

A Concern-Specific Metrics Collection Tool

Júlio Cesar Taveira

Department of Computing and
Systems, University of Pernambuco
Recife, Brazil
jcft2@dsc.upe.br

Juliana Saraiva

Department of Computing and
Systems, University of Pernambuco
Recife, Brazil
julianajags@dsc.upe.br

Fernando Castor

Informatics Center, Federal
University of Pernambuco
Recife, Brazil
castor@cin.ufpe.br

Sérgio Soares

Informatics Center, Federal University of Pernambuco
Recife, Brazil
scbs@cin.ufpe.br

Abstract

Static code metrics are widely used to evaluate quality attributes of software. Automated collection of these metrics is a must, which reduces errors inherent to manual collection, also improving the measurement process and guaranteeing the uniformity of the results, making it possible to compare them. Most metrics collection tools implement generic metrics that sometimes are inappropriate to evaluate quality attributes related to specific concerns of software systems, such as reuse. Furthermore, few of them are capable of measuring aspect-oriented programs. This paper presents an approach that uses a tool to collect metrics relative to a specific concern, Exception Handling. The proposed tool was used in a case study to collect metrics from various software systems written in Java and AspectJ. Feedback from these studies and comparison with other collection approach and tool are also presented.

1. Introduction

Source code metrics express information about software systems ranging between the basic, such as the number of lines of code and the sophisticated, e.g., dependencies between system elements and how focused these elements are [4, 8]. If these metrics are collected automatically, the measurement process is more efficient: (i) errors inherent to manual collection are avoided; (ii) the results of measurement

are uniform, which makes it possible to compare them; and (iii) the overall time and effort required to collect metrics are greatly reduced.

There are some tools that can collect static code metrics from programs written in different programming languages like Metrics¹ and FindBugs², which collect several metrics from Java programs, and Resource Standard Metrics³, which targets C, C++, C# and Java programs.

To the best of our knowledge, all the existing fully-automated metrics collection tools focus on generic metrics, that is, metrics that do not take into account the different concerns of the system under measurement. These tools do not collect concern-specific metrics. Consequently, it is not possible to evaluate some quality attributes that are strongly related to a specific concern, like Exception Handling.

Some researchers [3, 8, 7] believe that, to evaluate the impact of new development approaches, such as Aspect-Oriented Programming (AOP), considering quality attributes e.g., reuse, metrics focused on specific concerns are more relevant than metrics that are not. In fact, there is already evidence [17] that some attributes, such as code reuse within a system, are more precisely measured by using concern-specific metrics. It is important to evidence that this approach allow us a understanding in-depth of software attributes, such as reuse. In our study case specifically, we want to know if the exception handling code is reused in the system.

The use of generic metrics can lead to a wrong conclusion about the approach evaluation, where its benefits, limitations, and drawbacks will be inappropriately expressed. When a specific concern, mapped to specific parts of the

¹ <http://metrics.sourceforge.net/>

² <http://findbugs.sourceforge.net/>

³ <http://msquaredtechnologies.com/>

code, needs to be measured or evaluated, these generic metrics sometimes cannot capture the nuances and peculiarities of that concern in the same way that concern-specific metrics can.

It is not a simple task to adapt a tool that collects generic metrics to also collect concern-specific metrics. Currently, there is a tool available that counts the amount of spread code associated to selected concerns of the software system: the ConcernTagger tool⁴. In spite of its many virtues, ConcernTagger requires its user to manually select which parts (methods or class) of the code refer to the desired concern. It also imposes some limitations on the lines of code that its users can select, within a method. This prevents users from collecting some fine-grained metrics, such as CDLOC [10]. Another limitation of ConcernTagger is that it can only measure Java applications.

Despite the growing popularity of AOP [12], there are few tools that are able to collect metrics of aspect-oriented programs. The only one in widespread use is AopMetrics⁵, which collects metrics from both Java and AspectJ programs [13], but only implements generic metrics. AJATO⁶, on the other hand, similarly to ConcernTagger, allows users to manually mark parts of the source code of a program in order to collect concern-specific metrics.

To better convince the academy and the software industry about the benefits of AOP, exhaustive, in-depth evaluation, both qualitative and quantitative, is demanded. In this paper, we argue that a concern-specific, automated measurement approach should be employed to achieve more accurate results in empirical studies, as a complement to existing metrics collection tools. We propose a new measurement tool that collects metrics pertaining to a specific concern, exception handling, in both Java and AspectJ systems.

