

An MDA Approach for Database Profiling and Performance Assessment

Fabio Perez Marzullo

Federal University of
Rio de Janeiro – UFRJ,
COPPE Database
Laboratory, Brazil
fpm@cos.ufrj.br

Rodrigo Novo Porto

Federal University of
Rio de Janeiro – UFRJ,
COPPE Database
Laboratory, Brazil
rodrigo@cos.ufrj.br

Geraldo Z. da Silva

Federal University of
Rio de Janeiro – UFRJ,
COPPE Database
Laboratory, Brazil
zimbrao@cos.ufrj.br

José Roberto
Blaschek

State University of Rio
de Janeiro – UERJ –
FAF, Brazil
blaschek@attglobal.net

Jano Moreira de Souza

Federal University of
Rio de Janeiro – UFRJ,
COPPE Database
Laboratory, Brazil
jano@cos.ufrj.br

Abstract— This paper describes a Model Driven Architecture (MDA) approach for assessing database performance. The increasingly area of component-based development is reaching to a point where performance issues are critical to successful system deployment. It is widely accepted that the model development approach is playing an important role on IT projects; therefore the profiling technique discussed here, presents a way of assessing performance and identify flaws while performing software construction activities. The approach is straightforward: using new defined MDA stereotypes and tagged values, in conjunction with the JAMon [11] and the InfraRED [15] profiling libraries, the proposed MDA extension enables code generation to conduct a thorough set of performance analysis. This new implementation uses the well know MDA framework AndroMDA [2], the Hibernate Cartridge and creates a Profiling and Testing Cartridge, in order to generate the assessment code.

Index Terms— MDA, Database, Profiling, benchmarking, performance testing, code generation.

This work was supported by the Federal University of Rio de Janeiro – UFRJ, COPPE Database Laboratory. Database Performance Assessment Using MDA.

Fabio Perez Marzullo, M. Sc. is with Federal University of Rio de Janeiro Rio de Janeiro, Brazil. (e-mail: fpm@cos.ufrj.br).

Rodrigo Novo Porto is with Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, (e-mail: rodrigo@cos.ufrj.br).

Geraldo Zimbrão da Silva, D. Sc. is with Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, (e-mail: zimbrao@cos.ufrj.br).

José Roberto Blaschek, D. Sc. is with State University of Rio de Janeiro, Rio de Janeiro, Brazil, and with the Faculdade de Administração e Finanças (e-mail: blaschek@attglobal.net).

Jano Moreira de Souza, Ph.D. is with Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, and is the head of the Database and Knowledge Management Department (e-mail: jano@cos.ufrj.br).

I. INTRODUCTION

Researches conducted along recent year have shown that model approaches are becoming essential tools in software development [1]. This new paradigm implies on new ways of analyzing overall software performance. Attempts to integrate performance analysis with MDA have been done, and are still in course [3]. However, such attempts present, mostly, ways on generating test code regarding general software engineering aspects. The relevant scenario here is this: how do we assess performance on database specific aspects, such as joins, storage, memory load and so fourth? And mostly, how measuring such database aspects guides overall software architecture design, improving model and generated code?

The lack of an efficient representation for Database profiling has motivated us on extending the MDA models in order to cope with undesirable performance situations along software execution. Therefore, the idea relies on an MDA extension with the ability to mark business domain models with stereotypes and tagged values, seeking understanding of database performance aspects.

Since long the OMG has been revising requests and proposals to Software Profiling. As an example, the *UML 2.0 Testing Profile Specification* [4] defines a test modeling language that can be used with all major UML objects and component technologies and applied to testing systems in various application domains. However its objective is to create a testing environment that is concerned on software engineering aspects.

Despite of what is being done, there are still gaps involving proper database analysis. Still, different approaches focusing on benchmarking MDA techniques [5], or testing and verification of model transformation [6] is currently in activity, and our work comes to join all efforts in upgrading the MDA world.

Although MDA promises to implement a self-sufficient model driven development theory, supported now by a few important companies and tools, it still needs to address several specific development aspects, including an efficient and complete set of Meta Object Facility [7] models and objects to address database specifics.

II. PROBLEM DEFINITION

Since the beginning of computational growth, there have been efforts addressing performance tuning. Regardless of the hardware or software involved, the pursuit of a systematic approach on writing code, which efficiently implements software functionality, still lingers in today's projects. Few techniques are available to help developers on writing better code; hence, one that has been gaining visibility is the profiling technique. Many companies are now engaged on creating tools to aid software profiling; enabling developers to identify system flaws and rewrite problematic code [8, 9, 10].

