, Guimarães, Portugal*

Keywords: Component-based software, Adaptability, Business tier, Databases, Impedance mismatch.

Abstract: Object-oriented and relational paradigms are simply too different to bridge seamlessly. Architectures of database applications relying on three tiers need business tiers to bridge application tiers and database tiers. Business tiers hide all the complexity to convert data between the other two tiers easing this way programmers' work. Business tiers are critical components of database applications not only for their role but also for the effort spent on their development and their maintenance. In this paper we propose an adaptable business component (ABC) able to manage SQL statements on behalf of other components. Other components may create in run-time a pool of SQL statements of any complexity and delegate their management to the ABC component. The only constraint is that the SQL statements schema must be in conformance with one of the predefined schemas (interfaces) provided by the ABC component. The main contribution of this paper is twofold: 1) the presentation of an adaptable business component and 2) to show that the ABC source code may be automatically generated. The main outcome of this paper is the evidence that the ABC component is an effective alternative approach to build business tiers to bridge object-oriented and relational paradigms.

## 1 INTRODUCTION

Good programming practices advise the development of database applications relying on a multi-tier architecture. The three tier architecture is the most widespread one comprising the application tier, the database tier and the middle tier known as the business tier. The business tier may provide a clear separation (technological, business and administrative/administration) between host databases and client applications (CA). When database tiers and application tiers rely on different paradigms, as the relational and object-oriented, respectively, business tiers are responsible for relieving programmers of client applications from several critical issues being *impedance mismatch* (David, 1990) the most noticeable one. Impedance mismatch is an outcome of the diverse foundation of both paradigms raising a major hindrance for their integration, being an open issue for more than 50 years (Cook and Ibrahim, 2006). Despite their advantages, business tiers present some weaknesses among them we emphasize their inertia to evolve in consequence of maintenance needs. These needs may have their origin in the need for new queries, the need to update existent queries or changes in the database schema. Inertia may reach increased relevancy if SQL statements are wrapped into classes with improved usability to ease their usage by programmers of client applications. In this case, maintenance activities will not only comprise lower-level issues as writing or re-writing the SQL queries but will also comprise the development or maintenance of the involved wrappers to keep their usability (examples: getter and setter methods). Figure 1 shows a simple example of two interfaces to wrap the SQL statement *Select id, fName, lName, grade from ...*. A study reported in (Keene, 2004) shows that business tiers may consume up to 30%-40% of the total effort of a project. The difficulties to build and maintain business tier components may have a shelter on the Component-Based Software Engineering (CBSE) (Heineman and Councill, 2001) subject. CBSE is widely recognized as a sub-discipline of Software Engineering to build complex systems. The main goals of CBSE are threefold: 1) to provide guidelines for the development of systems as assemblies of components; 2) to provide guidelines for the development of components as reusable artifacts and finally 3) to provide guidelines for the maintenance of systems through the adaption and replacement of their constituent components. Using commercial off-the-shelf (COTS) software components to build software systems may be seen as a goal for many system architects. Unfortunately, component reutilization may raise several technological difficulties and, maybe not least important, may easily gather voices against its adoption. In reality, despite the relevancy of the CBSE postulates, several issues are difficult to tackle as the replacement and adaptation of components. Component replacement has some disadvantages conveying an impact on the overall system. Some of the disadvantages are (Costa et al., 2007): 1) the state of the replaced component may be lost; 2) component or even system availability may be affected; 3) performance decay during the replacement process – additional power computation is required. In order to avoid the replacement of components, components must be able to adapt dynamically at run-time, which is one of the crucial aspects of CBSE (Bracciali et al., 2005). The adaptation of components should comprise not only the configuration process but mainly the replacement of old services and also the definition of new services in a seamlessly way.

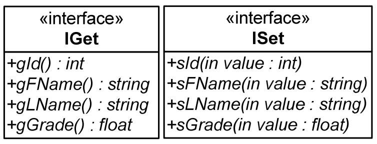


Figure 1: Example of interfaces to wrap a business entity.

In this paper we are focused on a reusable business tier component for database applications where client applications are developed in the object-oriented paradigm and the host database relies on the relational paradigm. The component is herein known as Adaptable Business Component (ABC). ABC component pretends to achieve the following three main goals: to comply with full expressiveness of SQL, to provide an enhanced usability from client applications point of view and to provide supervised adaptability to SQL statements deployed in run-time. Full SQL expressiveness is achieved by using Call Level Interfaces (CLI) (ISO, 2003) as a low-level API to communicate with the host database. Call Level Interfaces will be addressed with some detail in Section 4. Usability is assured by the implemented interfaces to communicate with client applications. These interfaces are based on the schema of the SQL statements and are also type-safe. This issue will be addressed with more detail in Section 5. Supervised adaptability is assured by the ability to dynamically, in run-time, as a server component, receive messages from "*authorized*" entities to accept new, remove existent and update existent SQL statements of any complexity and, on behalf of client applications, manage their execution. The results of their execution are accessible to client applications through the aforementioned interfaces. This issue will be addressed with more detail in Section 5 and Section 6. Figure 2 presents a general view of the interaction between ABC components and other client entities. Authorized entities may create and update a pool of SQL statements in run-time and delegate their management to the ABC component. Then client applications may ask the execution of SQL statements through the interfaces provided by the ABC components.