The proposed tool is called EH-Meter and is able to collect 11 source code metrics [8, 16] in a completely automated way. Ten of them deal with the exception handling concern. We have chosen to measure exception handling, instead of other concerns, due to a number of reasons: (i) the authors have vast experience in studying exception handling, AOP, and the interrelations among them (ii) exception handling code is syntactically demarcated, which allows automatic concern identification, avoiding some concern mining [15] difficulties, also allowing users to focus on the measurement activity; and (iii) many studies [2, 6, 11, 14, 17, 3] in the literature have employed exception handling as a case study to evaluate the benefits and drawbacks of aspect-oriented techniques and, as a consequence, further studies can benefit from the tool.

⁴ <http://www1.cs.columbia.edu/eaddy/concerntagger/>

⁵ <http://aopmetrics.tigris.org/>

⁶ <http://www.teccomm.les.inf.puc-rio.br/emagno/ajato/>

2. EH-Meter

EH-Meter trades off generality for automation, since it only collects exception handling metrics. We advocate a new approach to the design of metrics collection tools specifically aimed at evaluating aspect-oriented techniques. We believe these tools should collect metrics related to specific concerns, instead of only metrics that apply to the whole system. In this manner, it will be possible to obtain a more in-depth knowledge about the impact of AOP on different system concerns, further refining our knowledge on its advantages and limitations. EH-Meter can be used as a base, both conceptually and as an implementation framework, for the construction of such measurement tools.

An obstacle to this approach to the design of metrics collection tools is that concern-specific tools need to be capable of identifying in the code the crosscutting concern that they are attempting to measure. Identifying exception handling in most object-oriented languages is straightforward, since the concern code is textually delimited. In general, this is not an easy task [15]. However, we believe that, by developing new tools that focus on the syntactically identification of specific concerns in code, instead of attempting to detect every possible crosscutting concern [5], it is possible to achieve greater accuracy and completeness in concern identification. This suggests new paths for future research, following along the line of recent domain-specific approaches for AOP [1, 9].

Another important issue is the set of metrics to be collected. New metrics can be proposed, based on the concerns to be measured. However, in this paper, we have focused on well-known concern metrics that can be tailored to the specifics of each concern, as discussed in Section 2.1). Also, we employed size metrics that directly apply to the elements that are relevant to the exception handling concern: `catch` blocks, `try` blocks, number of lines of error handling code, among others. It is outside the scope of this paper to propose new metrics for more complex quality attributes, such as coupling, cohesion, and complexity.

This section presents the main features of EH-Meter. Section 2.1 describes the metrics that were implemented by the tool. Section 2.2 explains how the tool can be used. It also briefly describes how EH-Meter can be extended by developers implementing new metrics. Additional information, including the latest version of the tool, usage instructions, and its source code, is available.

2.1 Metrics Suite

EH-Meter is able to collect various metrics related to exception handling. The metrics can be fit into two distinct groups: Size Metrics and Concern Metrics. The first group comprises some metrics that determine the amount of code related to the exception handling concern. This means that, in this group, we have traditional size metrics, such as number of Lines of Code (LOC) and Lines of Exception Handling Code (LOCEH), and other less used metrics, such as

number of `catch` (NOC) and `finally` (NOF) blocks. In Table 1, the first six metrics are size metrics. A lower value is better for all the metrics of Table 1. For the NOT, NOC, and NOF metrics, the `try`, `catch`, and `finally` blocks are not counted if they appear within another `catch` or `finally` block. This decision was made because, in this case, such inner blocks are already part of the error handling concern. As a consequence, it would not make sense to consider them as part of a separate concern.

The metrics of the second group, concerns metrics, measure how well localized is the code related to a given concern. EH-Meter considers that the system has only two concerns: error handling and normal activity. The latter comprises all the remaining system concerns. The first three metrics [10] are well-known in the AO software development community: Concern Diffusion over Components (CDC), Concern Diffusion over Operations (CDO), and Concern Diffusion over LOC (CDLOC). They measure the diffusion of the exception handling concern in terms of how many components (classes and aspects) and operations (methods) implement the exception handling concern; and how many transition points exist between exception handling and normal activity. In addition, we use two more recent metrics [8] that were proposed in order to circumvent some of the limitations of the aforementioned concern metrics. The first one, Lack Of Focus (LOF), is an adaptation of the Degree Of Focus (DOF) metric. DOF measures the extent to which a component's lines of code are dedicated to the implementation of a given concern. Its value ranges between 0 (completely unfocused) and 1 (completely focused). A component's LOF is equal to $1 - \text{DOF}$. The last concern metric, Degree of scattering (DOS) measures the extent to which the LOC related to a concern are contained within a specific component.

