APPLICATION VERSIONING, SELECTIVE CLASS RECOMPILATION AND MANAGEMENT OF ACTIVE INSTANCES IN A FRAMEWORK FOR DYNAMIC APPLICATIONS

Georgios Voulalas and Georgios Evangelidis
Department of Applied Informatics, University of Macedonia, 156 Egnatia St., GR-54006 Thessaloniki, Greece

Keywords: Model-driven development, Dynamic applications, Runtime compilation, Java platform, Data versioning, Selective recompilation, Meta-model.

Abstract: In our previous research we have presented the core functional and data components of a framework for the development and deployment of web-based applications. The framework enables the operation of multiple applications within a single installation and supports runtime evolution by dynamically recompiling classes based on the source code that is retrieved from the database. It is structured upon a universal database schema (meta-model). The contributions of this paper include a versioning mechanism that enables access to old data in their real context (i.e., within the version of the application that created this data), a proposal for selective recompilation of new classes that allows applications to evolve safely at the minimum processing cost, and a policy for handling active classes (i.e., classes that have running instances) that need to be dynamically recompiled in order to reflect changes.

1 INTRODUCTION

There is a growing class of applications that are large and complex, exploit object persistence, and need to run uninterrupted for long periods of time. Almost all of these applications require a combination of complex data models, advanced logic and persistence (Atkinson and Jordan, 2000). At present, these requirements are supported by a combination of database systems, programming languages and software production tools. In parallel, the steady increase in affordable computational power removes inhibitions on application complexity and results in demand for more complex software systems. On the other hand, this availability of powerful computational resources encourages the elaboration of sophisticated development platforms that enable developers to efficiently cope with the demand for more and better software.

Towards this goal, we presented (Voulalas and Evangelidis, 2008a), (Voulalas and Evangelidis, 2007) and (Voulalas and Evangelidis, 2008b) a development and deployment framework that targets to web-based business applications. The main principles that manage the operation of the framework are the following:

- The framework is structured on the basis of a universal database schema (meta-model).
- Deployment is supported by generic components and application-specific components that are retrieved from the database and compiled at runtime (Biesack, 2007) in order for the application to be dynamically created.
- The Database tier is common for all applications. The user does not write SQL code. Instead, he uses generic methods that materialize and dematerialize objects.
- For the development of the Application tier (i.e., classes) the user should be provided with a custom editor implementing a moderated environment, or (ideally) a code generation tool for transforming functional specifications provided by the user to code. In both cases the code is stored in the database along with the dependencies between the classes.
- For the Presentation tier, a more flexible and less moderated approach is proposed in order for the user to be able to creatively develop the user interface of the application.
- The framework supports the operation of multiple applications within a single installation. There

Voulalas G. and Evangelidis G.
APPLICATION VERSIONING, SELECTIVE CLASS RECOMPILATION AND MANAGEMENT OF ACTIVE INSTANCES IN A FRAMEWORK FOR DYNAMIC APPLICATIONS.
DOI: 10.5220/0001836703170322
In *Proceedings of the Fifth International Conference on Web Information Systems and Technologies (WEBIST 2009)*, page
ISBN: 978-989-8111-81-4
Copyright © 2009 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

always exists one deployed application, independently of the actual number of running applications.

- We can efficiently manage continuously evolving applications. We can deal with business logic changes at deployment time without interrupting the operation of the application, since changes are embedded in the application through the dynamic recompilation of modified classes.
- We can anytime refer to a previous version of an application and examine old data in its real context (i.e., within the version of the application that created this data) by retrieving the corresponding data instances from the database, without the need of maintaining additional installations (one for each different application version).

In order to verify the feasibility of our proposal, we have developed the core functional and data mechanisms. The underlying database schema resides in MySQL. For the functional components, Java was an obvious choice for us considering its high and still growing popularity, its increasing adoption for development of large and long-lived applications and of course the fact that it supports run-time compilation of classes in Java Platform^{TM}, Standard Edition 6 (Sun Microsystems, 2008). However, it is clear that a number of aspects of our research can be generalised for other programming languages that support run-time compilation (e.g. .NET). Last but not least, despite the fact that our framework focuses on web-based business applications, we believe that it can be extended to other families of applications.