Consequently, challenges faced by IT managers when developing software in this scenario, has only increased. Due to painstaking, detailed and domain specific functionalities, and ever growing knowledge consuming, it became urgent to invest in new ways to cope with the time to market of information systems without lacking proper product quality [11].

The promise of a model development approach has gained many adepts. The AdroMDA project [2] is on the top of model development tools, based on the OMG MDA specification [1]. From single CRUD application to complex enterprise applications, it uses a set of ready-made cartridges (which implements highly used java development API) to automatically generate up to 70% of software source code. Expectations are that all efforts should point to an interesting future where the modeling and coding stages will be merged without significant losses.

The same horizon is faced by database experts. The old promise of uniting the Object Oriented development and the Relational data representation is now very well taken care of by the Hibernate API [13, 14]. Also, different persistence technologies have been developed to relinquish, from the ordinary developer, the overhead of dealing with database laboriously queries and joining structures.

Nevertheless, in spite of all promises of database abstraction, it is not always possible to guarantee that what you see is what you get. For example, the Hibernate is a powerful, high performance, object/relational persistence and query service [13], however, using it in its default configuration might prove that working with huge amounts of data is cumbersome.

It lacks the ability to identify and target inefficient table joins, which may imply in performance degradation. It is known that after a few adjustments on configuration values it may become more reliable; however the work load might be significant.

Given the above analysis, it is necessary to explain the problematic scenario that we have been faced in our project. The AndroMDA environment comprises of a set of specialized cartridges that are used to generate the software code. One of its cartridges is the Hibernate Cartridge, which is responsible of generating all the database manipulation code. As for database code generation all seemed perfect, the development database was used to test the system while in construction. Only three to five users were simultaneously accessing the database and development seemed to go smoothly. When the system was deployed and put into production, a small set of functionalities presented longer response times than we were expecting. The production environment needed to be accessed by 10 to 20 users simultaneously and had to deal with thousands, even millions of records to be queried or joined. It suffices to say that the performance degradation perception was prompt, and as such, something should be done to locate and solve the problem source.

To solve this problem we began stating three topics that needed to be addressed immediately:

1. How do we isolate the functionalities and their problematic attributes?
2. How could we solve these problems in an elegant and transparent way for the end user?
3. How do we automate the solution extending the MDA objects to address Database Testing and Analysis?

Using the current infra-structure to address the database performance analysis was not the best way, since we wanted to create a systematic and self sustained analysis approach. It was necessary to extend the MDA infra-structure, more precisely the Hibernate Cartridge, and implement a new Profiling Cartridge as explained in the next section.

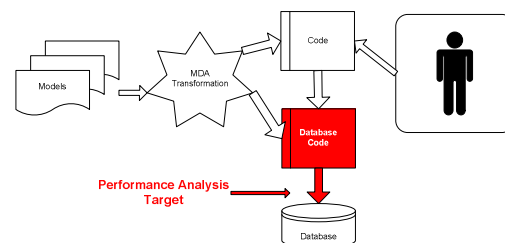


Figure 1. Schematics of the performance analysis target.

III. THE PROFILING EXTENSION

The profiling extension process was straightforward. We needed to create new design elements to indicate when it would be necessary to embed monitoring code inside the software. For that purpose we followed a five-step design process, in order to extend the MDA framework:

1. *Defining the profiling library.* This phase was responsible for identifying the profiling libraries that would be used. Our objective was to create an infrastructure that could cope with any profiling library available, so we decided to use design patterns to enable the ability to plug the library as necessary. This technique is explained in the following sections. However, after analyzing several libraries two appeared to be the best choices: the JAMon Library [11] and the InfraRED Library [15].