It is important to stress that, to the best of our knowledge, EH-Meter is the first tool capable of collecting these concern metrics in an entirely automated manner. Moreover, it is the first tool capable of measuring Concern Diffusion over LOC [10] (CDLOC). On previous work that we and others carried out [2, 3, 10, 16], this metric was collected manually. Not even existing tools where concerns are identified manually, such as ConcernTagger and AJATO, implement this metric, possibly due to its complexity and the need to verify some complex transitions between source code lines.

2.2 Executing and Extending EH-Meter

The EH-Meter tool is written in Java and an extension to AOPMetrics. It is executed as an Apache Ant task. There is an XML configuration file that specifies which Java and AspectJ programs will be analyzed, which libraries should be used, what is the language version, and what will be the name of the output file. The tool is platform-independent because we can set these specifications in the configuration file and run in any platform supported by Ant. Due to its compatibility with Ant, EH-Meter can be used from within the Eclipse platform and other IDEs.

Figure 1 presents an Ant XML script that defines a task (`taskdef` - line 2) to execute EH-Meter. After its definition, the `ehmeter` task is used by the configuration script (line 7). This task requires the user to set the work directory (`workdir` - line 7) where EH-Meter stores temporary files and the source code (`fileset` - lines 10 and 11). Others attributes are optional. For example, `export` (line 8) indicates the format of the output of the tool, Excel or XML - and `resultsfile` specifies the name of the exported file (line 9).—More information can be found in EH-Meter website.

```

1 <target name="run">
2   <taskdef name="ehmeter"
3     classname="br.upe.dsc.ehmeter.EHMeterTask"
4     classpath="{dir.librariesEHM}/ehmeter-0.4.1.jar">
5     <classpath location="{dir.work}/${jar.name}"/>
6   </taskdef>
7   <ehmeter workdir="{dir.work}"
8     sourcelevel="1.5" export="xml"
9     resultsfile="{dir.work}/metrics-results2.xml">
10    <fileset dir="{dir.src}" includes="**/*.java"/>
11    <fileset dir="{dir.src}" includes="**/*.aj"/>
12    <classpath refid="classpath.common"/>
13  </ehmeter>
14 </target>

```

Figure 1. EH-Meter XML configuration file..

The result of executing EH-Meter can be a XML document or a Microsoft Excel spreadsheet. The produced output presents a list with the value of each one of the metrics described in Section 2.1 for each analyzed component (classes, interfaces and aspects). The only exception to this is the DOS metric. Since it depends on the complete system, EH-Meter only provides global values for it. We provide more information about the use of EH-Meter in Section 4.

As we mentioned before, EH-Meter is an extension of the AOPMetrics. Therefore, resources such as its Java and AspectJ parsers, and its metrics collection engine were reused to simplify the implementation of the tool. Extending EH-Meter with new metrics is very easy and only two modifications are necessary. First, it is necessary to add the new metric to the metrics listing of the tool. This listing is implemented by the `Metrics` class. The name of the class implementing the new metric must be included in the listing and assigned a unique identifier. Second, one must create a class responsible for collecting the new metric. This class must implement one of the two metrics calculation interfaces of EH-Meter (`MetricsCalculator` and `MetricsCalculatorGeneral`). Information about the program (e.g., its syntax tree) is easy to obtain within a class implementing a new metric. EH-Meter, similarly to AOPMetrics, employs AJDT's implementation of AspectJ syntax trees. EH-Meter supports the collection of both metrics whose global value consists of the sum of the measurements obtained from individual elements (components, operations, etc.), e.g., number of LOC and CDC, and metrics that have only a global value and require global information to be calculated, e.g., DOS. AOPMetrics does not support the latter approach.

