

# A Practical MDA Approach for Autonomic Profiling and Performance Assessment

Fabio Perez Marzullo<sup>1</sup>, Rodrigo Novo Porto<sup>1</sup>, Divany Gomes Lima<sup>1</sup>,  
Jano Moreira de Souza<sup>1</sup>, José Roberto Blaschek<sup>2</sup>

<sup>1</sup> Federal University of Rio de Janeiro – UFRJ, COPPE Database Laboratory,  
Rio de Janeiro, RJ, Brazil  
{fpm, rodrigo, dglima, jano}@cos.ufrj.br

<sup>2</sup> State University of Rio de Janeiro – UERJ, Rio de Janeiro, RJ, Brazil  
blaschek@attglobal.net

**Abstract.** By releasing their Model Driven Architecture (MDA) as a new standard, the Object Management Group (OMG) [1] proposed a new development concept toward existing traditional paradigms. It set a new exciting research area in which it would be possible to develop truly independent and powerful programming environments capable of achieving new levels of productivity, performance and maintainability. With this goal in mind, this paper describes a research conducted with the purpose of improving database performance through the union of autonomic computing aspects and MDA. It is widely accepted that the model development approach is gaining importance in IT projects today; therefore the technique discussed here, presents a way of assessing performance, identifying flaws and improving software activities in order to create a self-managed environment. With new defined stereotypes and tagged values; in conjunction with profiling libraries, and relying on autonomic aspects, the proposed extension enables code generation in order to conduct a thorough set of performance analysis, indicating the best suitable database configuration for a given functionality. After setting the underlying problem, explaining tools configuration and concepts and describing the profiling technique, it presents a study based on a real project conducted by the Brazilian Ministry of Defense.

**Keywords:** MDA, Database, Profiling, benchmarking, performance testing, code generation.

## 1 Introduction

Research conducted in recent years has 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 to generate test code regarding general software engineering aspects.

The lack of an efficient representation for Database profiling has motivated us to extend the UML models (as implemented in the development environment) in order to cope with undesirable performance situations along software execution. Therefore, the idea relies on an extension with the ability to mark business domain models with stereotypes and tagged values, seeking to understand database performance aspects.

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

Also, as stated by the IBM Autonomic Computing Manifest, we have come to a point at which the IT industry creates powerful computing systems on a daily basis. In order to make individuals and business more productive, by automating their practices and processes, we face paths ahead showing that the autonomic approach might prove to be valuable in the near future [21].

By pursuing the vision of creating intelligent solutions with self-management capabilities, the proposed extension, defines a set of general rules and techniques that creates a self-configuration and self-assessment environment, capable of identifying the best system-database interaction according to the functionality involved.

## 2 Problem Definition

Since the beginning of computational growth, companies are engaged in creating tools to aid software profiling; creating autonomic algorithms; and enabling developers to identify flaws and re-factor problematic systems [8, 9 and 10].

The promise of an efficient model driven development and the comparable effort of researching good autonomic practices have gained many adepts. The AdroMDA project [2] is one of few important model development tools, based on the OMG MDA specification [1]. From single CRUD (Create, Read, Update and Delete) application to complex enterprise applications, it uses a set of ready-made cartridges (which implements highly widespread of Java development API) to automatically generate up to 70% of software source code. It is expected 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 autonomic computing experts. Although it might be considered a young research area, the autonomic computing perspective has brought to discussion the necessity of creating new techniques to cope with Information Technology increasing complexity.

Given the foregoing, it is necessary to explain the problematic scenario that we have faced with in our project. Our projects use an MDA Framework composed by the AndroMDA environment, the Maven tool [16] and the Magic Draw Case Tool [24]. 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 for generating all the database manipulation code. As for database code generation all seemed perfect, the development database was used to test the system while as it was being constructed. 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 had anticipated. 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 performance degradation perception was immediate, and as such, something should be done to locate and solve the root problem.

To solve it we began by stating three topics that needed to be immediately addressed:

1. How to isolate the functionalities and their problematic attributes?
2. How to solve these problems in an elegant and transparent way for the end user?
3. How to automate the solution extending the UML 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 adjust the Hibernate Cartridge, and implement a new Profiling Cartridge, embedding autonomic aspects, in order to create a self-configuring, self-healing and self-optimization infrastructure, as explained in the next section.