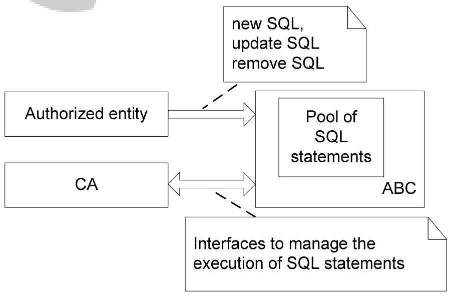


Figure 2: General view of CA and ABC interactions.

In this paper we are focused on defining a strategy based on software product lines for business tier components in the context of CBSE. Software product lines address specific needs of a domain analysis and are developed from a set of assets that share the commonality of particular features (Clements and Northrop, 2001). Some of the main advantages of software product lines approach are: reduced time-to-market (Hetrick et al., 2006), reduced cost (Pohl et al., 2005) and improved quality (Pohl et al., 2005, Hetrick et al., 2006). Product-lines enable more effective component reuse (Griss, 2000).

It is expected that the outcome of this paper will contribute to open a new approach to the development of business tier components.

Throughout this paper all examples are based on Java, SQL Server 2008 and JDBC for SQL Server (sqljdbc4.jar). Code may not execute properly since we only show the relevant parts for the points under discussion.

The paper is structured as follows. Section 2 presents the motivation; Section 3 presents the related work; Section 4 concisely presents the JDBC; Section 5 presents ABC architecture; Section 6 presents ABC Life-Cycle and Section 7 present the final conclusion and future work.

## 2 MOTIVATION

Database applications of some complexity may comprise hundreds of SQL statements to deal with business requirements. This leads to situations where the development and maintenance processes of business tiers are very tedious and exhaustive. Programmers are pushed to write similar source code for each SQL statement. Moreover, Select statements with a long select-list also convey to a repetitive source code around each element of the select-list in order to read its value. There should exist a methodology to relieve programmers from these tedious, exhaustive and error-prone processes.

```
-- a simple query
Select p.id,p.fName,p.lName
  From pilot p

-- a more complex query
select p.id,p.fName,p.lName
  From pilot p,circuit c,classif f
  Where p.id=f.id and
        f.date=c.date and
        f.position between 1 and 3
  Group by p.id,p.fName,p.lName
  Having count(f.position)=
           (select count(*) from ...)
Union
Select ...
   ...
   Order by ...
```

We may classify SQL statements in two orthogonal groups: by complexity and by schema. Complexity says if a SQL statement is simple or complex. The schema characterizes each SQL statement in terms of the schema of its parameters and the schema of the returned relation (only for Select statements). An SQL statement may be simple and have a simple or a complex schema or an SQL statement may be complex and have a simple or a complex schema. Moreover, several SQL statements, simple or complex, may share the same schema. The two queries previously presented share a simple schema. The first query is very simple and the second query is not very simple. The code to execute the query and to read the returned data is the same for both queries. This evidence raises the following question: if several SQL statements may share the same schema why not make use of it to optimize the source code editing process? The first question to be put is: "*Have we been spreading boilerplate code in business tiers?*". Another relevant issue is the access and manipulation of data/information/ knowledge kept in databases. While the issue previously discussed was essentially technological, the access and manipulation of data/information/knowledge concerns the soul of companies. Very often data/information/knowled-ge is the most important asset in companies conveying an unavoidable need to completely control and protect it.

In this paper we present the ABC component relying on an architecture aimed at coping with two important features: 1) Re-use of source code to manage different SQL statements, simple or complex, defined after ABC deployment (the unique constraint is the common schema they must rely on). 2) To follow the separation of concerns regarding the use of ABC components by programmers of client applications from all other issues related to the development, configuration and administration processes.

## 3 RELATED WORK

In (Schmoelzer et al., 2006) is presented a concept for model-typed interfaces concept relying on generic interface parameters. These parameters are characterized as Model-defined Types whose schema is defined by a Data Model. The authors claim that by this way complex data structures (based on Data Models) may be transferred between components in a single method invocation avoiding successive calls to accomplish the same task. This methodology is very useful when two conditions occur simultaneously: 1) the involved components do not share the same working address space and 2) the component playing the client role has full control and knowledge about the amount of data being transferred. In this work ABC components share the same address space as client applications and the access to the returned data (from Select statements) is to be implemented in attribute by attribute and row by row basis. This work could profit from (Schmoelzer et al., 2006) if or when ABC components and client application run in different address spaces.

Data Transfer Objects (Flower, 2002) is a design pattern used whenever an entity gathers a group of attributes that must be accessed in a swift way. Accessing those attributes one by one through a remote interface raises several disadvantages such as the increase of the network traffic, latency is increased, performance is negatively affected, demand on server and client processing is increased. Data Transfer Objects are tailored to address these situations. They are organized in serializable classes gathering the related attributes and forming a composite value. An entire instance of the serialized object is transferred from the server to the client. This approach is quite similar to the previous conveying the same disadvantages.

O/RM tools (Hibernate (Bauer and King, 2007), TopLink (2011), LINQ (Erik et al., 2006)) are powerful tools to integrate object-oriented applications with relational databases. Their extended functionalities are mostly used to build persistent business tiers relying on object to relational mapping techniques. They may also support native queries, proprietary SQL language, language extensions and other relevant tools to ease programmers' work. Their scope is too wide and deeply diverges from the scope of this work. Anyway, if required, ABC components may be developed with and above O/RM frameworks. Their services have to be coordinated, integrated and wrapped by the architecture herein presented.

