International Journal of Engineering Science Invention
ISSN (Online): 2319 – 6734, ISSN (Print): 2319 – 6726
www.ijesi.org Volume 2 Issue 6 ǁ June. 2013 ǁ PP.01-09

An Object Relational Mapping Technique for Java Framework

Ogheneovo, Edward Erhieyovwe,[1] Asagba, Prince Oghenekaro,[1] Ogini, Nicholas Oluwole[2]

[1]*College of Natural and Applied Sciences, Faculty of Physical and Information Technology, Department of Computer Science, University of Port Harcourt, Port Harcourt, PMB 5323, Choba, Rivers State, Nigeria*
[2]*Faculty of Science, Department of Mathematics and Computer Science, Delta State University, Abraka, Delta State, Nigeria*

ABSTRACT: *Database technology is in a period of intensive change and innovation that is both revolutionary and evolutionary. The revolutionary aspect refers to the innovation of an object-oriented programming technology. Object relational mapping (ORM) frameworks address the impedance problem of software implementation level. In this paper, we discussed the problem of building an object relational mapping by presenting a framework based on Java platform that will guide developer in handling incompatibility issues. This paper exploits the powerful features of Java as a managed object-oriented platform. The major goal of the work is to map data between incompatible type systems in the object-oriented programming language by linking the SQL database to object-oriented language concepts, thus creating in effect a "virtual object database". The work discusses how a mini bank application can be designed and implemented as well as mapping of the object-oriented constructs to entities of the relational databases. Implementing the design in code is a straightforward process that results in two Java class libraries (.JAR files) that have to be deployed in the development environment. Java programming language is a modern platform that enables quick and efficient development of information systems. This work focused on developing an object relational mapping based on the Java framework. The framework was successfully used in the development of a new lightweight simulation banking application focusing on eliminating/reducing the huge amount of code required in relational databases.*

KEYWORDS - *Object relational mapping, Object-oriented programming, Persistence databases, relational database*

I. INTRODUCTION

Database technology is in a period of intensive change and innovation that is both revolutionary and evolutionary. The revolutionary aspect refers to the innovation of an object-oriented programming technology. The evolutionary aspect refers to the promotion of a new extended version of relational database technology under the name object-relational database management system [5]. Object-relational database management systems (ORDBMSs) involve the extension of relational database systems to add object-oriented features or direct representation of application objects in relational databases [23]. ORDBMS enhances object-oriented technology into the relational database management system (RDBMS). As an evolutionary technology, ORDBM allows users to take advantages of reuse features in object-oriented technology to map objects into relations and to maintain a consistent data structure in the existing RDBMS. The combination of the two data models was proposed before coining the term object-relational [11]. Thus the ORDBM is a paradigm used to bridge the gap between RDBMS and object-oriented technology.

Relational databases [8] and object-oriented programming languages are based upon distinct paradigms, with technical conceptual and cultural incompatibilities. The set of those incompatibilities is commonly referred to as the object-relational impedance mismatch problem [2]. Object-relational mapping (ORM) frameworks address the impedance problem of software implementation level [3], providing the developer with ways to declare how each technical incompatibility should be treated [22]. ORM tools brings to the same level relational resources, such as data querying and object-oriented resources, such as inheritance, polymorphism, enabling to explore the synergy between those constructs [10]. The Java persistence APIs (JPA) emanated as a result of a community process involving major ORM tool providers in order to standardize ORM tools and frameworks for the Java platform, and is perceived to be the most important cross-vendor standard to ORM frameworks. The mappings of JPA can be used not only to tell the framework how to translate objects to database tuples but also to generate the database structure of the system.As enterprise applications become more complex, it becomes more challenging to map their object models to relational database tables. Object relational mapping (ORM) in computer software is a programming technique for converting data between incompatible type systems in databases and object-oriented programming languages. This creates, in effect, "virtual object databases" which can be used from within the programming language.

In other words, ORM work much harder at hiding the underlying tables, SQL, etc., inside the RDBMS. Due to the popularity of relational data storage with its broad installation base and the use of object-oriented concepts for software development, application objects inevitably need to be stored in (object-) relational databases [9], [16]. Enterprise applications are often developed using an object-oriented programming language (Java or C#) and a relational database. Object relational mapping tools are frameworks for storing and retrieving persistent objects; their goal is to support the complex activity of managing the connections between objects and a relational database. ORM tools allow the programmer to manage the persistence of objects by means of standard API's, such as the JDO (Java Data Objects). Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. Therefore, object relational mapping tool - Hibernate is used for Java environments. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC [4], [21].