### 3 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 stage was responsible for identifying the profiling libraries that would be used. Our objective was to create an infra-structure 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 stage was responsible for creating all the stereotypes necessary to model configuration. For each feature available a stereotype was created:

#### 4 A Practical MDA Approach for Autonomic Profiling and Performance Assessment

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;

3. *Defining Tagged Values*. This stage 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 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 at which the profiling code should initiate monitoring. For example: it is not interesting to monitor 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 at which the profiling code should initiate Trace monitoring. For example: it is not interesting to monitor 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;

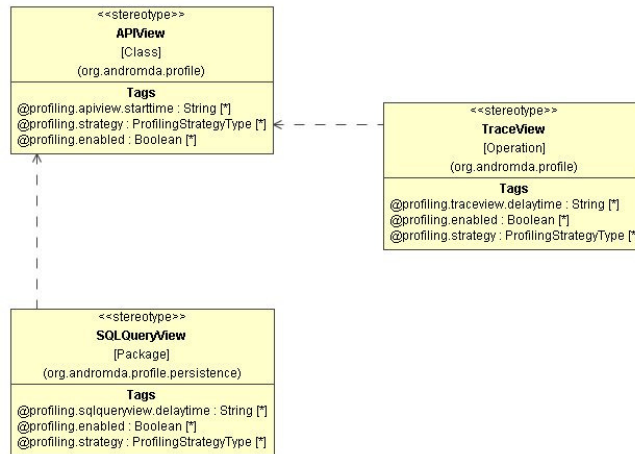


Fig. 1. Stereotypes and tagged values hierarchy.

4. *Adjusting the Hibernate Cartridge.* This stage 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 an unique Session. Normally, when a database connection is opened, a Hibernate Session is opened to accommodate that database connection. It stays open as the hibernate continues to access the database. Given those rules, we realized that we would have, not only to analyze the software according to a 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 a *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 stage was responsible for creating the profiling cartridge responsible for generating the monitoring code. The cartridge is explained in the following section.

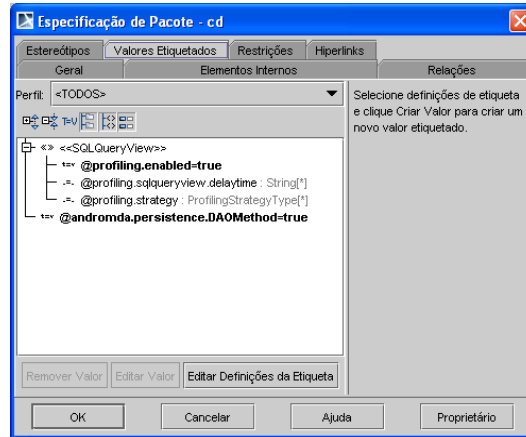


Fig. 2. Associating the stereotype to the persistence package.

Next, an autonomic environment should be set in order to create the self-managed system-database relationship. This environment was based on the following aspects:

1. *Self-Configuration Aspect*. This aspect was responsible for identifying the best hibernate session configuration according to functionality's database access needs. Our objective was to create an infra-structure that could cope with the system demands, such as table joins and cache, and were also able to remember this configuration after resources have been released and/or system has been shutdown.

2. *Self-Optimization Aspect*. This aspect was responsible for identifying the best configuration values for previously detected attributes. It should also persist long after system shutdown.

3. *Self-Healing and Self-Protection Aspects*. These aspects are not specifically attended in this implementation. It is still under development. There is enough literature addressing connection breakdown, web server crashes recovery and network disconnection. IBM Tivoli [22] is one that covers such elementary hardware issues.

## 4 The Autonomic Profiling Strategy

The autonomic profiling approach 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 and 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}.

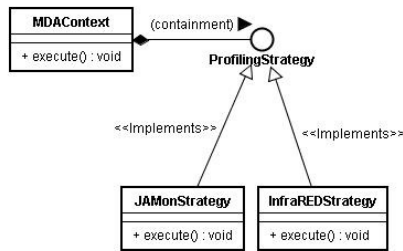


Fig. 3. illustrates the strategy pattern of interchanging profiling libraries and profiling configuration.

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.