This paper elaborates on three important mechanisms:

- a simple data versioning technique that allows us to keep all different versions of the modelled applications (Section 2),
- a method for selecting only those classes that have been modified and need to be recompiled, in order for redundant processing costs to be avoided (Section 3), and
- a policy for handling active instances of classes that need to be recompiled in order to reflect changes (Section 4).

In Section 5, we provide a conclusive summary of the paper and we identify our future research plan.

2 MAINTAINING DIFFERENT VERSIONS OF APPLICATIONS

Since source code and data structures of the deployed applications are stored in the database, in order to be able to return to a previous version of an application, we just need to apply a data versioning technique. As a basis for the data versioning technique we used the Envers Project (entity versioning) of the jboss community. Envers project (Warski, 2008) was mainly selected due to its simplicity in comparison with other techniques (Zhu, 2003).

For each versioned entity a “versions” table is created. The versions table has the following fields:

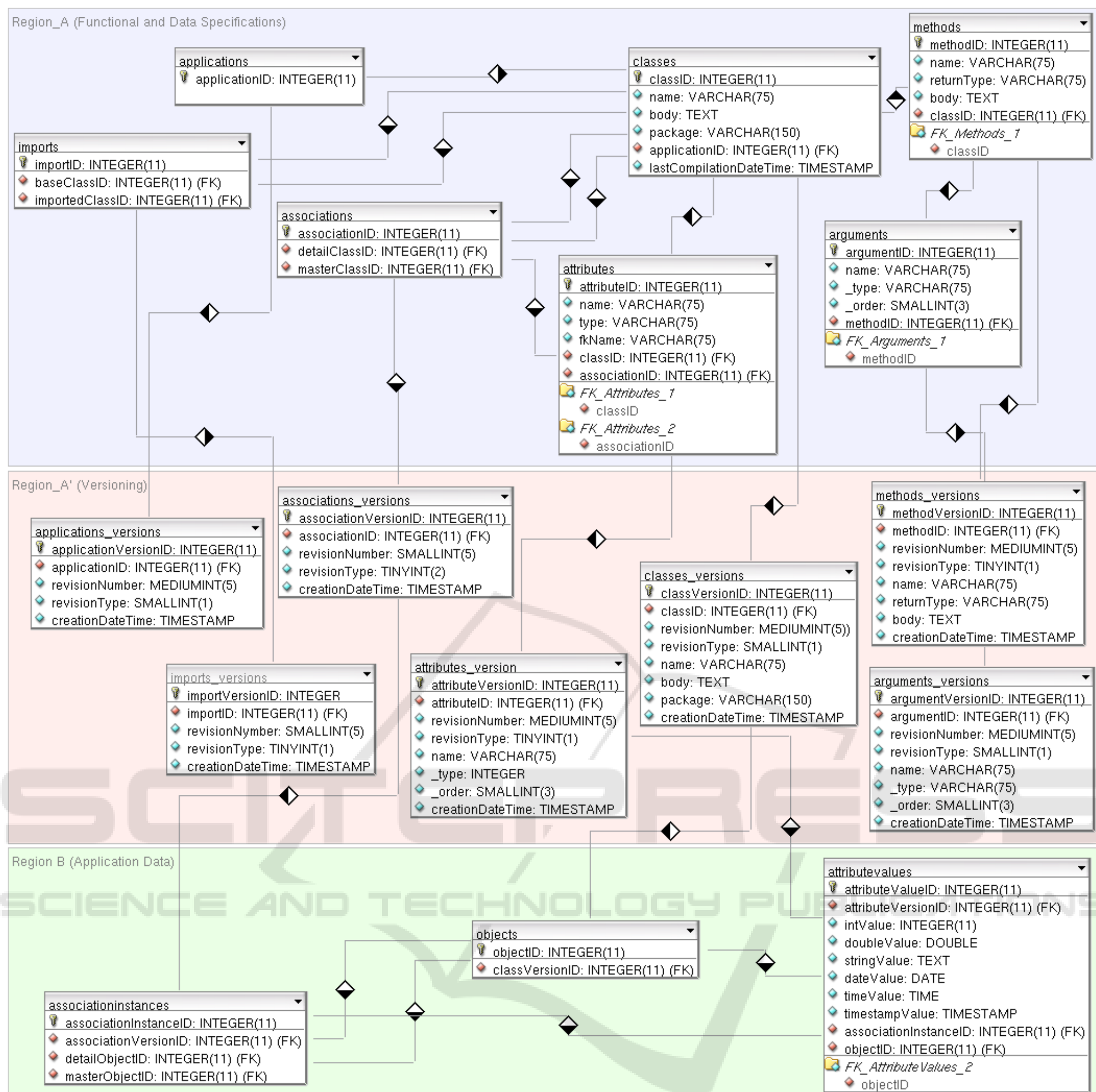
- primary key – auto number
- id of the original entity
- revision number – an integer
- revision type – a small integer: this field can have four values: 0, 1, 2, 3 which mean, respectively, ADDED, MODIFIED, DELETED and UNCHANGED. A row with a revision of type DELETED will only contain the id of the entity and no data (all fields NULL), as it only serves as a marker saying ”this entity was deleted at that revision”.
- creation date and time – timestamp
- versioned fields from the original entity