Group	Metric	Description
Size Metrics	LOC	Number of lines of code in a program, excluding comments and blank lines.
	NOT	Number of <code>try</code> blocks.
	NOC	Number of <code>catch</code> blocks.
	NOF	Number of <code>finally</code> blocks.
	LOCEH	Number of lines of code related to the handling and capture of exceptions.
	NEH	Number of blocks of code that to contribute to handle exceptions. This includes methods called from <code>catch</code> blocks, when they are part of classes tagged with the <code>@ExceptionHandler</code> annotation.
Concern Metrics	CDC	Number of components (classes and aspects) that include some exception handling code and the number of components that access them.
	CDO	Number of operations (methods and advice) that include exception handling code and the number of operations that access them.
	CDLOC	Number of transition points for each concern through the lines of code. The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas.
	LOF	Variance of the dedication of a component to every concern. The result is normalized between 0 (completely focused) and 1 (completely unfocused).
	DOS	Variance of the concentration of a concern over all components. The result is normalized between 0 (completely localized) and 1 (completely delocalized, uniformly distributed).

Table 1. The employed metrics suite

For all the metrics implemented by default in AOPMetrics, metrics collection consists of measuring program units in isolation and then summing up the results in order to obtain the global value of the metric. For example, in a system with only two classes, A and B, the overall number of LOC is the sum of the number of LOC of A and the number of LOC of B. This approach works well for most of the metrics we are aware of. However, a few metrics require a global view of the system. For these metrics, simply measuring each component in isolation is not sufficient. One such metric is DOS. For this metric, we had to modify the AOPMetrics metrics collection engine to compute it for the entire system. With this modification, it is possible to extend EH-Meter in ways that are not possible without some extra work in AOPMetrics, to implement additional metrics that also require a global view of the system.

3. EH-Meter Internals

The architecture of EH-Meter is very simple and was separated into modules (packages) with well-defined purposes. The main module is responsible for receiving user input and executing the metrics collection engine. Another module includes parsers for Java and AspectJ programs. A third module is responsible for interfacing with AJDT, an AspectJ Eclipse plugin. Another module contains the classes implementing the metrics that the tool collects. Most of the changes applied to AOPMetrics consisted of including new classes, corresponding to new metrics, in this module. Finally, the last module is responsible for exporting the results of measurement to XML or Excel's XLS workbooks format.

New metrics were implemented by means of the Visitor pattern, as AOPMetrics, supported by Eclipse AST. Most of the time, these visitors start out by locating source code related to exception handling (`try`, `catch` and `finally`

blocks). Upon encountering error handling code, the visitor registers the information pertaining to the metric being collected. Results for size metrics were obtained by counting specific pieces of code and their implementation is straightforward. Interfaces were not considered for the purposes of calculating size or concern metrics. All the metrics that EH-Meter implements need to analyze the implementation of operations within classes and aspects and interface methods are always abstract. CDC and CDO calculation is also straightforward. For each component, EH-Meter just counts each operation that includes at least one statement pertaining to error handling. If one such statement exists, CDO is increased by one. If, when analyzing a given component, CDO is increased by at least one, CDC is also increased by one. To obtain the LOF (metric derived from DOF — Degree of Focus) and DOS, we just use information obtained from other metrics. For example, to compute DOS, the values of LOC and LOCEH for all the components must be accumulated throughout the measuring process. When all the system components have been measured, DOS is computed, as proposed by Eaddy et al. [8].

Amongst all the metrics that EH-Meter implements CD-LOC is the most complex and most difficult to collect. There is a wide range of different situations that must be considered when developing an algorithm to collect it.

Figure 2 shows a code snippet that illustrates the difficulty of measuring CDLOC. The grey LOC are part of the error handling concern. To calculate this metric we counted various cases of transitions between LOC referring to both different concerns and the same one. We considered a wide array of complicating factors, such as nested `try` blocks, lexically consecutive exception handlers, and different kinds of blocks, such as `if`, `for`, and `while` statements. We also tested the tool using a large number of examples, comparing

the results obtained by the tool with the results of a manual measurement.

```

public static void example() {
    System.out.println("Starting!");
    try{
        ...//Code that can throws a exception
    }catch (Exception e){
        ...//Exception Handler Code
    }
    System.out.println("Step 1!");
    If (true) {
        try{
            ...//Code that can throws a exception
        }catch (Exception e){
            ...//Exception Handler Code
        }
    }
    System.out.println("Step 2!");
    try
        try{
            ...//Code that can throws a exception
        }catch (Exception e3){
            ...//Exception Handler Code
        }
        }catch (Exception e2){
            ...//Exception Handler Code
        }
    System.out.println("Step 3!");
}

```

Figure 2. Examples of code transitions between concerns.