Also, the configuration process was responsible for identifying the best system-database relationship for the specified functionality. The configuration and optimization processes are best explained in figures 5 and 6:

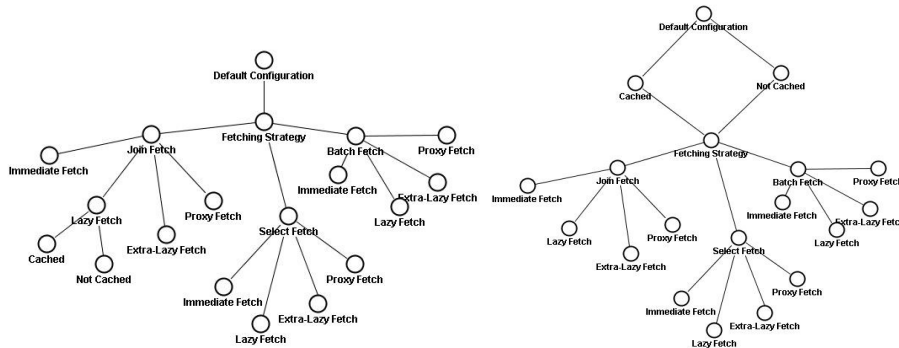
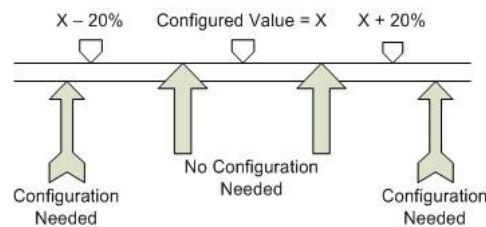


Fig. 4. Decision Tree for configuration definition. Configuration takes into consideration a fetching strategy first and a cache strategy first.

The overall idea is this: (1) the hibernate configuration initiates in its default values; (2) when the source code is generated, it embeds all the autonomic code that will measure any configuration attribute and evaluate it to find the best performance value; (3) the profiling code creates a decision tree and identifies the best configuration for that functionality; it first evaluates a fetching strategy as first perspective and after a cache strategy as first perspective. After acquiring the assessment measures, it identifies the best perspective solution; and (4) it then stores the configuration for future access, whenever that functionality is called.

Additionally, we needed to guarantee that the configuration kept optimized through software execution. Meaning that, when in production, the system database would probably have the number of records increased, and join strategies could have its execution time affected. Therefore, the autonomic configuration policy should be re-executed and reconfigured in order to guarantee a best perspective configuration.

For example, whenever a new software release was put into production the configuration process was executed and used as base configuration for accessing functionality's data. This value is then used for assessing whether the actual execution time is well adjusted with the stored configuration. So, whenever the functionality was called, a comparison was made between the actual time and the configured time. If the actual time falls below or above 20% of the configured value the system reruns the configuration process searching for a new configuration more suitable for that functionality.



**Fig 5.** This figure presents the range of values used to guarantee that the best accessing configuration was kept along system use.

## 5 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 Relational Database through Hibernate API [13, 14]. 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, but 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 recreate all involved models. On the other hand, we needed to ensure that the new modeling would not create new bottleneck sources.

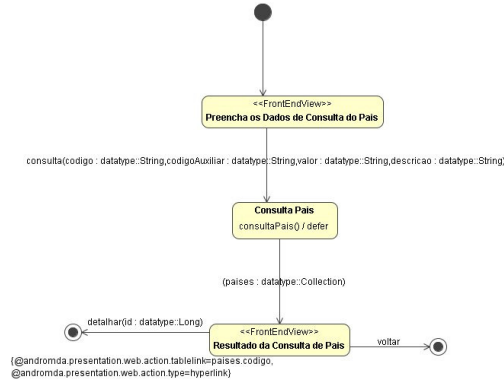


Fig 6. Use Case used as Test Case.

The profiling technique enabled us 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 of.

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

**Table 1.** Non-Optimized Environment: Identifying problematic functionalities. Functionalities F2 and F3 show that the real execution time were alarmingly out of range. Functionality names were omitted for confidentiality purposes. Time values averages were rounded up.

Functionalities	Expected Execution Time	Actual Execution Time	Problematic Functionality
Single Table (F1)	10 to 50 ms	30 ms	No
Join with 2 Tables (F2)	100 to 1000 ms	50,000 ms	Yes
Join with 3 Tables (F3)	500 to 1000 ms	10,000,000 ms	Yes

**Table 2.** Optimized Environment after reconfiguration: Solving problematic functionalities. After database access optimization, functionality F3 still presented an execution time 100% higher the estimate one. We assumed it as estimation error and corrected the non-functional requirement specification. Functionality names were omitted for confidentiality purposes. Time values averaged were rounded up.