The current entity data is stored in the original table and in the versions table. According to Envers members, this duplication of data is acceptable, since it makes the query system much faster. However, this aspect will be tested with real production data since performance issues in versioning systems depend on how often old data is accessed.

The proposed technique supports global revisions, i.e., for each modification of a data instance a new revision is created that includes not only modified but also unchanged data. Global revisions are suitable for data that is not often modified. Since a real-word application is not modified on a daily basis, we believe that this technique meets our needs.

In Figure 1, we present an improved version of the Database Model (Voulalas and Evangelidis, 2008b). Region A holds the functional specifications of the modelled application. In includes the following entities: Classes, Attributes, Methods, Arguments, Associations and Imports (class dependencies).

Region A’ is the new part of the database model. It includes all tables that are required in order for versioning to be supported. Note that versioning applies only to Region A, i.e., to the part of the model that holds the functional and data specifications of the



SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Figure 1: The Database Model.

modelled applications. For each versioned table we have created a companion table using "_versions" as suffix. Note that:

- Although the combination of the primary key of the original entity with the revision number is unique and not null, we have created a new field to be used as primary key. This is because we wanted to avoid extra complexity produced by the complex primary keys (consisting of more than one fields) of the original entities.
- Foreign keys are not versioned. For example the methodID field in the arguments table is not versioned, since an argument is tightly associated with a method and cannot be moved to another method.
- The versions table that is associated with the associations table does not include any versioned field. This is because an association can only be deleted and not modified.

Region B includes tables that store data produced by the applications, structured in a way that is independent of the actual data structure of the applications. Thus, changing the database of a modelled application (e.g. adding a field in an existing table or creating a table) does not affect Region B. Versioning of the data that is produced by the deployed applications is out of the scope of our research.

3 SELECTIVE CLASS RECOMPILATION

In our previous research (Voulalas and Evangelidis, 2008b), we have presented a generic method that handles the execution of entry methods of the deployed applications. The method works as follows:

- Using the input arguments (className and packageName) identifies the class that implements the method and retrieves its source code from the database.
- Along with the main class, all imported classes are also retrieved from the database (see Imports table in Region A). Note that this applies only to application-specific classes and not to java classes or generic classes of our framework that do not change.
- All classes are recompiled and the method is executed using the getDeclaredMethod operation that is implemented by java.lang.Class. Arguments are passed as an array of java.lang.Class objects.

This simple approach needs to be improved in order to reduce or even zero unnecessary recompilations and avoid runtime errors (e.g. runtime type incompatibility, method not found) produced because not all dependent classes have been correctly identified and recompiled.

The main idea is that a class needs to be recompiled only in case it has changed itself or a class that is dependent on has changed. This requires for the following two improvements.

First, we have to query the database in order to see if the class or an associated class has changed until the last time it was compiled. For this reason we have added a field in the Classes table (Region A) that is used for tracking the exact date and time each class was compiled for the last time. The query just needs to select only those records from the classes table that are associated with the method to be invoked and satisfy the following condition: LASTCOMPILATIONDATETIME is smaller than the MAX(CREATIONDATETIME) of all versions of the class having revision type different than UNCHANGED. In case no such class exists the last class file can be loaded and used for the execution of the method.