Proper object relational integration requires a strategy for mapping the object model to the relational model in order for Java objects to become persistent to the RDBMS. The heart of the problem is translating those objects to forms that can be stored in the database for easy retrieval, while preserving the properties of the objects and their relationships. These objects are then said to be persistent and cannot be directly saved to and from relational databases. While objects have identity, state, and behavior in addition to data, an RDBMS stores data only. This paper looked at the problem(s) of building an object relational mapping framework based on the Microsoft Java platform. The goals of our study are as follows:
(1) To save time and effort by simply focusing on writing Java classes, and not worrying about maintenance of matching SQL script, schema evolution XML based schemas or generating code.
(2) To provide a model that reduces large code thus allowing the developers to focus on the business logic of the application rather than repetitive CRUD (Create, Read, Update, Delete) logic.
(3) To provide rich query capability thus allowing application developers to focus on the object model and not be concerned with the database structure or SQL semantics, and
(4) To support concurrency by allowing multiple users to update the same data simultaneously.

1.1 Objects-Relational Database Management Systems

Object relational database management system (ORDBMS) is an extension of relational database management systems, which added object oriented features or direct representation of application objects in relational databases. ORDBMS may be the most appropriate choice for a DBMS that processes complex data and compiles queries according to Stonebbraker's four quadrant view of the database world [19]. The success of relational DBMSs in the past decades is very evident. However, the basic relational model and earlier version of SQL are inadequate to support object presentation [20]. Traditional SQL DBMSs are not capable of handling complex data found in application areas such as hardware and software designs, science and medicine, document processing, mechanical and electrical engineering, etc. To cope with the above challenges, the object-relational DBMS emerged as a way of enhancing the capabilities of relational DBMS with some of the features that appeared in object oriented DBMSs. ORDBMSs combines the traditional benefits of relational databases with the ability to deal with complex data [23]. ORDBMS is a hybrid of RDBMS and OODBMS that can provide a natural and productive way to maintain a consistent structure in the database; and also inherit the robust transaction and performance management features of its relational database and the flexibility of its object-oriented database. By utilizing ORDBMS, we are able to solve the problems that cannot be solved well in relation database**.**

1.2 Object Relational Mapping

Object relational mapping is a technology that is employed to bridge the impedance mismatch between object-oriented programs and relational database. It tries to eliminate the duplication of data and maintenance cost and susceptibility to error(s) associated with it [13]. It has therefore become a well established programming practice for modern application due to the fact that as enterprise applications become more complex, it becomes more challenging to map their object models to relational database tables. There are a lot of problems to solve when implementing ORM framework. They are connected with effectiveness, data consistency and ease of use of framework by application programmers [25]. Today's trend in programming languages is to utilize objects, thereby making OODBMS ideal for Object-Oriented (OO) programmers because they can develop the product, store them as objects, and can replicate or modify existing objects to make new objects within the OODBMS [17], [1]. Therefore, storing objects in OODBMS can help to significantly reduce the cost of managing and retrieving data especially in large databases.

Thus to be able to access a relational database from an object-oriented application, there are in principle three different possibilities [16]. First, is the direct inclusion of SQL statements into the application's code. This approach allows for the rapid development of prototypes and small systems. Secondly, is the use of tightly-coupled application between application and relational schemas. This approach is not flexible since modification of the schema often requires changes to the source code. The process of encapsulating SQL statements too into own data classes. This approach is slightly better from the architectural point of view although changes to the relational schema still lead to modifications of the application source code. Finally, the third approach is the use of a dedicated middleware, which decouples relational databases from object-oriented applications. In this approach, the object-relational middleware is configured with application specific definitions on how object structures have to be mapped to a relational schema.

1.3 Java Persistence API and Database

Java persistence is the process of storing Java objects to relational databases using the Java persistence API (JPA). There are many ways to persist data in Java. These include: Java Database Connectivity (JDBC), Serialization, file I/O, JCA, Object databases, or XML (eXtensible Markup Language) databases. However, most data are persisted in databases or XML databases. Most activities carried out on websites that concerns data storage involves accessing a relational database. Thus relational databases are the standard persistence store for most corporations. Different types of data can be stored in databases from Java. These include: strings, numbers, dates and byte arrays, images, XML and Java objects. Many Java applications use Java objects to model their application's data, since Java is an object-oriented language, storing Java objects is a natural and common approach of persisting data from Java.

