# DEVELOPING AND DEPLOYING DYNAMIC APPLICATIONS
## *An Architectural Prototype*

Georgios Voulalas, Georgios Evangelidis
*Department of Applied Informatics, University of Macedonia, 156 Egnatia St., Thessaloniki, Greece*
*voulalas@uom.gr, gevan@uom.gr*

Keywords: Model-driven Development, Dynamic Applications, Reflection, Runtime Compilation, Java Platform

Abstract: In our previous research we have presented a framework for the development and deployment of web-based applications. This paper elaborates on the core components (functional and data) that implement the generic, reusable functionality. Code segments of the components are presented, along with a short sample application. In addition, we introduce some changes that are mainly driven by usability and performance improvements, and are in adherence with the principal rules of the framework's operation. Those changes enable us to extend the applicability of the framework to other families of applications, apart from web-based business applications.

# 1 INTRODUCTION

In (Voulalas & Evangelidis, 2007 and Voulalas & Evangelidis, 2008) we introduced an extension of the Model Driven Architecture (Kleppe, Warmer & Bast, 2003) that additionally to coping with the problems of productivity, documentation, maintenance, portability and interoperability, aims to manage efficiently the **continuously evolving requirements** (Miller & Mukerji, 2001) of web-based business applications and to ensure **consistency between the produced code and the preceding design models.**

The framework is structured on the basis of a **universal database schema** (meta-model). Development is supported by modelling tools that elicit functional specifications from users and transform them into formal definitions, and by data structures that are utilized for the storage of the definitions. Deployment is supported by **generic components that are dynamically configured at run-time** according to the functional specifications provided during development, and by application-independent data structures (part of the meta-model) that hold all application-specific data. No code (SQL, Java, C++, JSP, etc.) is generated. We can **deal with business logic changes at deployment time** without recompiling and redeploying the application, since changes are translated in modifications to the underlying data instances. **Consistency between the final application and the modelled specifications is ensured.**

The framework supports the operation of **different applications and application versions within a single installation**. Thus, we can anytime refer to a previous version of an application and examine old data in its real context by retrieving the corresponding data instances, **without the need of maintaining multiple installations**.

During our research, **performance and usability issues** motivated us to differentiate the way some of the goals are achieved. In particular, instead of utilizing exclusively generic components for the provision of the functional behaviour of an application, we decided to use a mix of generic modules and application-specific modules. Application-specific modules will be retrieved from the database and compiled at run-time (Biesack, 2007) in order for the application to be **dynamically created**. This change improves the performance of the generated application since the extensive utilization of reflection requires increased computational resources (Sun Microsystems, 2008). Additionally, the developer has more means to implement sophisticated mechanisms that could not be incorporated into a pre-built generic component. Last but not least, this shift enables us to easily extend the applicability of the framework to other families of applications, apart from web-based business applications.

This paper aims to present an application scenario (Section 2) that will indicate the way the developer interacts with the development environment, and verify the feasibility of the proposed solution though an architectural prototype (Sections 3 and 4). The last section concludes the paper and identifies our future steps.

## 2 APPLICATION SCENARIO

Suppose that we have to develop a real estate portal, through which agents will be able to post property ads and potential buyers / renters to search for properties. Three types of properties ads are identified: residencies, business properties, and development land. A property can be available for sale or rent, or for both sale and rent.

In order to verify the feasibility of our proposal, we have developed two versions of the portal. In both versions, Java Platform™ Standard Edition 6 is the underlying platform. The first version was developed as a typical application with its own database schema (residing in MySQL) and functional components that were developed from scratch through an Integrated Development Environment (IntelliJ IDEA). For the development of the second version, we followed the steps that are described below. No database schema was created, since that version utilizes the generic database schema that is presented in Section 3 (and which also resides in our MySQL Server). Application-specific code was written through IntelliJ IDEA.

- Identify the business entities that participate in the domain model, e.g. Property, Residency, Business Property, Development Land and Realtor. Create a new **Class** for each one. Look-up structures should be also modelled as **Classes** (e.g. Transaction Type: sale or rent).
- Identify the properties of each business entity, e.g. size of Property, construction year of Residency. Create a new **Attribute** for each property and associate it with the proper **Class**. Properties that refer to other **Classes** should be also modelled as **Attributes**.
- Identify many-to-many relationships between **Classes**, e.g. Property – Transaction Type and create a new **Association** for each one.
- Model properties that are carried by the **Associations**, e.g. sale cost. Create a new Attribute for each property and associate it with the proper **Association**.
- Identify the operations that are implemented by each class, e.g. insert operation that creates a new property. For each method create a **Method** and link it to the proper **Class**.
- Identify the parameters that should be passed in order for an operation to be invoked, e.g. size, construction year. For each parameter create an **Argument** and link to the proper **Method**.