2. *Defining Stereotypes.* This phase was responsible for creating all the stereotypes necessary to model configuration. For each feature available a stereotype was created:

1. *API timing:* Average time taken by each API. APIs might be analyzed through threshold assessment and first, last, min, max execution times:

→ `<<APIView>>`: which enables api monitoring;

2. *JDBC and SQL statistics:* Being the objective of our research, we created the following stereotype for accessing JDBC and SQL statistics:

→ `<<SQLQueryView>>`: which enabled the displaying of the created hibernate queries;

3. *Tracing and Call Information:* responsible for showing statistics for method calls. The following stereotype was created to mark/enable this option:

→ `<<TraceView>>`: which enabled a detailed call tracing for method calls;

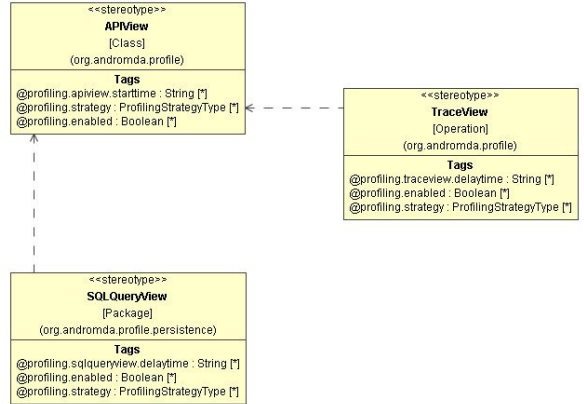


Figure 2. Stereotypes and tagged values hierarchy.

3. *Defining Tagged Values.* This phase was responsible for creating all tagged values necessary to support and configure stereotypes:

1. `@profiling.active`: defines whether the profiling activities are going to be executed. Setting its value to “yes”, implies in generating the profiling code, and consequently enabling the profiling activities; setting to “no”, the code will not be generated. Applies to `<<APIView>>`, `<<SQLQueryView>>` and `<<TraceView>>` stereotypes;
2. `@profiling.apiview.starttime`: defines the starting time in which the profiling code should initiate monitoring. For example: it is not interesting monitoring every aspect of the system, therefore we indicate the minimum limit in which the profiling code should initiate the logging process. Applies to the `<<APIView>>` stereotype;
3. `@profiling.sqlqueryview.delaytime`: defines the starting time in which the profiling code should initiate SQL monitoring. For example: it is not interesting monitoring all query execution in the system. It is only necessary to assess queries that surpass a certain delay threshold. Applies to the `<<SQLQueryView>>` stereotype;
4. `@profiling.traceview.delaytime`: defines the starting time in which the profiling code should initiate Trace monitoring. For example: it is not interesting monitoring all method calls in the system. It is only necessary to assess the calls that surpass a certain delay

threshold. Applies to the `<<TraceView>>` stereotype;

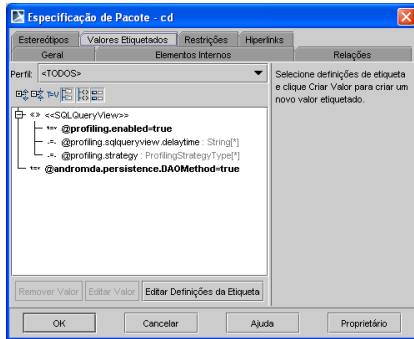


Figure 3. Associating the stereotype to the persistence package.

4. *Adjusting the Hibernate Cartridge.* This phase was responsible for extending the hibernate cartridge so it would be able to cope with the new stereotypes. The Hibernate API works through the concept of *Sessions*, where a set of instructions, referring to database interactions, creates the notion of a unique Session. Normally, when a database connection is opened a Hibernate Session is open to accommodate that database connection. It stays opened while the hibernate remains accessing the database. In face of such rules, we have realized that we would have, not only to analyze the software in class-based scope, but we needed a broader and more complete approach in which the profiling would be able to gather information globally. The profiling scope should reach the entire persistence layer. This constraint motivated us to design the stereotype as having *package* scope, not only class scope.

The solution involved the extension of the AndromDA persistence base package (`andromda-profile-persistence`) in order to include support for our profiling stereotype and tagged value. That allowed us to embed the necessary profiling code in the hibernate generated code.

5. *Creating new configurable profiling cartridge.* This phase was responsible for creating the profiling cartridge responsible for generating the monitoring code. The cartridge is explained in the following section.

IV. THE PROFILING CARTRIDGE

The Profiling Cartridge was built on top of the AndromDA framework [2]. According to the background presented previously, we had two profiling libraries candidates to use. Both presented efficient and complementary functionalities, so we decided to design

a way to interchange both libraries as necessary. The configuration strategy, as the term describes, was to use the well known *strategy* pattern. With this approach, we managed to set the profiling library by means of a tagged value:

- `@profiling.strategy`: defines which library should be used during profiling analysis. Assigned values are: `{JAMON, INFRARED}`.