Second, since classes that belong to the same package are not imported when used by a class, it is not safe to identify dependencies only by recursively analyzing imports. Recompiling by default all classes of the same package is not efficient, especially for large systems with many classes. Thus, we should use a tool for analyzing classes and finding all dependencies between them. Dependencies, for instance, stem from class inheritance or subtyping, method calls, and attribute accesses. Dependency information is typically gathered from source code. In the case of Java, it is possible to get this information from class files. Although the approach of finding dependencies based on the class files seems to be more effective (Barowski and Ii, 2002), in our case it is not applicable since dependencies are needed in order to decide which classes should be recompiled. Using old class files is not safe because the modified source code may include additional dependencies that will be ignored, resulting in runtime errors. Examples of existing tools that analyze class files are jGRASP (Barowski and Ii, 2002), CDA (Class Dependecy Analyzer) and STAN (STructure ANalysis). In contrast, OptimalAdvisor, RECODER and DA4J (Pinzger et al., 2008) identify dependencies by analyzing the source code files. IntelliJ IDEA is an advanced Integrated Development Environment (IDE) that supports compilation of all source files that have been modified since the last compilation in the selected module as well as in all modules it depends on recursively. Its current version (IntelliJ IDEA 7) includes the Dependency Structure Matrix (DSM), a sophisticated tool for visually analyzing the dependencies between project classes (IntelliJ IDEA, 2008).

Selecting and incorporating a technique that identifies dependencies between Java Classes is one of our future steps. The main point is that the preliminary research that we have conducted shows that such tools and methods already exist.

4 HANDLING OF ACTIVE CLASSES

A class that is replaced at run-time may have active methods. Thus, the biggest problem of runtime evolution is to match the old and the new program code and state. In a relevant research effort (Dmitriev, 2001), three policies for dealing with active old methods of changed classes are presented. Although the particular effort is focused on the implementation of an environment that supports evolution of Java applications at a lower level (as part of the Java Virtual Machine) compared to our research, the presented policies are quite interesting and relevant:

On-the-fly Method Switching. According to this policy a point in the new method that "corresponds" to the current execution point in the old method, should be identified. Then, execution is continued from this

point in the new method. Identifying this point is quite hard and even if this is done, it seems that many security gaps exist.

Wait Until there are no Active Old Versions of Evolving Methods. This policy guarantees that two versions of any method can never co-exist simultaneously for a given application. In order for this policy to be applied in practice, the developer must somehow ensure that the execution will reach the point where no active old methods exist. The implementation of this policy would have to keep track of all of the activations of the old methods. Once the last such activation is complete, the threads should be suspended and method replacement should be performed. Again, this task is more complex in case of a multi-threaded application, since it may happen that while one thread completes the last activation of the old method, another thread calls this method once again. Also, this solution may not always work (we may wait forever), if, for example, one of the evolving methods is the main method of the program.

All Existing Threads Continue to Call Old Code, whereas New Threads (Threads Created after Class Transformation) Call New Code. This policy looks more difficult to implement than the others, since it allows the old and the new code to coexist, and the old code can be called over and over again. It is currently unclear, if possible at all, to determine whether or not an arbitrary thread may call a given class. Perhaps the developer should specify this explicitly. This policy may be the most suitable solution for certain kinds of applications, e.g., servers that create a new, relatively short-lived thread in response to every incoming request.

It looks that none of these solutions can be regarded as panacea. They are different policies and each one of them may be preferable in certain situations. Evaluating them in the context of our research project, it seems that the third policy (all existing threads continue to call old code, whereas new threads call new code) best suits the needs of the family of applications that we are targeting to. Web-based applications, in general, generate short-lived threads in response to different incoming requests initiated by end-users (e.g., when a user submits data through a web form, the system creates a thread that parses submitted data and stores it in the database).

Now, we have to find out a way to support this policy. Returning back to the database model we can see that objects (in Region B) are associated with the classes_versions table (in Region A) and not with the classes table that was our initial approach in the first version of the database model (Voulalas and Evangelidis, 2008b). This modification enables us to identify all objects, including the values of their attributes and the way they are associated with other objects, that have been produced from a specific version of an application. Thus, this information can be used in order to identify which version of the application should be invoked in order for a specific object to be processed.