In the final prototype those steps could be supported by a forms-based wizard.

The system stores the information provided by the user in the predefined database structures (see Figure 1, Region A). Note that no new structures are created and no existing structures are modified. Interaction with the database is limited to data insertion. Then, the system generates code skeletons for each **Class**. In Java SE the code skeletons include attributes, setters and getters for each attribute, default constructor, signatures for the methods and static declarations that link the Class to the database records created in the previous step.

Now, it is the user's turn to implement the methods. The coding process is strictly restricted to implementing the methods that have been already defined, or defining new private methods that are visible only within the scope of the specific class. The code that is written is stored in the database (see Figure 1, Region A, column body of the Methods table). In the final prototype, the user instead of writing code, he will provide functional specifications in an upper level (e.g. with the use of a designer) that will be transformed to code. Both functional specifications and generated code will be stored in the database, along with the definition of **Classes**, **Attributes**, **Methods**, etc. Consistency between the functional specifications and the generated code will be ensured by preventing the user from writing directly to the database.

Until now, we have covered the implementation of the Application tier. The Database tier is generic for all applications. The user does not write SQL code. Instead, he uses generic methods that materialize and dematerialize the objects. For the User Interface, we propose that the developer should be able to freely and creatively implement it.

## 3 DATABASE MODEL

In Figure 1, we present the database model upon which the development and deployment platform is built. The model is abstractly divided into two parts:

- The first part (Region A) holds the functional specifications of the modelled application. It includes the following entities: Classes, Attributes,

Methods, Arguments, Associations and Imported Classes. The Applications table enables the operation of multiples applications within a single installation.

- The second part (Region B) holds the application data that are generated during application execution and includes the following entities: Objects, AssocitionInstances and AttributeValues.

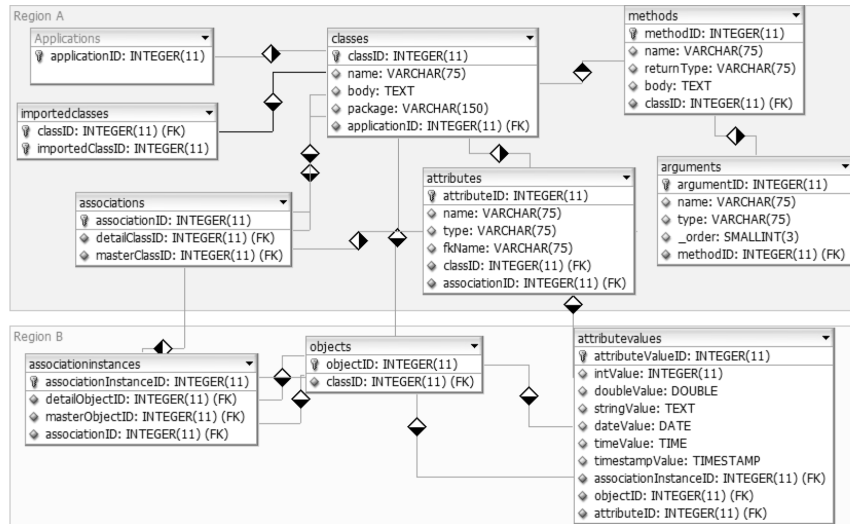


Figure 1: The Database Model.

# 4 ARCHITECTURAL PROTOTYPE

In order to verify the feasibility of the framework we have implemented a method that posts a new property ad. The method is implemented by the Property class and has the following signature:

```
public int insert(Integer realtorID, …,
Hashtable transactionTypes);
```

In the following sub-sections we present segments of code derived from the two versions of the real estate portal. For space saving and simplicity reasons, we omit error handling and code details that are not important for the reader.