## 4 JDBC

One of the key requirements of ABC component is the ability to execute any SQL statement, very simple ones and very complex ones. Before this requirement, the option for a database driver (DB Driver) is a key issue. The choice fell upon low-level APIs being Call Level Interfaces an important candidate. Call Level Interfaces are considered important options whenever performance and SQL expressiveness are considered key issues (Cook and Ibrahim, 2006). Call Level Interfaces provide mechanisms to encode Select, Insert, Update and Delete SQL expressions inside strings, easily incorporating the power and the full expressiveness of SQL. JDBC (Microsystems, 2008) and ODBC (Microsoft, 2010) are two of the most relevant Call Level Interfaces. We will explore JDBC as representative of Call Level Interfaces. JDBC has two main interfaces to manage the execution of SQL statements: the Statement interface (Microsystems, 2010b) and the ResultSet interface (Microsystems, 2010a). The Statement interface is used to execute SQL statements and to return the possible results they produce (only for Select statements). The returned results are managed by the ResultSet interface. Loosely speaking, ResultSet interface provides two orthogonal functionalies: *scrollability* and *updatability*. Scrollability defines the ability to scroll over the retuned relation from the database. There are two possibilities: *forward-only* – in this case cursors may only move forward one row at a time; *scrollable* – in this case cursors may move in any direction and jump several rows at a time. Updatability defines the capacity to change the in-memory data managed by the ResultSet interface and therefore the content of the host database. There are two possibilities: *read-only* – the content of the ResultSet is read-only and, no changes are allowed; *updatable* – changes may be performed over the in-memory data (insert new rows, update current rows and delete rows). These functionalities are defined at instantiation time of the parent Statement (or PreparedStatement) entity and will be incorporated and made available in the ABC architecture.

## 5 ABC ARCHITECTURE

The main goal of this paper is to present a component, known as ABC component, with the ability to manage and execute a pool of SQL statements on behalf of client applications. The pool of SQL statements is dynamically updated in runtime by an external authorized entity. From client applications (CA) point of view ABC components always play the role of server components. From host database point of view ABC components always play the role of client components. Figure 3 presents a possible deployment of database applications relying on ABC components. The communication between client applications and ABC components are always through predefined interfaces. The static architecture of ABC components comprises three main blocks: Business Manager (BM), Business Entities (BE) and the Database Driver (DB Driver), as shown in Figure 4. The DB Driver is the block responsible for providing the required services to manage the communication with a local or remote database management system (DBMS).

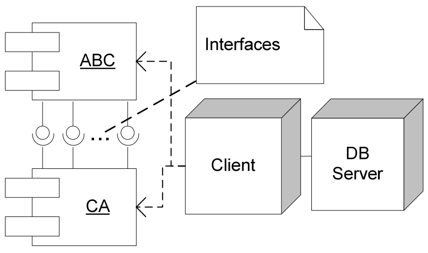


Figure 3: Possible deployment for database applications.

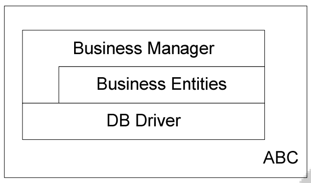


Figure 4: ABC main blocks.

The services include, but not exclusively, the management of connections to the database and the execution of SQL statements. Business Entities are responsible for the definition of contracts between ABC components and client applications. Each contract is specified by one interface. Business Manager is responsible for the management of the local pool of SQL statements. Business Manager also manages a local pool of active connections with the database in order to improve the overall system performance. This functionality demands the interaction between the Business Manager block and the DB Driver block. The connection pool is out of the scope so it will not be addressed in this paper.

## 5.1 DB Driver

DB Driver is responsible for providing internal services to ABC components in order to ease their communication with the host DBMS. The choice for the specific DB Driver depends on several issues as the host DBMS and the host programming language of the client application. In this paper the programming language is Java and the host DBMS is SQL Server 2008 and, therefore, we will use sqljdbc4.jar, which is an implementation of JDBC for SQL Server. In this paper we will present ABC components architecture coping with the following functionalities of ResultSet: forward-only and read-only, forward-only and updatable, scrollable and read-only and, finally, scrollable and updatable.

## 5.2 Business Entities

Business Entities (BE) are software artifacts responsible for the implementation of contracts (services) between ABC components and client applications. Each Business Entity implements one contract which is specified by an interface known as Business Interface (BI). ABC components may comprise one or more Business Entities. Figure 5 presents a concise class diagram for Business Entities and Business Interfaces. The correspondent dynamic artifacts are the

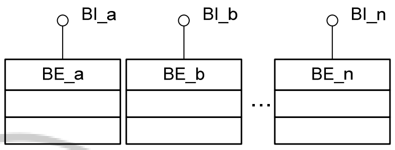


Figure 5: BE and BI.

Business Workers (BW). Business Workers are active entities, in other words, are running instances of Business Entities. A Business Worker accepts one SQL statement whose execution context is in accordance to the Business Interface implemented by its source Business Entity. Thus, Business Entities address the key issue of *reuse* of computation (Elizondo and Lau, 2010). Figure 6 presents the basic relation between Business Entities and Business Workers.