Before concluding this section we must make an important note. Suppose that we have a web-based business application that supports the submission of requests by public (external) users and the processing of the requests in multiple steps by enterprise (internal) users. A business change forces the developer to implement and make available a new version of the application without interrupting the operation of the system. According to the policy presented above, the running instances of the objects will be handled by the old version of the application. This means that, a request submitted by a user before the system change, will be parsed and stored in the database by the methods of the old classes. Based on the same policy, when the request will be fetched from the database in order to be processed by an internal user, the new object instance will be created by the new version of the class. However, in our platform the developer may have another advanced option: depending on the type of the change he can declare that objects created by a previous version of the application should be processed throughout their lifecycle by the same version of the application. This extra option is very useful in case of changes that seriously affect business rules. Defining the lifecycle of an object is a critical issue that should be examined, since it may involve not only the specific object but also associated objects (with their own lifecycle). This implicitly entails that we should consider introducing the concepts of roles, activities and workflows that are present in all business systems either embedded in code, or handled by a discrete workflow system.

5 CONCLUSIONS & FURTHER RESEARCH

Our research effort aims to provide a platform that will support the development and deployment of continuously evolving web-based business applications. The operation of the platform complies with the following rules:

- Source code resides in the database and is retrieved and compiled at run-time. Changes are dynamically embedded in the application without the need of interrupting its operation.

- The platform facilitates the developer in the implementation process by providing pre-build mechanisms for the interaction with the database. The user does not have to design a database since all applications rely upon a common database model.
- Multiple applications and multiple versions of the same application can operate in parallel upon a single installation of the platform.

Code modifications can be either simple changes, for example, additional System.out.println() statements for better logging, or even complex changes that require the implementation of a new method or new information to be stored in the database. Even in the case of simple changes, it is of great importance to implement them without interrupting the operation of the system.

Having implemented an architectural prototype that enabled us to verify the feasibility of the proposed solution in general (Voulalas and Evangelidis, 2008b), this paper elaborated on three important aspects:

Versioning of the Modelled Applications. We presented a simple data versioning technique that allows us to keep all different versions of the modelled applications. We modified the database schema in order to embed the versioning mechanism.

Selective Class Recompilation. We presented a method for selecting only those classes that have been modified after being compiled. Further work is required in order to embed a process for identifying class dependencies. Having this information will enable us to safely select all classes that need to be recompiled.

Managing Active Instances of Classes that have to be Recompiled in Order to Reflect Changes. We selected a policy that defines that all existing threads should continue to call old code, whereas new threads (threads created after class transformation) should call new code. We presented the way this policy can be efficiently supported by our system, plus some extra thoughts for extending the policy in order to enable complex business processes to evolve more smoothly.

Our next research effort will focus on testing the feasibility of the platform with some real application scenarios. Based on the case study that we have presented in our previous work (Voulalas and Evangelidis, 2008b), we are going to apply all possible types of changes in order to evaluate how the system reacts. We should also validate in practice how the system operates when two different versions of the same application co-exist, especially when they have active instances in parallel.

REFERENCES

Atkinson, M. and Jordan, M. (2000). *A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persisteng Programming Platform*. Sun Microsystems, Inc., Mountain View, CA.

Barowski, L. A. and Ii, J. H. C. (2002). Extraction and use of class dependency information for java. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 309, Washington, DC, USA. IEEE Computer Society.

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 *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*.

IntelliJ IDEA (2008). Dependency analysis. `http://www.jetbrains.com/idea/features/dependency_analysis.html`.

Pinzger, M., Graefenhain, K., Knab, P., and Gall, H. C. (2008). A tool for visual understanding of source code dependencies. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 254–259, Washington, DC, USA. IEEE Computer Society.

Sun Microsystems (2008). The reflection API. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

Voulalas, G. and Evangelidis, G. (2007). A framework for the development and deployment of evolving applications: The domain model. In *ICSOFT International Conference on Software and Data Technologies*, Barcelona, Spain.

Voulalas, G. and Evangelidis, G. (2008a). Developing and deploying dynamic applications: An architectural prototype. In Filipe, J., Shishkov, B., and Helfert, M., editors, *Communications in Computer and Information Science*, volume 10, pages 293–306. Springer Berlin Heidelberg.

Voulalas, G. and Evangelidis, G. (2008b). Developing and deploying dynamic applications: An architectural prototype. In *ICSOFT International Conference on Software and Data Technologies*, Porto, Portugal.

Warski, A. (2008). Data versioning and envers. `http://www.jboss.org/downloading/?projectId=envers&url=/envers/downloads/presentations/envers_nurnberg.pdf`.

Zhu, N. (2003). Data versioning systems. Technical report, Computer Science Department, Stony Brook University.