## 4.1 Invoking a method from the User Interface

In the first version of the real estate portal the insert method is invoked as follows:

```
Property p = new Property();
// tt is a hash table with supported
// transaction types (sale, rent)
p.insert(new Integer(2), …, tt);
```

In this second version, method invocation includes:

- **Specifying the method that is to be executed** by providing its name, the name of the class it implements it and the package of the class.
- Creating an **array of objects** that includes the arguments that will be used.
- **Invoking the execute method** of the _Method class (generic class**).**

```
String cls = "Property";
String pckg = "application.entities";
String mthd = = "insert";
Object[] args = {new Integer(2), …, tt};
_Method.execute(cls, pckg, mthd, args);
```

## 4.2 Executing the Method

The signature of the execute method that controls the execution of a method is the following:

```
public static void execute (String
className, String packageName, String
methodName, Object[] arguments)
```

Using the first two arguments the class is identified and retrieved from the database. As a second step, the classes that are imported by the class are fetched (Region A of the database model, table ImportedClasses). Then, the method and its arguments (name and type for each one) are retrieved. The next step is to compile all involved classes. Finally, the method is invoked:

```
// load the class that implements the
// method that should be invoked
File classesMap = new File("c:/classes");
URL[] urls =  new URL[]
       {classesMap.toURI().toURL()};
URLClassLoader ucl;
ucl = new URLClassLoader(urls);
Class _class;
_class = ucl.loadClass(pckg + "."+ cls);
// create an array of Class objects that
// identify the method's formal parameter
// types (based on the types retrieved
// from the db)
Class[] paramTypes;
paramTypes = new Class[argms.size()];
for (int i = 0; i < argms.size(); i++) {
 arg = (_Argument)argms.elementAt(i);
 paramTypes[i] = Class.forName
                (arg.getType());
} // end of for statement
// invoke the method
_class.getDeclaredMethod(mthd, paramTypes)
    .invoke (_class.newInstance(), argms);
```

## 4.3 Inside the Method

The code of the method that inserts a new property ad is mainly the same for both versions of the real estate portal. Differences are found in the interaction with the database. In the second version, for each Class two private methods are automatically generated. Those methods cover the materialization and the dematerialization of the objects using the database structures that belong to Region B of the Database Model.

# 5 CONCLUSIONS & FURTHER RESEARCH

Our approach enables the dynamic configuration of every aspect of the application in the business-logic tier, in contrast to other efforts (e.g. Pinto, Jimenez, & Fuentes, 2005) that are restricted to common services (e.g. authentication, message filtering). Additionally, having a single database meta-model that supports the operation of all applications and their versions, we are able to dynamically handle any change in the database tier. Similar efforts (e.g. Dmitriev, 2001) enable only simple changes such as field renaming.

Future research efforts will focus on:

- Implementing a policy that deals with active classes (i.e. classes that are used by active threads) that need to be dynamically recompiled in order to reflect changes.
- Implementing a forms-based wizard that will support the first steps of the application scenario presented in Section 2.
- Extending Region A of the Database Model in order to hold functional specifications that can be forward-engineered to code and develop a code generation tool.
- Applying data versioning techniques in the Database Model in order to be able to dynamically (i.e. on runtime) recall past data and previous versions of the applications.

# REFERENCES


Biesack, D., 2007. *Create dynamic applications with javax.tools*, http://www.ibm.com/developerworks/ java/library/j-jcomp/index.html

Dmitriev, M., 2001. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. *In Workshop on Engineering Complex ObjectOriented Systems for Evolution, October 2001.*

Kleppe, A., Warmer, S., Bast, W., 2003. *MDA Explained. The Model Driven Architecture: Practice and Promise*, ch. 1. Addison-Wesley, Reading.

Miller, J., Mukerji, J., 2001. *Model Driven Architecture – A Technical Perspective*, http://www.omg.org/ cgi-bin/doc?ormsc/2001-07-01

Pinto, M., Jimenez, D., Fuentes, L., 2005. *A Dynamic Component and Aspect Oriented Platform,* The Computer Journal.

Pinto, M., Jimenez, D., Fuentes, L., 2005. Developing Dynamic and Adaptable Applications with CAM / DAOP: A Virtual Office Application, In GPCE 2005. Springer-Verlag, LNCS 3676, pp. 438–441, 2005.

Sun Microsystems, 2008. *The Reflection API*, http://java.sun.com/docs/books/tutorial/reflect/index.html

Voulalas, G., Evangelidis, G., 2007. A framework for the development and deployment of evolving applications: The Domain Model, ICSOFT 2007

Voulalas, G., Evangelidis, G., 2008. Introducing a Change-Resistant Framework for the Development and Deployment of Evolving Applications, ICSOFT 2006, CCIS 10, pp. 293–306, 2008