A basic function for object-oriented software is producing, changing and viewing persistent objects. Persistent objects are objects that exist beyond the lifetime of the application. This can be achieved by storing their data in some kind of a data store. The most common persistence mechanism presently is the relational database management system [12]. Relational databases are an efficient and proven technology, they have widespread support in development languages and third party tools, and people are familiar with them. Persistence is the storage of data from working memory so that it can be restored when the application is run again [4]. The Java persistence architecture API (JPA) is a Java specification for accessing, persisting and managing data between Java objects/classes and the relational databases. It is defined as part of the EJB 3.0 specification as a replacement of the EJB 2 CMP Entity Beans specification. It is now considered the standard industry approach for object-relational mapping (ORM) technique [10]. JPA is a specification and not a product; as a result, it cannot perform persistence by itself. It is a set of interfaces and thus requires an implementation. JPA allows POJO (Plain Old Java Objects) that can persist without requiring the classes to implement any interface or method as required in EJB 2 CP specifications. JPA allows an object's ORMs to be defined through standard annotations or XMLs by defining how the Java class maps to a relational database tables. It also defines a runtime EntityManager API for processing queries and transaction on the objects against the database.
Persistence units are defined by the persistence XML configuration file. Fig. 1 is an example of persistence.xml file [15].

```
<persistence>
   <persistence-unit name “OrderMgmt”>
     <provider>com.acme.PersistenceProvider,/provider>
     <jta-data-source> jdbc/MyOrderDBc/jta-data-source>
     <mapping-file>order-mapping.xml</mapping-file>
     <jar-file>myparts.jar</jar-file>
     <properties>
          <property>
               name = “com.acme.persistence.logSQL”
               value = “ALL”/>
     </properties>
    </persistence-unit>
</persistence>
```

Fig.1: Example of persistence.xml file.

1.4 The Impedance Mismatch

The impedance mismatch problem is a well-known problem in persistence objects in relational databases between both the object model and the relational model and between the object programming language and the relational query language [6], [24].

Object-oriented paradigm and relational databases have different concepts that can in practice make it difficult to integrate them together. While object-oriented development is based on the concept of data and their behaviours, relational databases are based on purely data [18], [3]. The object-oriented paradigm views data mainly as behaviours, that is, objects are important not just because of the data they contain, but because of their ability to perform tasks on the data and exchange hand, the relational paradigm places emphasis on the data itself and its structural (not behaviour) relationship with other data [7]. Thus there is always a problem of mismatch when trying to map or integrate these concepts together. This mismatch problem is what is referred to as impedance mismatch [24]. Thus there is need to solve this problem in order to improve the interaction between object-relational paradigm and relational database technology.

The mismatch problem is a very common source of performance problems in Java applications due to the way objects are accessed in Java and in relational databases. Experience has shown that up to 30 per cent of code written in Java applications is meant to handle SQL/JDBC and the manual bridging of the object/relational paradigm mismatch. Despite all these efforts, there is no significant achievement to resolve this ugly trend as many projects have failed due to the complexity and inflexibility of their database abstraction layers. Thus to resolve this problem, there is usually a process of modeling both the object-oriented and relational paradigms such that they are often subjected to some bending and twisting of the object-oriented paradigm until it matches the underlying relational database technology [4]. Therefore, this research was carried out to bridge the gap between the object-relational paradigm and relational database technology.

II. MATERIALS AND METHODLOGY

On the development station, we used a system with a Pentium Dual-core 2.10Ghz Processor, 2GB Ram, 64-bit System Architecture, and Windows 7 Operating System. We also deployed Java Platform version 1.6.0_20 using Java SE runtime build 1.6.0_20-b02**.** Java was used as the implementation language. It provides by default a wealth of frameworks, API’s, standards implementations, and libraries that foster reusability and reduces development time.

2.1 Methodology

The object-oriented paradigm is based on software engineering principles such as coupling, cohesion, and encapsulation, whereas the relational paradigm is based on mathematical principles, especially those of relations and set theory. The two different theoretical foundations have their own strengths and weaknesses. Furthermore, the object-oriented paradigm is focused on building applications out of objects that have both data and behaviour, whereas the relational paradigm focused on storing data. The “impedance-mismatch” comes into play when looking at the preferred approach to data access. With the object-oriented paradigm, objects are traversed via their relationships, whereas, with the relational paradigm data are duplicated to join the rows in tables. This foundational difference results in a less-than-ideal combination of the two paradigms.Typically, databases access frameworks or application programming interfaces (APIs) are built as a platform or bridge between the database and the program/system/user trying to gain access. Databases can interact with SQL statements or some variant of it. Thus the framework acts as a channel through which the SQL statement can be communicated to the database server. This means that developers on the other side of the framework will spend more time building queries from domain objects to the flat form of the relational data model supported by databases. Many databases access frameworks have been built using this or some variation or extension of this architecture. Two of the most popular are:

- Active Data Objects (ADO): this was Microsoft’s flagship data access framework before the era of .NET. Even the new version developed for the .NET platform is still based on the same architecture, though it supports more features.
- Java Database Connectivity (JDBC): this is the standard access API used on the Java platform. It adheres strictly to this architecture, varying and/or enhancing it in only few areas. JDBC is a standard Java Interface for connecting from development architectures to connect to SQL Server. Java Server Pages (JSPs), Java Servlets, and Enterprise Java Beans (EJB) all use JDBC for database connectivity, as well as many other Java application architecture [14].