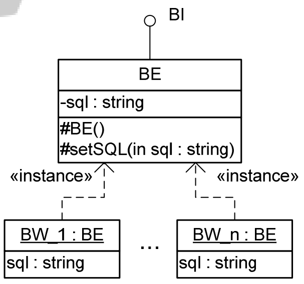


Figure 6: Relationship between BE and BW.

SQL statements are defined through the *setSQL* to allow its definition and updating processes to be carried out after Business Workers instantiation. These processes are managed by Business Manager in a transparent way for client applications.

### 5.2.1 Business Interfaces

A Business Interface is a contract that a Business Entity is committed to implement. Business Interfaces define how client applications and Business Entities may communicate. The schema of a Business Interface is directly dependent on the queries to be processed and, in case of Select statements, on the functionalities to be made available (scrollability and updatability). There are three types of Business Interfaces: 1) Alter Business Interface (A-BI) – for Insert, Update and Delete SQL statements; 2) Select Active Business Interface (SA-BI) – for Select SQL statements that create updatable ResultSets and 3) Select Passive Business Interface (SP-BI) – for Select SQL statements that create read-only ResultSets. Select Active Business Interfaces and Select Passive Business Interfaces implement one of the two scrollability facets: forward-only or scrollable.

Before delving into the Business Interfaces details, let's consider a single table with the following schema:

```
Table(SqlDT1 att_1,
      SqlDT2 att_2,
      ...,
      SqlDTn att_n)
```

where SqlDTn is the SQL data type of the attribute *att_n*. The correspondent data type of SqlDTn in the host programming language is represented by DTn. This table will be used as the basis for the Business Interfaces specification. Each Business Interface may be considered as aggregations of sub-interfaces. Therefore, we will begin the description of Business Interfaces by their elementary sub-interfaces.

The sub-interface *IExecute*, presented in Figure 7, is shared by all Business Interfaces. It comprises only one method. This method is invoked by client applications to trigger the execution of the associated SQL statement.

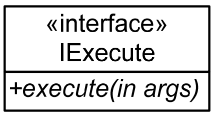


Figure 7: Sub-interface IExecute.

The method may be invoked as often as necessary to re-execute the SQL statement. The argument *args* comprises all the arguments to be used in conditions inside the SQL statement and also as the values to be inserted or updated on tables of the host database. As an example, the following SQL statement:

```
Update Table
  Set (att_1=@v_1,
       att_n=@v_2)
  Where (att_2=@v_3);
```

leads to the following signature:

```
void execute(DT1 v_1,
             DTn v_2,
             Dt2 v_3)
```

The sub-interface *IGet* gathers all necessary methods to read all attributes of one row from the ResultSet in-memory data.

```
Select *
  From Table
```

For the SQL statement the correspondent IGet interface is presented in Figure 8. The method signatures are based on the schema of the Select statement and are also type-safe.

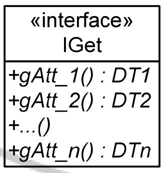


Figure 8: Sub-interface IGet.

These features improve ABC component usability when compared with the standard JDBC API. Users of ABC components are before signatures type-safe and schema oriented easing this way both the understanding of their meaning and the associated data-type.

The sub-interface *ISet* gathers all necessary methods to update/insert data in the ResultSet in-memory. This is only possible if the ResultSet is updatable. For the statement the correspondent ISet interface is presented in Figure 9. The signatures of setter methods are based on the schema of the Select statement and are also type-safe as with the IGet interface. The remaining methods are used to implement the protocols to update and to insert rows.

```
Select *
   From Table
```

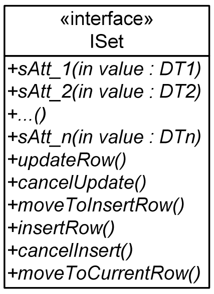


Figure 9: Sub-interface ISet.

The sub-interface *IFowardOnly* comprises all methods associated to the scrolling policy of forward-only ResultSets. Figure 10 only presents the main method which allows the cursor to move one row forward.

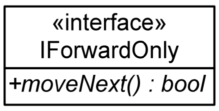


Figure 10: Sub-interface IForward-only.

The sub-interface *IScrollable* gathers all methods associated to the scrolling policy of scrollable ResultSets. Figure 11 only presents three of the main methods.

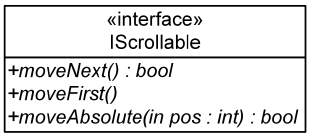


Figure 11: Sub-interface IScrollable.

All the sub-interfaces of Business Interfaces have been individually presented. Now let's present the schema for each Business Interface type. Figure 12, Figure 13 and Figure 14 present the Business Interfaces for A-BI, SP-BI and SA-BI, respectively.

A-BI, shown in Figure 12, only comprises the interface IExecute.

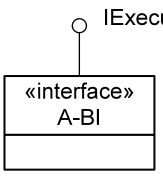


Figure 12: A-BI interface.

SP-BI, shown in Figure 13, comprises three interfaces: IExecute, IGet and, depending on the instantiation of the parent Statement interface, IForwardOnly or IScrollable.