Figure 4 illustrates the strategy of interchanging profiling libraries:

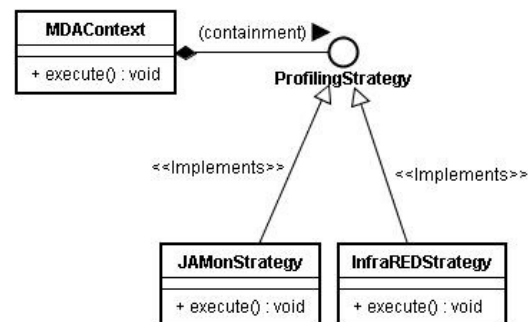


Figure 4. Strategy pattern for profiling configuration.

The specification of the profiling configuration details the functional relation between the MDA, as the source model, and the profiling technique, as the target implementation. This configuration method enabled us to interchange any profiling library of interest.

As for the cartridge implementation in AndromDA we executed six steps, in which the basic process for cartridge development was fulfilled:

1. *Identify, design and generate PSM metaclasses.* This first step was useful to plan how the profiling libraries would be used and mapped as stereotypes and tagged values;

2. *Identify transformation rules.* After planning profiling attributes, we have created all transformation rules needed to generate the code to the corresponding stereotype and tagged value;

3. *Model, generate and write metafacades.* Modeling stereotypes and tagged values was the first real difficulty encountered along the extension process. Deficiencies were uncovered and solved to match every PSM metaclasses and transformation rules planned earlier;

4. *Write templates.* This step was responsible for using Velocity as template language to reference objects

defined in Java code [18]. Templates for both profiling libraries were created; and java code generation could be set according to the strategy assigned to the `@profiling.strategy` tagged value;

5. *Write deployment descriptors.* The AndroMDA core uses cartridge descriptors to discover available capabilities such as: the supported metafacades, stereotypes, outlets, templates, property references, etc. The cartridge descriptor must reside in the META-INF/andromda subdirectory and must be named cartridge.xml [2].

6. *Deploy cartridge.* This step is responsible for deploying the new cartridge into the AndroMDA environment.

V. PROFILING ANALYSIS AND RESULTS

For the definition of the analysis process, we must understand the execution scenario. The development was focused on creating a web-based application, using Struts [19], running on the Jboss Application Server (JAS) [20], and accessing a Data Warehouse. System modeling was done using an UML tool, the AndroMDA, and the maven tool [16], and for java development we used the eclipse IDE [17]. The data load, was irrelevant at development time, however it became crucial by the time the system was put into production. The generated hibernate code and configuration did not comprise with the system's response time non-functional requirements.

As soon as the system performance started to interfere with overall system usability, we found urgent to locate the bottleneck source and refactor all involved models. On the other hand, we needed to ensure that the new modeling would not create new bottleneck sources.

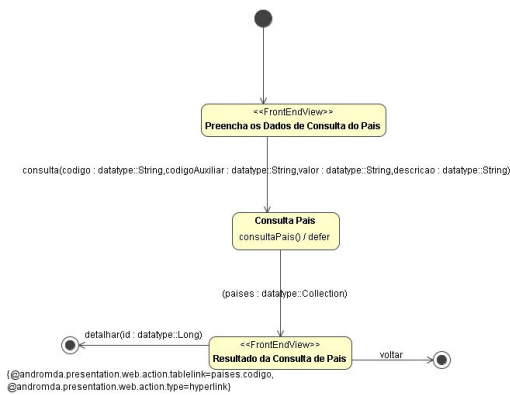


Figure 5. Use Case used as Test Case.

By means of the profiling technique, we were able to embed monitoring code that gathered all necessary

information and pointed the features that needed to be remodeled. The analysis below shows what the profiling technique is capable.

Statistics were acquired along two different scenarios: (1) less than 10,000 registers in a non optimized and an optimized environments; (2) more than 2,000,000 registers with a non optimized and an optimized environments. After performing measurements in both scenarios, we came up with averages presented bellow. Tables 1-A and 1-B presents three functionalities that were assessed to test the MDA-Profiling strategy:

Functionalities	Expected Execution Time	Actual Execution Time	Problematic Functionality
Single Table (F1)	10 ms	30 ms	No
Join with 2 Tables (F2)	400 ms	50,000 ms	Yes
Join with 3 Tables (F3)	600 ms	10,000,000 ms	Yes

Table 1-A. Identifying problematic functionalities. Functionality names were omitted for confidentiality purposes (Non-Optimized Environment). Time values averages were rounded up.