2.2 Proposed System

The objective of any object relational mapping (ORM) framework/API is to relieve the programmer from the drudgeries of manual SQL statement writing and manipulation, and allow them to focus on manipulating data structure from the problem domain. However, the proposed model aims for optimum scalability. This scalability refers to the ability to write code that executes business logic and access persisted data once, irrespective of the backend in use. Thus a shift from one database backend to the other will have absolutely no effect on the system.

The overall architecture for the proposed system utilizes structures that represent tables, databases, and connections to these databases. It also houses lower-level structures modeling table rows, columns, data types, and converters. The framework should be configurable using eXtensible Markup Language (XML) files by following a specified format. This configuration will also foster scalability objectives by providing a medium for specifying connection settings. This is useful because specific issues are separated from the logic of the code, and can exist in an easily editable, dynamically uploaded medium. Thus with the right structural format of the database objects, any database can be specified and connected to it. Another notable consideration is the ability of the framework to dynamically discover all structural data and meta-information about them from the database. This will boost usability of the framework, and reuse configuration time because duplication of the structures will not be specified in the configuration files. The system should be able to build all internal structures that define its functionality from the context of the new paradigm shift. Fig. 2 shows the architecture of the framework components.

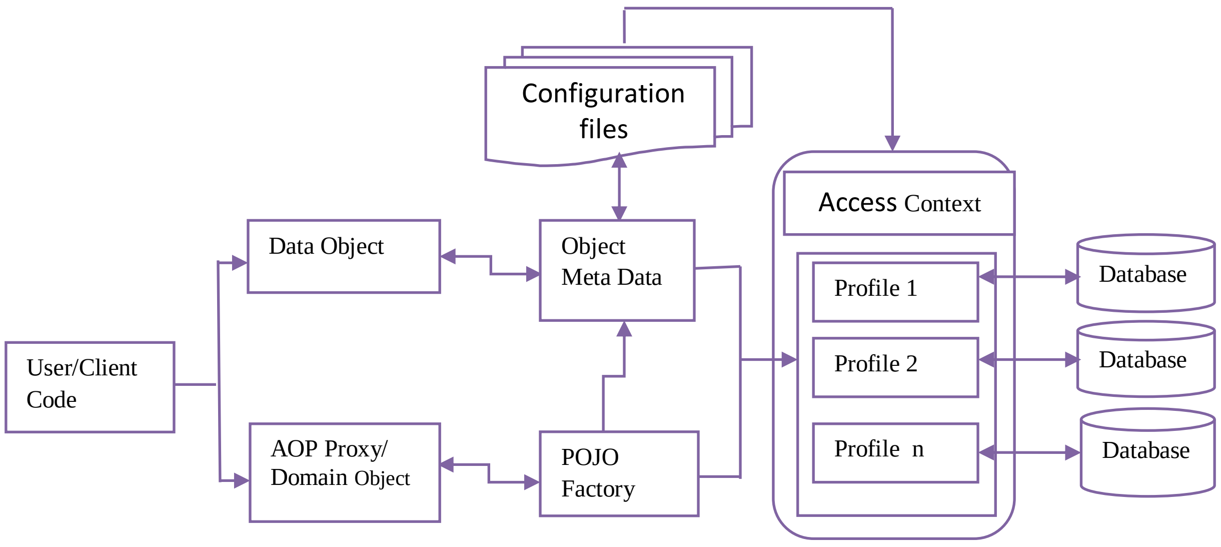

Fig. 2: Architecture of the framework components.

Aspect-oriented programming (AOP) is implemented to wrap services around the client implementation of the business/domain object interface. This way, as well as updating the properties of domain interface implementation with data, the generic data-object gets updated also; so a call to a designated persist method will cause the generic data-object to get persisted totally and transparently to the client code. The focal point of the framework is the data-object. The data object is a generic representation of an object. An object from the design point of view is an entity that encapsulates properties and functionality**.** Typical ORM focuses on persistence of data as functionality cannot easily be translated to the relational data model; this framework is no exception. The data-object encapsulates properties and is mapped to rows of tables; in essence, each data object roughly represents a row of a relational table. Upon initialization, the framework reads connection and configuration information from the configuration files and initializes internal structures. Among these structures are: Connection-Profiles, Object-Meta-Data, Converters, and the Plain Old java Object-Config. (POJO-Config.) POJO-Config is used internally to set up the AOP proxy factory for creating domain objects that are extended to be persisted by the ORM framework. The next section discusses in details the functionalities of the identified components.