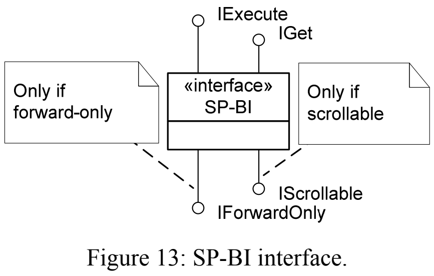


Figure 13: SP-BI interface.

SA-BI, shown in Figure 14, comprises four interfaces: IExecute, IGet, ISet and, depending on the instantiation of the parent Statement interface, IFoward or IScrollable.

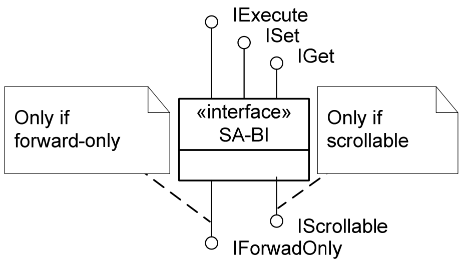


Figure 14: SA-BI interface.

### 5.2.2 Business Workers

Business Entities are classes and Business Workers are running instances of those classes. From each Business Entity it is possible to create as many Business Workers as necessary. Each Business Worker is identified by its type (parent Business Entity) and the SQL statement to be executed. Each SQL statement is uniquely identified by a token. There cannot be two tokens with the same value in the same ABC component instance. Business Workers instantiated from the same Business Entity are called sibling Business Workers. Business Workers running the same SQL statement are called true sibling Business Workers and Business Workers running different SQL statements are called false sibling Business Workers. Examples of two SQL statements managed by the same Business Entity and, therefore, running on false sibling Business Workers could be:

```
Select *
  From Table
  Where att_1=@v_1

Select t1.*
  From Table t1, Table t2
  Where t1.att_1=@v_1 and
        t1.att_2=t2.att_2 and
        t2.att_3=1
```

Despite some restrictions, each Business Entity may support an unlimited set of SQL statements. SQL statements may be as simple or as complex as necessary. The restrictions are only centered on the implemented Business Interface. Particularly the interface IExecute, *execute(args)*, may eventually convey a significant weakness on the Business Entity adaptability. This weakness is felt at the level of the SQL statements *where* and *having* clauses. Anyway, if required, this weakness may be avoided by using the following signature *void execute()*. From now on, the SQL statement parameters, if required, must be set by the client application as shown in the following example:

```
Select *
  From User
  Where Grade>10 and Grade<16 and
       Substring(FName,1,1) like 'A'
```

This strategy may improve Business Entities' flexibility but forbids the use of parameterized queries as pre-parsed queries (prepared statements in JDBC) provoking this way decrease in performance. Another important drawback of this approach is the decrease of ABC usability: IExecute may no longer be used to help programmers on setting up the parameters of SQL statements.

### 5.3 Business Manager

The Business Manager is the entry point of all ABC components. ABC's administrators and programmers of client applications access ABC components functionalities through their entry static methods. Business Manager has a static method for each Business Entity to create a singleton instance of its factory class, as shown in Figure 15. Business Entities factories implement two interfaces, see Figure 16: one is only at administrator's disposal to update the pool of SQL statements and is known as IAdm. The other interface is at common programmers' disposal to create Business Workers and is known as IUser. Each SQL statement to be used by Business Workers is uniquely identified by a token defined by administrators in the *add* method. The class diagrams for the interfaces is shown in Figure 17.

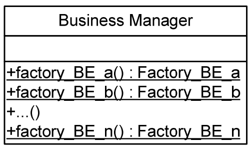

Business Manager
+factory_BE_a() : Factory_BE_a
+factory_BE_b() : Factory_BE_b
+...()
+factory_BE_n() : Factory_BE_n

Figure 15: Business Manager.

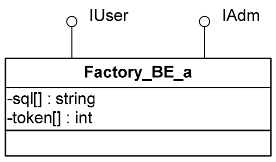


Figure 16: Business Entity Factory.

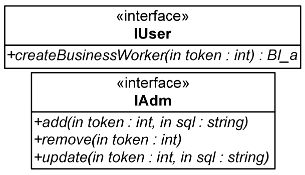

«interface»
IUser
+createBusinessWorker(in token : int) : BI_a
«interface»
IAdm
+add(in token : int, in sql : string)
+remove(in token : int)
+update(in token : int, in sql : string)

Figure 17: Interfaces of Business Entities Factories.

The next blocks of code depicts examples of source code to create SQL statements in the pool and source code to create a Business Worker.

```
// add SQL statement
IAdm a=Manager.factory_BE_a();
a.add(tk_a1,sql_a1);
a.add(tk_a2,sql_a2);
// create Business Worker
IUser u=Manager.factory_BE_a();
BI_ai a=u.createBusinessWorker(tk_a1);
```

## 6 ABC LIFE CYCLE

ABC components comprise two types of software sources: outsource (software from other suppliers) and insource (software developed to ABC components). Table 1 presents the main sub-systems by software source used in our examples. The catalog of ABC components is defined within the context of *idealized component life cycle* (Kung-Kiu and Zheng, 2007). The life cycle is based on the *development for reuse* and *development with re-use* processes in accordance to CBSE principles and considers three phases: *design*, *deployment* and *runtime*.