4. Evaluation

There are some tools that collect metrics to evaluate software quality attributes. A software metric relates individual measures in a meaningful way, so it is an indicator that provides an insight into the software project, process, or product [18] being assessed. Metrics express a quantitative evaluation to help us making strategic decisions about how to complete the common process framework activities, monitor progress during software development, and control product quality. In this context, automation is very important to accelerate the measurement process and provide consistent and uniform results. However, it is very difficult to automatically collect metrics to evaluate AOP techniques in an entirely automated way, at least using general (non concern-specific) metrics. EH-Meter sacrifices generality in order to achieve this goal for one specific concern: exception handling. In this section, we describe our current evaluation of EH-Meter. We have assessed the tool in terms of the time saved when using it, in comparison to other approaches for collecting concern metrics. EH-Meter was also employed to support an empirical study targeting three different versions of three medium-sized Java applications (for a total of nine versions). These results are described in the rest of this section.

Comparing Measurement Approaches. We made comparative tests to evaluate the benefits obtained from adopting a measurement approach based on EH-Meter. Initially, we checked the time spent to collect exception handling-specific metrics using three different measurement approaches. The first consisted of manually collecting the metrics. The second and third were partially and completely automatic, using ConcernTagger and EH-Meter, respectively.

Three versions of one system used in the case study were chosen to compare metrics collection approaches. The application was JHotDraw⁷, a Java GUI framework for technical and structured graphics. JHotDraw comprises 23k LOC and more than 400 classes and interfaces. This system also has 67 exception handlers. Here, we focused on Concern Metrics, specifically on CDC, CDO and CDLOC, described in Section 2.1).

Table 2 presents the obtained results. The columns present the adopted metrics collection approach and the rows present the refactored system versions. The numbers in the table indicate the number of minutes that each approach required. Original Code is the original version of the system. Ref. OO version refers to a version where exception handlers were extracted to Java classes, in order to remove duplicated handlers. Finally, the Ref. AO version uses aspects to modularize the exception handling. N/A stands for not available, since the ConcernTagger tool does not collect metrics from AspectJ programs.

	Manual	ConcernTagger	EH-Meter
Original Code	10 min	15 min	¡1 min
Ref. OO Version	10 min	15 min	¡1 min
Ref. AO Version	10 min	N/A	¡1 min

Table 2. Comparative metrics mode in minutes.

The results show that EH-Meter was more efficient than the other ways of collecting metrics. Manual metrics collection obtained results more expressive than when the ConcernTagger tool was used, but the manual collection is tiring and error-prone. The manual collection was better because was spent a lot of time to identify and relate (drag and drop) the parts of code (Methods and Classes) that refer to a specific concern in the ConcernTagger. Another important issue is that ConcernTagger cannot collect the CDLOC metric. It is important to stress that, after careful review, the results obtained by employing each approach were the same for the three metrics.

A Case Study. A case study was conducted to identify the effects of AOP on the reuse of exception handlers within an application (intra-application reuse). EH-Meter was used to measure all the metrics described in Table 1. We modularized exception handling in three software systems, each one with a Java and an AspectJ version. A total of 9 applications were used, 3 aspect-oriented (AspectJ) and 6 object-oriented (Java).

Manual collection of the metrics required 16 hours for 2 of the versions of these applications, while EH-Meter collected the same values in about 1 minute. The manual counting also presented some uniformity problems that did not occur with our tool. Finally, due to some conventions that were defined throughout the experiment, it was necessary to measure the various versions more than once. This procedure,

⁷ <http://www.jhotdraw.org/>

infeasible using manual metrics collection, was performed in a straightforward manner using EH-Meter.

The use of concern-specific metrics emphasized that AOP in fact promotes reuse of exception handlers. It also indicates that the overhead associated to aspectization is non-negligible, since the code associated to exception handling increased considerably when extracted from a non-trivial system, even though the number of protected regions (try blocks), handlers (catch blocks) and clean-up actions (finally blocks) have all decreased sensibly in the target systems of the study. Indeed, we believe that a better reuse of exception handlers will be improved if we can reduce the number of handlers. It is a consequence of reusing the catch/finally blocks with some refactorings. Also, the combination of these two findings complements previous studies [2, 14] about AOP and exception handling. These findings are strongly correlated to the approach that we propose in this paper. More information is available elsewhere [17].