Functionalities	Expected Execution Time	Actual Execution Time
Single Table (F1)	10 ms	23 ms
Join with 2 Tables (F2)	400 ms	1000 ms
Join with 3 Tables (F3)	600 ms	2000 ms

Table 1-B. Solving problematic functionalities. Functionality names were omitted for confidentiality purposes (Optimized Environment). Time values averaged were rounded up.

Each system functionality was assigned an estimated execution time, based on the system non-functional requirement, as shown in Table 1-A. The system was deployed in production environment and measured without the optimized code. Those functionalities that had overcome the estimated value by 20% were marked as problematic (that in our case easily overcame this requirement). After, we regenerated the code and redeployed the system. Obtaining the actual execution times we observed that functionalities F2 and F3 were highly out of the required range.

Analyzing the JAMon profiling results we discover a number of items that could be refactored in order to produce better optimized code. For instance, we have detected that the Hibernate session was not reusing SQL statements, and was creating new SQL plans each time a query was submitted to the database.

Values presented in table 1-B show that the new optimized code reduced significantly the execution time. It showed a large reduction in database resource consuming, taking into consideration that in a web-based

environment such optimization might prove to be extremely relevant.

VI. CONCLUSIONS

This paper presented a new approach of embedding profiling techniques in MDA development. Our contribution aimed on creating a MDA extension to help identifying and solving performance problems regarding information system and database (data warehouse) communications. In addition, we defined an extensible, easy-to-use profiling infra-structure that can be configured to execute different profiling libraries and techniques, obtaining a more complete set of results.

The analysis results have validated the profiling approach and proved that the MDA extension might be used to analyze the code as soon as it is deployed. The initial effort to create the infra-structure proved laborious, however following developments shall not suffer the same problems, as it has already been implemented and added to the AndroMDA features.

Finally, the intention was to obtain development information in order to allow developers and analysts to make proper decisions regarding software design. According to the analysis results, the extension was able to expose flaws and delays during system execution, and, consequently, promote the necessary corrections to ensure that the generated code was reliable in both scenarios.

Future works point to the use of aspect based development to add considerable new techniques on profiling the system and using autonomic computing, in conjunction with the profiling libraries, in order to detect the problems, understand the problem sources and use embedded intelligence to allow the system to self correct and recover from eventual failures.

REFERENCES

- [1]. "Model Driven Architecture", <http://www.omg.org/mda>.
- [2]. "AndroMDA, v3.0M3", <http://www.andromda.org/>.
- [3]. Zhu, L., Liu, Y., Gorton, I., Bui, N. B., "MDAbench, A Tool for Customized Benchmark Generation Using MDA", OOPSLA'05, October 16-20, 2005, San Diego, California.
- [4]. "UML 2.0 Testing Profile Specification". <http://www.omg.org/cgi-bin/doc?formal/05-07-07>.
- [5]. Rodrigues, G. N., "A Model Driven Approach for Software System Reliability", Proceedings of the 26th International Conference on Software Engineering (ICSE'04), IEEE 2004.
- [6]. Lamari, M., "Towards an Automated Test Generation for the Verification of Model Transformations". SAC'07, ACM 2007.
- [7]. "Meta Object Facility", <http://www.omg.org/mof>.
- [8]. "Eclipse Test & Performance Tools Platform Project" <http://www.eclipse.org/tptp/>.
- [9]. "ej-technologies JProfiler", <http://www.ej-technologies.com/products/jprofiler/>.
- [10]. "JAMon (Java Application Monitor)", <http://jamonapi.sourceforge.net/>.
- [11]. "NetBeans Profiler", <http://profiler.netbeans.org/>.

- [12]. Frankel D. S., "Model Driven Architecture – Applying MDA to Enterprise Computing", OMG Press, Wiley Publications. 2003.
- [13]. "Hibernate", <http://www.hibernate.org>.
- [14]. Bouer, C., King, G., "Hibernate in Action". Manning Publications Co. 2004.
- [15]. "InfraRED – Performance and Monitoring Tool for Java", <http://sourceforge.net/projects/infrared/>.
- [16]. "Maven project management and comprehension tool", <http://maven.apache.org>.
- [17]. "Eclipse Project". <http://www.eclipse.org>.
- [18]. "Velocity Project". <http://velocity.apache.org/>.
- [19]. "Struts Project", <http://struts.apache.org/>.
- [20]. "JBoss Application Server", <http://www.jboss.org/>.