2.3 Access Context

This is the entry point to the framework. Every client code should have at least one of these to access the functionality of the framework. Initializing an Access Context requires passing as a parameter, a reference to the XML configuration file. The contents of this file are used to configure the internal structures of the Access Context. The Access Context possesses one or more connection profiles, called DBProfiles. These specify connection properties used in connecting to the database. Other configuration may be specified within the context of the profiles.

The DBProfile on the other hand is less compared to the Access Context. It houses structures used in creating registered POJOs (Plain Old Java Objects), accessing information about the database the profile is connected to, retrieving any of the Object-Meta-Data objects that are mapped to table objects, as well as creating objects that represent SQL statements used to query the database in a conventional manner. Each profile is given a name that must be unique within the Access Context, as well as all other structures within the context; thus valid names are given or can be derived for each structure to uniquely identify it from any context.

2.4 Object Meta Data

Each object-meta-data represents a table in the database. Thus upon initialization, information about the database tables have to be supplied framework. This, instead of being supplied through the configuration, is automatically extracted from the database through Java's JDBC API. This single feature ensures that the framework can connect with any database without prior knowledge of its schema - in essence, configuration time is greatly reduced. Customization can be applied, however, to the table column data in the form of converters. Imagine a field holding information for "sex". This field will typically hold values "male" and "female". If from the domain standpoint these values have been implemented as enumeration or objects, it will be suitable to work with these automatically than normally (and occasionally erroneously) convert each time the values are to be applied. Converters can be configured on any field. They are applied each time data is retrieved from and sent to the database. The object-meta-data is also responsible for retrieving records from the database and converting them to generic data objects or POJO objects. The conversion to generic objects is possible because the entire table schema information is contained within the object-meta-data. POJOs on the other hand are actually proxy objects that implement all business interfaces within the configuration for a certain object, and then a couple other interfaces necessary for identifying objects and interfaces that are to be mapped to which object Meta data. Thus each Meta data object can automatically create the appropriate POJO when requested.

2.5 Data Objects and Fields

Data objects are mapped directly to the business objects, which in turn are what client codes work with. A data objective is an encapsulation of a field. These fields are identified by names - thus internally, are stored using a Hash-Map that maps the names to the fields for quick access. The data object provides methods that implement the basic CRUD (Create, Retrieve, Update, Delete) methods, except for the Create method. For the retrieve method, a better name would be 'refresh' because what it does is retrieving the result of that row from the table, repopulating the fields with the new values. As stated earlier, the proposed model uses the relational mapping technique. Field on the other hand map to columns of the table but on the data object represents properties. Each field has a unique name in the context of the data object, and as such valid names can be derived for them. Each field has a type description that contains information describing the data type of the field both within the Java application and the database in order to attain appropriate conversion while transferring data to fro**.** Each field also contains information stating whether or not it is a primary key, a foreign key, a unique index, a nullable, etc. Most importantly, each field contains a cache used for its values.

2.6 AOP Domain Object Factory

The AOP object factory is used internally to create proxies for the domain interfaces that will wrap around some actual domain object. The factory is configured from the XML configuration files and has extensions mapped to each object Meta data. Thus using the proxies is as easy as requesting one created, or retrieving results as proxies instead of as generic objects. The AOP proxy factory also allows developers attach their own advice (method service) to methods, configurable via the same configuration file. The default advice given delegates property updates, persistence requests, and other CRUD method calls to the target domain object.

2.7 SQL Structures

Within the object Mata data and the data objects, CRUD requests automatically generate SQL statements that are marshaled off to the database server. These structures are simple classes that represent the distinct parts of an SQL CRUD query. These structures are made public to the developers so they can exploit the dynamism they provide in generating queries automatically. Fig. 3 shows an UML diagram depicting classes, interface and their structural relationships.