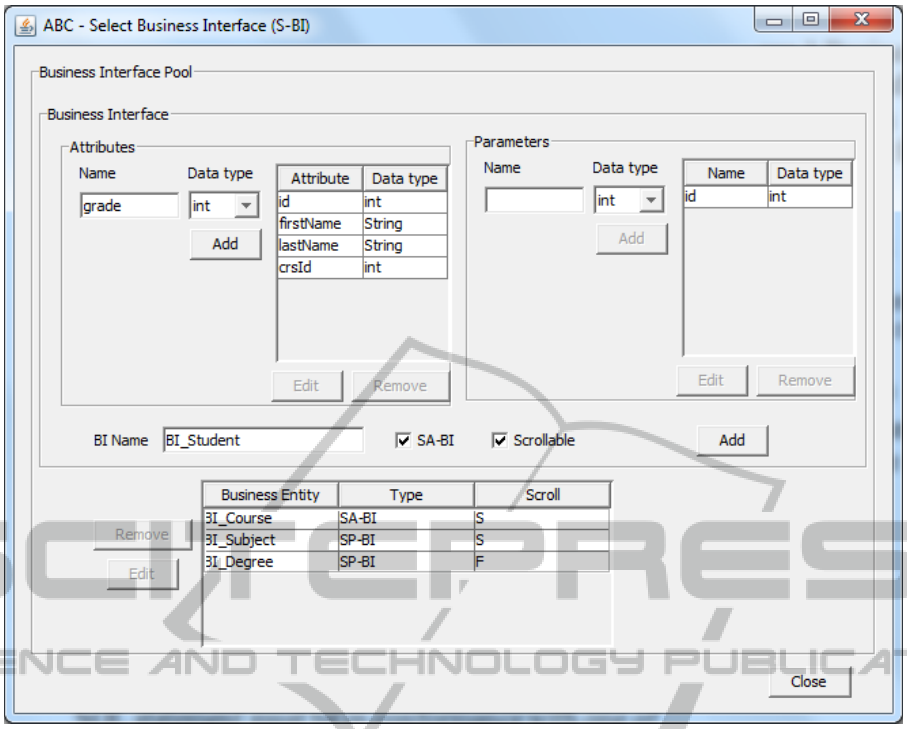
Table 1: ABC sub-systems/source.

| Source | Sub-system |
|---|---|
| outsource | DB driver |
| insource | Manager<br>Business Entities<br>Business Interfaces |

### 6.1 Design Phase

The design phase is focused on the development of ABC components. Developers of ABC components may follow three distinct approaches: global approach, the activity approach or the entity approach. The entity approach is based on the development of ABC components with only one business entity. The activity approach is based on the development of ABC components by each activity such as accounting, clients, suppliers, warehouse, data warehouse and OLAP. The global approach is based on the development of a single ABC component for all activities.

It is also possible to follow any combination of the three approaches. The decision is up to the system administrator. Independently of the chosen approach, each ABC component may deal with several actors where each one plays a specific role conditioned by the queries he may execute.



Attributes — Name: grade; Data type: int

| Attribute | Data type |
|---|---|
| id | int |
| firstName | String |
| lastName | String |
| crsId | int |

Parameters — Name: ; Data type: int

| Name | Data type |
|---|---|
| id | int |

BI Name: BI_Student  [✓ SA-BI]  [✓ Scrollable]

| Business Entity | Type | Scroll |
|---|---|---|
| BI_Course | SA-BI | S |
| BI_Subject | SP-BI | S |
| BI_Degree | SP-BI | F |

Figure 18: Widget to create S-BI.

Different instances of the same ABC component may run different sets of SQL statements for each Business Entity this way promoting its reuse by different actors. Moreover, every software subsystem (SS), such as warehouse management, gathers several activities such as clients, suppliers and orders this way promoting the component reutilization. There is a wide range of possibilities for component reuse: by activity, by actor or any other combination. Business Interfaces cannot be modified, added or removed after the design phase.

They materialize the contract between each individual ABC component and client applications. Client applications trust ABC components to manage SQL statements since each SQL statement is in conformance with one of the implemented Business Interfaces. Any change in the contract after the design phase of ABC components compels the re-opening of the design phase. After the design phase SQL statements may be created as needed (this is the degree of freedom they have) but each SQL statement must be in conformance with one of the available Business Interfaces (this is the restriction they must obey).

The insource code for each ABC component may be generated by a tool as the one shown in Figure 18. Only the GUI used to create Select Business Interfaces is shown. We may see 3 Business Interfaces in the pool (BI_Course, BI_Subject, BI_Degree) and a new one is being edited (BI_Student) and ready to be inserted in the pool. This Business Interface supports, for example, the following SQL statement:

```
Select id,firstName,lastName,
       crsId,grade
   From Student
   Where id=@id
```

Figure 19 shows a partial view of the correspondent BI_Student.

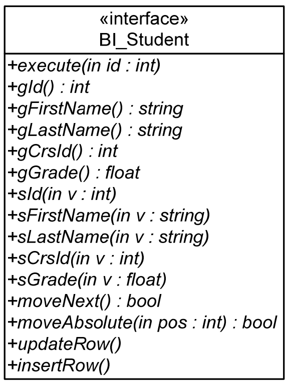


Figure 19: Partial view of BI_Student.