5. Conclusion

This paper proposed a new approach to the construction of metrics collection tools to evaluate AOP techniques. It presented a tool that adheres to this approach and collects, in an automated way, metrics related to a specific concern: exception handling. We described the metrics implemented by the tool, how a developer can extend it, to use as a basis for other concern-specific metrics collection tools, and the evaluation we have performed up to now. In fact, in the future, we intend to apply the proposed approach to other concerns of interest that are syntactically identified, such as transactional execution [13].

EH-Meter is able to collect metrics from both Java and AspectJ programs. To the best of our knowledge, there are no any other tools that can automatically collect metrics pertaining to a specific concern. In addition, not even generic metrics collection tools implement the CDLOC metric. The tool is also a technical contribution and can be used by both researchers and practitioners to measure programs. EH-Meter, its source code, and additional information are available at the tool's website⁸.

Acknowledgments

We would like to thank the anonymous referees, who helped to improve this paper. Júlio Taveira is supported by CAPES. Sérgio Soares is partially supported by CNPq, grants 309234/2007-7 and 480489/2007-6. Fernando Castor is partially supported by CNPq, grants 308383/2008-7, 481147/2007-1, and 550895/2007-8. This work is partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

⁸ <http://www.dsc.upe.br/jcft2/eh-meter/>; <http://www.cin.ufpe.br/fjclff/eh-meter/>

References

- [1] Cacho, N., Castor Filho, F., Garcia, A., and Figueiredo, E. *Ejflow: taming exceptional control flows in aspect-oriented programming*. In Proceedings of AOSD'08, pages 72-83.
- [2] Castor Filho, F. et al. *Exceptions and aspects: The devil is in the details* 14th SIGSOFT FSE, pages 152-162, 2006
- [3] Castor, F. et al. *On the Modularization and Reuse of Exception Handling with Aspects*. Submitted to Software - Practice & Experience, 2009.
- [4] Ceccato, M. and Tonella P. *Measuring the Effects of Software Aspectization*. In: 1st Workshop on Aspect Reverse Engineering (WARE 2004), 2004.
- [5] Ceccato et al. *A Qualitative Comparison of Three Aspect Mining Techniques* In Proceedings of the 13th International Workshop on Program Comprehension, pages 13-22, 2005.
- [6] Coelho, R. et. al. *Assessing the impact of aspects on exception flows: An exploratory study*. Volume 5142 of Lecture Notes in Computer Science, pages 207-234.
- [7] Couto, C. F. M.; Faria, J.; Valente M. T. O. *Estimativa de Mtricas de Separao de Interesses em Processos de Refatorao para Extrao de Aspectos*. In: VI Workshop de Manuteno de Software Moderna, Ouro Preto. p.1-8, 2009.
- [8] Eaddy, M., Aho, A., Murphy, G. C. *Identifying, Assigning, and Quantifying Crosscutting Concerns*. In: Proceedings of the 1st ACoM Workshop, 2007.
- [9] Fabry, J., Tanter, E., and D'Hondt, T. *Kala: Kernel aspect language for advanced transactions*. Sci. Comput. Program., 71(3):165-180, 2008.
- [10] Garcia, A. et al. *Modularizing Design Patterns with Aspects: A Quantitative Study*. In: 4th AOSD, 2005
- [11] Hoffman, K.; Eugster, P. *Towards Reusable Components with Aspects: An Empirical Study on Modularity and Obliviousness* In Proceedings of the 30th ICSE.
- [12] Kickzales, G. ; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irwin, J. *Aspect-Oriented Programming*. In: 11th ECOOP, 1997.
- [13] Laddad, R. *AspectJ in Action*. 1st ed., Manning, 2003.
- [14] Lippert, M.;Lopes, C. *A study on exception detection and handling using aspect-oriented programming*. In: Proceedings of the 22nd ICSE, pages 418-427, 2000
- [15] Robillard, M. P.; Murphy, G. C. *Concern graphs: finding and describing concerns using structural program dependencies*. In: Proc. of the 24th ICSE, 2002.
- [16] Sant'Anna, C.; Garcia, A; Chavez, C.; Lucena, C; Staa, A. von *On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework*. In: XVII SBES, Manaus, Brazil, 2003.
- [17] Taveira, J. C. et al. *Assessing Intra-Application Exception Handling Reuse with Aspects*. To appear in Proc. of the 23rd Brazilian Symp. on Softw. Eng., 2009.
- [18] Pressman, R.S. *Software Engineering: A Practitioner's Approach* 2th ed., R.S. Pressman & Associates, Inc., 2001.