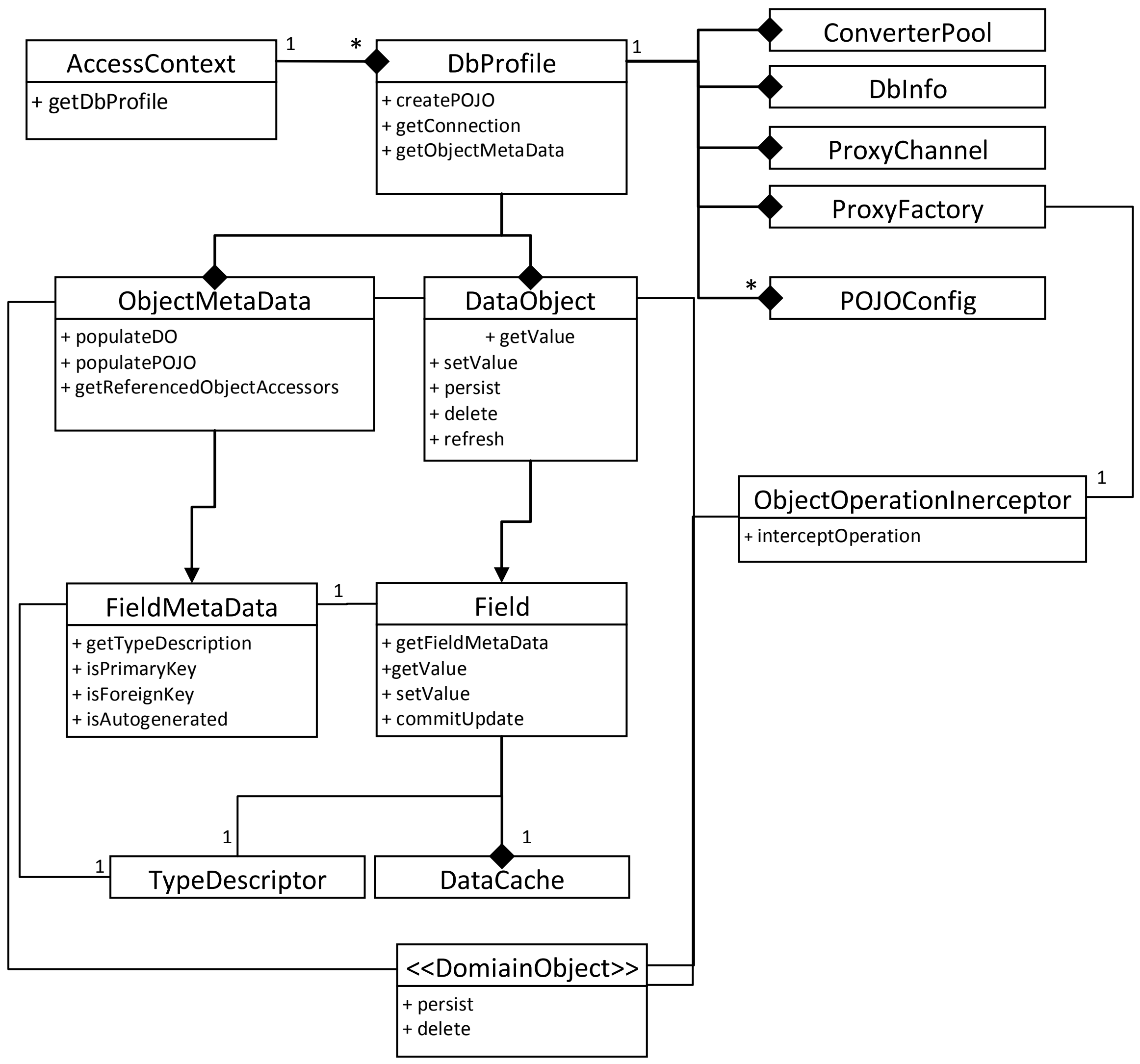

Fig. 3: UML diagram depicting classes, interface and their structural relationships

2.8 A Bank Simulation Process Overview

To demonstrate the operations of the framework presented in this paper, we developed a sample application. This application is a lightweight bank simulation, modeling three of the most common transactions that occur in a bank's day to day activities. These are: account debit, account credit, and funds transfer. To implement this simulation, we identified and extracted out the following entities as the domain objects: user, account information, personal information, transaction, and customer. Fig. 4 shows an illustration of the process flow of the simulation application where bank staff logs into the system and is presented with the options of viewing transactions, posting transactions or managing customers. Posting transactions simply involves entering the appropriate data for the transaction, by clicking the "post" button. Managing users simply involves either creating new ones or modifying personal information of existing ones.