The next code concisely shows the Java source code for the methods *gId()* and *sId(int v)*.

```
ResultSet rs=...
...
public int gId() {
   return rs.getInt(1);
}
public  void  sId(int  v)
{
   rs.updateInt(1,v);
}
```

Alternatively, the insource source may be derived from a general model herein known as the Business Component Model (BCM). In order to automatically generate the insource code the Busuiness Component Model requires the following information:

- For each Business Entity: 1) Its type: alter, select passive or select active; 2) the Business Interface schema; 3) the scrollability policy.
- The source code programming language.
- The host database management system.
- The DB Driver to be used

These two approaches, a tool or the BCM, relieves programmers from writing any source code and therefore to avoid the deployment of ABC components with undesirable errors.

The final stage of the design stage is attained when ABC components are compiled and packed as components ready to be deployed.

As summary, the design phase embodies the process of development for reuse and is mostly focused on the definition of Business Interfaces to be implemented by reusable ABC components. The concrete SQL statements to be deployed in each instance of an ABC component are not defined in the design phase but in a later phase.

## 6.2 Deployment Phase

In the deployment phase, developers use ABC binary components to develop each subsystem of their database applications. Developers may play the role of an administrator developer or the role of a common developer. The former role is used to write source code to update the pool of SQL statements. The latter role is used to write source code for subsystems. Database applications may incorporate one or more subsystems, each subsystem may incorporate one or more activities and each activity may support one or more actors. System administrators may define the strategy for the ABC components reutilization in the deployment phase. Common developers may not know anything about the SQL statements to be made available in each ABC component. All they need to know is the location of the required SQL statement in terms of ABC component instance and Business Entity.

As summary, the deployment phase embodies the process of *development with reuse*.

## 6.3 Runtime Phase

In this phase all components are running. Figure 20 concisely presents a possible running phase of two subsystems, SS_a and SS_b. Figure 20 a) shows two instances of SS_a (SS_a1, SS_a2). Both share the same business components, ABC_y and ABC_w. If SS_a1 and SS_a2 correspond to two different actors, the SQL statements available to each one may be different. Figure 20 b) shows one instance of SS_b (SS_b1). It embodies two shared ABC components with ABC_a (probably with its own sets of SQL statements) and also the ABC_z component.

Remind that the SQL statements are created and updated in run-time through the IAdm interface which is not shown. The administrator role may or may not be protected by some security policy such as authentication and/or authorization to grant access to the configuration process. This topic is out of scope of this work.

## 6.4 Seamless Operation

SQL statements updating process is executed in a seamless way. This means that the process to insert, update or delete SQL statements from the pool of ABC components may be executed without any restrictions. Actually, ABC components assume a passive attitude. They do not provide any service to inform client applications from any relevant or critical occurrences.

These occurrences should be coordinated between client applications and the component that plays the administrator role.

Inserting new SQL statements does not raise any critical question. Subsystems are not allowed to use what still do not exist. The identification token should only be available after inserting the SQL statement into the pool. Updating and removing SQL statements that are being used by one or more Business Workers are the critical situations. When a SQL statement is being used and it is updated or removed, Business Workers keep their states unchanged.

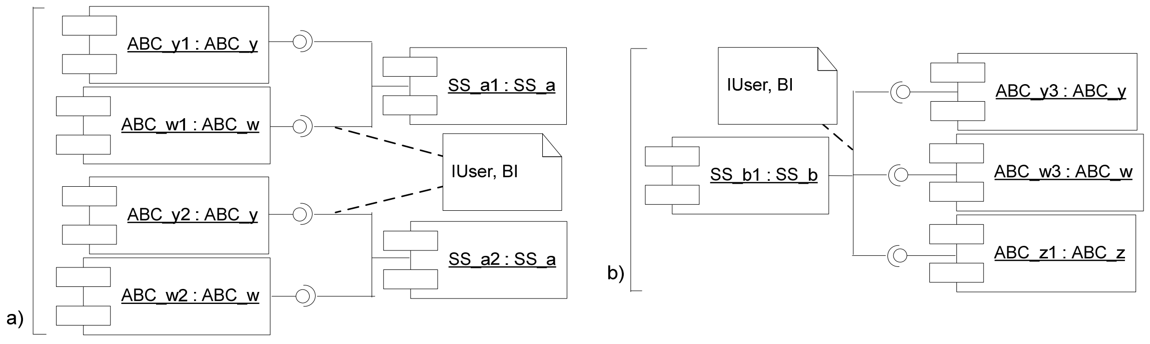


Figure 20: Client application and ABC components deployment.

This assures that client applications may continue their work. Business Workers' state will only be updated when the client application re-invokes the *execute* method. Then, Business Workers will re-execute the most recent SQL statement. The *execute()* method should not be invoked if the SQL statement has been removed from the pool. To prevent any undesirable situation, it is advisable that client applications become aware of the actions taken by the administrator in order to proceed with the most convenient measures. Business Manager does not interfere or change the state of any Business Worker. The states of Business Workers are always under sub-systems' control. Therefore, each application should define its own protocol between administrator and application components.