Functionalities	Expected Execution Time	Actual Execution Time
Single Table (F1)	10 to 50 ms	23 ms
Join with 2 Tables (F2)	100 to 1000 ms	1000 ms
Join with 3 Tables (F3)	500 to 1000 ms	3000 ms

Each system functionality was assigned an estimated execution time, based on system's non-functional requirements, as shown in Table 2. The system was deployed in production environment and measured without the optimized code. Those

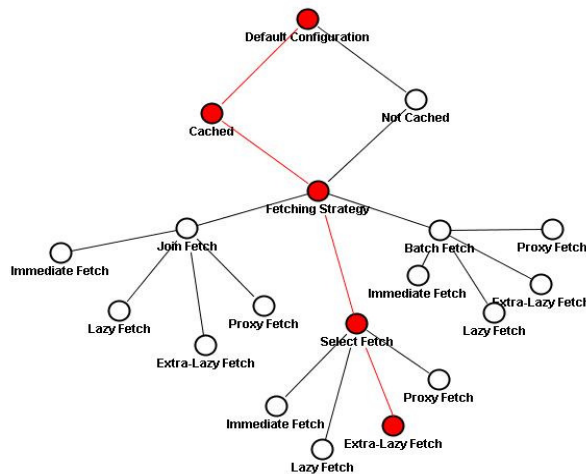
functionalities that had overcome the estimated value by 20% were identified as problematic.

After generating the code and deploying the system, the optimization process started searching for the best configuration possible. At this time, we were able to guarantee that the performance observed that functionalities F2 and F3 were the best that the configuration aspects could offer.

Values presented in table 2 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.

**Table 3.** Configuration process for functionality F3. The best path ended with the select fetch strategy and the extra-lazy fetch type, noticing that it was configured on a cache first perspective..

Cache Strategy	ON			
Fetch Strategy	Fetch Type	Time (in seconds)		Selection
Join Fetch		Average (Total)	(Hits)	
	Immediate	50 (250)	5	
	Lazy	16 (81)	5	
	Extra-Lazy	14 (70)	5	
	Proxy	12 (60)	5	
Batch Fetch				
	Immediate	6 (30)	5	
	Lazy	20 (100)	5	
	Extra-Lazy	20 (100)	5	
	Proxy	8 (40)	5	
Select Fetch				
	Immediate	6 (110)	21	
	Lazy	3 (30)	11	
	Extra-Lazy	3 (30)	12	X
	Proxy	5 (50)	12	



**Fig 7.** The best path for functionality F3 in the decision tree. Both Lazy and Extra-Lazy were candidates, but the number of hits (which means the number of database access) where lower for Lazy, therefore the overall average was higher. Values were rounded up by the profiling library.

## 6 Conclusions

This paper presented a new approach for using profiling techniques in MDA development. Our contribution aimed at 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 autonomic 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 although 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 and optimized in both scenarios.

## References

1. OMG, Model Driven Architecture, <http://www.omg.org/mda>, 2007.
2. AndroMDA, v3.0M3, <http://www.andromda.org/>, 2007.
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. OMG, UML 2.0 Testing Profile Specification. <http://www.omg.org/cgi-bin/doc?formal/05-07-07>, 2007.
5. Rodrigues, G. N., A Model Driven Approach for Software System Reliability, Proceedings of the 26<sup>th</sup> 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. OMG, Meta Object Facility, <http://www.omg.org/mof>, 2007.
8. Eclipse Test & Performance Tools Platform Project, <http://www.eclipse.org/tptp/>, 2007.
9. EJ-Technologies, JProfiler, <http://www.ej-technologies.com/products/jprofiler/>, 2007.
10. JAMon (Java Application Monitor), <http://jamonapi.sourceforge.net/>, 2007.
11. NetBeans Profiler, <http://profiler.netbeans.org/>, 2007.
12. Frankel D. S., Model Driven Architecture – Applying MDA to Enterprise Computing, OMG Press, Wiley Publications. 2003.
13. Hibernate, <http://www.hibernate.org>, 2007.
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/>, 2007
16. Maven project management and comprehension tool, <http://maven.apache.org>, 2007.
17. Eclipse Project, <http://www.eclipse.org>, 2007.
18. Velocity Project, <http://velocity.apache.org/>, 2007.
19. Struts Project, <http://struts.apache.org/>, 2007.
20. JBoss Application Server, <http://www.jboss.org/>, 2007.
21. Autonomic Computing – IBM's Perspective on the State of Information Technology. IBM, <http://www.ibm.com/research/autonomic/>, 2007.
22. Tivoli Software, IBM, <http://www-306.ibm.com/software/br/tivoli/>, 2007.
23. Centro de Catalogação das Forças Armadas - CECAFA, <http://www.defesa.gov.br/cecafa/>, 2007.
24. No Magic Inc., Magic Draw Case Tool, <http://www.magicdraw.com>, 2007.