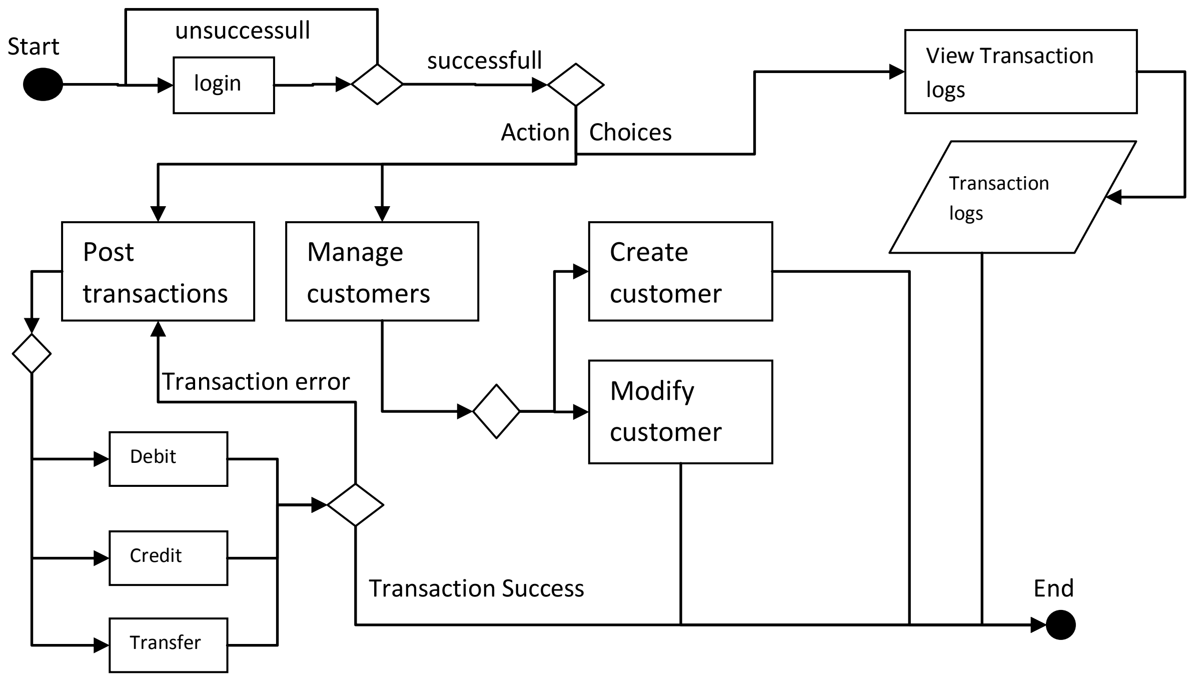

Fig. 4: An illustration of the process flow of the simulation application

III. RESULTS

The framework discussed in this paper was implemented using the banking activities on transaction processing of customers. It demonstrated the simulation of bank's activities on daily basis on whether a transaction is successfully completed or aborted, as presented in Fig. 4. Our ORM framework features two modes of operations: generic object mode and domain object mode. Both modes can be operated simultaneously depending on development requirements or design architecture, or preference. The framework depends on both configuration (through XML) and the JDBC framework to communicate with, as well as provide critical Meta data information about the database being connected to. Implementing the design in code is a straightforward process that results in (after compilation) two Java class libraries (.JAR files) that have to be deployed in the development environment. Configuration information is then supplied as an XML file with a specific name and deployed in the execution directory of the application. The class designs are translated into java classes, appropriate logic is applied to provide functionality where needed. As seen in the design, the Data Objects are implemented much like hash maps, but only much more complex. They encapsulate name-field maps, and provide shortcuts for accessing values of the fields. Each field in turn is bound to a Field Meta Data which emanates from the Object Meta Data unto which the Data Object is bound. The fields have data types that correspond to java platform's primitive types, and a general o*bject* type for more complex data. Converters may be provided by the client code through the configuration files to convert specialized data types to those used on the database. Data objects as well as their surrogate domain objects will have methods to perform CRUD operations. On the domain objects, these are actually trivial because the logic provided by the domain object will play no part in the actual operation dealing with the database - because this is supported by the domain object.

The Object Meta Data has the functionality of creating data objects and converting them to domain objects. It is also the point where requests (filtered or not) are made for data objects (records) from a particular table. Filtered requests are made by either supplying a string expressing an SQL query condition, or building the condition using the structures provided. Each object Meta data houses the schema information of specific table on the database. Domain objects on the other hand are created by inspecting an in-memory version of the configuration for specified domain interface. The class objects of this is obtained and queried for all extended interfaces. These are bundled into an array and used to generate a proxy serviced by the default framework's dynamic method service. Custom services may be provided by the developers as speculated in the design. These are applied to the proxy as well before it is returned. The results of this process are cached so that subsequent calls need not repeat the process - the interfaces and services are applied automatically. The DBProfile, like the data object, is implemented as a hash map. Its primary objective is providing information about the connection environment, as well as providing Object Meta Data objects on request. The DBProfile maintains the connection till Access Context is shut down.

IV. CONCLUSION

Java programming language is a modern platform that enables quick and efficient development of information systems. This work focused on developing a technique, the relational mapping technique that is capable of constructing an object relational mapping based on the Java framework. The framework was successfully used in the development of a new lightweight simulation banking application focusing on eliminating/reducing the huge amount of code required in relational databases. This pattern makes it possible for one to model entities based on real business concepts rather than based on the database structure. Therefore, a banking simulation application is a better way to handle bank transactions due to the fact that it makes it easier for object relationships to be navigated transparently.