## 7 CONCLUSIONS

In this paper we have presented an adaptable business component to bridge object-oriented applications and relational databases. ABC components are in line with the context of CBSE supporting the process of development for reuse and development with reuse. The reutilization intensity is determined by the chosen approach for the development of ABC components (global, activity, entity or mixed) and also by the intensity of *reuse* of computation. The ABC component main architecture relies on Business Entities created before the deployment of ABC components. Business Entities define the contracts, based on Business Interfaces, between ABC components and client applications. Each Business Entity is able to manage any set of SQL statements that conform to its associated Business Interface. Moreover, SQL statements may be deployed to each running instance of ABC component in an unbalanced way. This means that the same Business Entity may have different sets of SQL statements in two different running instances of the same ABC component. The SQL statements update process is accomplished in run-time, in any moment and as often as necessary.

All Business Interfaces were designed to improve, from programmers' point of view, the usability of ABC components. The signatures of their methods are type-safe and syntactically based on the schema of their SQL statements.

The development of ABC components is completely decoupled from the development of client applications. Moreover, the process of definition and deployment of SQL statements on each running instance of ABC components may be completely controlled by authorized entities. These two issues allow the separation of concerns through the definition of two main actors: administrators and common programmers.

Despite its complexity, source code of ABC components may be automatically generated by a tool or through a Model Driven Engineering process relieving programmers from the manual development and maintenance processes.

As an outcome of this work, it is expected that this work may open new approaches to the development of business components for database applications. Future work may be divided in two stages: short term and long term.

Short term: 1) Assess and compare ABC performance with a solution based on a standard JDBC. 2) Evolve Business Interfaces in order to support a client-server architecture

Long term: 1) Create a Wide Business Entity able to manage any SQL statement. Its Wide Business Interface supports all known Business Interfaces and, in run-time, by analyzing the metadata returned by queries and by reflection uses the correct interface to communicate with the client application. 2) Assess and compare the Wide Business Entity performance with the standard Business Entities.

## REFERENCES

2011. *Oracle TopLink* [Online]. Oracle. Available: http://www.oracle.com/technetwork/middleware/toplink/overview/index.html [Accessed 2011 Feb].

Bauer, C. & King, G. 2007. *Java Persistence with Hibernate*, Manning.

Bracciali, A., Brogi, A. & Canal, C. 2005. A formal approach to component adaptation. *Journal of Systems and Software,* 74**,** 45-54.

Clements, P. & Northrop, L. 2001. *Software Product Lines: Practices and Patterns*, Addison-Wesley.

Cook, W. & Ibrahim, A. 2006. Integrating programming languages and databases: what is the problem? Available: http://www.odbms.org/experts.aspx#article10 [Accessed 2006].

Costa, C., Pérez, J. & Carsí, J. 2007. Dynamic Adaptation of Aspect-Oriented Components. *In:* Schmidt, H., Crnkovic, I., Heineman, G. & Stafford, J. (eds.) *Component-Based Software Engineering.* Springer Berlin / Heidelberg.

David, M. 1990. Representing database programs as objects. *Advances in Database Programming Languages.* N.Y.: ACM.

Elizondo, P. V. & Lau, K.-K. 2010. A catalogue of component connectors to support development with reuse. *Journal of Systems and Software,* 83**,** 1165-1178.

Erik, M., Brian, B. & Gavin, B. 2006. LINQ: reconciling object, relations and XML in the .NET framework. *In:* ACM SIGMOD International Conference on Management of Data, Chicago,IL,USA. ACM, 706-706.

Flower, M. 2002. *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Griss, M. L. 2000. Implementing product-line features by composing aspects. *Proceedings of the first conference on Software product lines : experience and research directions.* Denver, Colorado, United States: Kluwer Academic Publishers.

Heineman, G. T. & Councill, W. T. 2001. *Component-Based Software Engineering:Putting the Pieces Together*, Addison-Wesley.

Hetrick, W. A., Krueger, C. W. & Moore, J. G. 2006. Incremental return on incremental investment: Engenio's transition to software product line practice. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* Portland, Oregon, USA: ACM.

ISO 2003. ISO/IEC 9075-3:2003. *In:* STANDARDS, I. (ed.). ISO.

Keene, C. 2004. Data Services for Next-Generation SOAs. *WebServices Journal* [Online], 4. Available: http://soa.sys-con.com/read/47283.htm [Accessed 2011 Jan].

Kung-Kiu, L. & Zheng, W. 2007. Software Component Models. *Software Engineering, IEEE Transactions on,* 33**,** 709-724.

Microsoft. 2010. *Microsoft Open Database Connectivity* [Online]. Microsoft. Available: http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx [Accessed 2010 Mar 18].

Microsystems, S. 2008. *JDBC Overview* [Online]. Sun Microsystems. Available: http://java.sun.com/products/jdbc/overview.html [Accessed 2010 Feb 27].

Microsystems, S. 2010a. *Interface ResultSet* [Online]. Sun Microsystems. Available: http://java.sun.com/javase/6/docs/api/java/sql/ResultSet.html [Accessed 2010 Jul].

Microsystems, S. 2010b. *Interface Statement* [Online]. Sun Microsystems. Available: http://java.sun.com/javase/6/docs/api/java/sql/Statement.html [Accessed 2010 Jul 2010].

Pohl, K., Bockle, G. & Linden, F. J. V. D. 2005. *Software Product Line Engineering*, Springer.

Schmoelzer, G., Teiniker, E., Kreiner, C. & Thonhauser, M. 2006. Model-typed Component Interfaces. *In:* Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on, Aug. 29 2006-Sept. 1 2006. 54-63.