REFERENCES

[1] S. Agarwal, C. Keene, and A. M. Keller, (1995), Architecting object applications for high performance with relational databases. Technical Report, Persistence Software. Available at http://www.persistence.com, Retrieved August 20, 2012.

[2] S. Amber (2003), Agile database techniques: effective strategies for the agile software developer, USA: John Wiley and Sons, http://www.agiledata.org/, Retrieved April 20, 2011.

[3] S. W. Amber (2006), Mapping objects to relational databases: O/R mapping in detail, http:///www.ambysoft.com/mappingObjectsTut.html.

[4] C. Bauer and G. King (2005), Hibernate in action, manning publishers Co. Boston/Massachusetts, pp. 1-25.

[5] M. R. Blaha, W. J. Premerlani, and J. E. Rumbaugh (1988), Relational database design using an object-oriented methodology, Communications of the ACM, Vol. 31, No. 4, pp. 414 - 427.

[6] R. G. G. Catell (1991), Object data management: object-oriented and extended relational database systems, Addison-Wesley Publishing Company.

[7] R. G. G. Catell, D. Barry, M. Barler, J. Eastman, D. Jordan, C. Rusell, O. Schadow, T. Starienda, and F. Velez (2000), The object data standard: ODMG 3.0, Morgan Kaufmann Publishers, San Franscisco.

[8] Codd, E. F. (1970). A Relational Model of Data for large Shared Data Banks. Communications of the ACM, Vol. 13, No. 6, pp. 377-387.

[9] C. J. Date (2004), An introduction to database systems, 8th Ed., Boston, Addison-Wesley Publisher.

[10] L. DeMichiel and M. Keith (2006), JSR 220: Enterprise JavaBeans^{TM}, version 3.0. Java Persistence API, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[11] M. Fotache and C. Strimbei (2009), Object-relational databases: An area with some theoretical promises and few practical achievements. Communications of the IBIMA, Vol. 9, No. **7**, ISSN: 1943-7765.

[12] M. Fussel (1997), Fundations of object-relational mapping, ChiMu Corporation, www.chimu.com/publications/objectRelational/, retrieved August 20, 2012.

[13] A. Keller and C. Keene (1993), Persistence software: bridging object-Oriented programming and relational databases, In Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, pp. 540-541.

[14] S. Kost (2004), An introduction to SQL injection for Oracle developers, Wikipedia.org, Retrieved September 21, 2011.

[15] Oracle Corporation, Hibernate in action, http://www.oracle.com/hibernate.html, retrieved August 20, 2012.

[16] S. Philippi (2005), Model driven generation and testing of object-relational mappings, Journal of Systems and Software, 77, pp. 193-207.

[17] S. Ramanathan and J. Hodges (1997), Extraction of object-oriented structures from existing relational databases. ACM SIGMOD, Vol. 26, No.1.

[18] K. Smith and S. B. Zdonik (1987), Intermedia: A case study of the differences between relational and object-oriented database systems. In Proceedings of the OOPSLA'87 Conference on Object-Oriented Programming Systems, Languages and Applications.

[19] M. Stonebraker, J. Anton, and E. Hanson (1987), Extending a database system with procedures, ACM Transactions on Database Systems 12(3), pp. 350-376.

[20] M. Stonebraker, L. A. Rowe, B. Lindsey, J. Gray, M. Carey, M. Brodie, P. Bernsteid, D. and D. Beech (1990), Third-Generation database systems manifesto, ACM SIGMOD Record, 19(3), 1990, pp. 31-44.

[21] Sun Microsystems (2006), The java persistence API - A simpler programming model for entity persistence, **http://java.sun.com/developer/technicalArticles/J2EE/jpa.**

[22] A. Torres, R. Galante, and M. S. Pimenta (2009), Towards a UML profile for model-driven object-relational mapping, IEEE Computer Society, pp. 94 -103.

[23] M. Wang (2010), Solving relational database problems with ORDBMS in an advanced database course. In Proceedings of Information Systems Educators Conference (ISECON'10) Nashville Tennessee, USA, Vol. 9, ISSN: 1943-7765, pp. 47-55.

[24] P. V. Zyl, D. G. Kuorie, and A. Boake (2006), Comparing the performance of object databases and ORM tools, In Proceedings of SAICSIT'06, pp. 111-113.

[25] A. E. Doroshenko and V. Romenko (2004), Object relational mapping techniques for .NET Framework, ISTA, 2004, 81-92, http://www.informatik.unitrier.de/.../d/Doroshenko:Anatoly_E=.html.