Towards a Framework for Guiding Aspect-Oriented Software Maintenance Empirical Studies

Marcelo Moura
Sérgio Soares [*]
Fernando Castor Filho [†]
Mário Monteiro
University of Pernambuco
{mlmm,sergio,fernando.castor,mqm}@dsc.upe.br

Phil Greenwood
Alessandro Garcia
Lancaster University
{greenwop,garciaa}@comp.lancs.ac.uk

Elliackin Figueiredo
Diego Araújo
University of Pernambuco
{emnf,daa}@dsc.upe.br

Abstract

Aspect-Oriented (AO) software development aims to improve software maintenance through the encapsulation of crosscutting concerns. However, empirical studies assessing the maintainability of AO software are still limited. Even worse, they are rarely replicated by different groups of researchers, thereby hampering the progress of the field. One of the key reasons for this is the lack of support for designing AO software exemplars or benchmarks to be reused across the community. This paper presents a framework for AO software maintenance, which has the purpose of guiding: (i) the planning, evaluation and replication of empirical studies, and (ii) the development, adaptation and evaluation of benchmark applications. The framework defines criteria to be satisfied by representative AO applications and their releases. We assess the effectiveness of the framework by comparing two different designs of the same empirical study: one that leverages the framework and another that was designed by an expert in empirical assessments of AO techniques.

1. Introduction

Aspect-Oriented Software Development (AOSD) is increasingly being applied to a wide range of application domains, such as Web-based systems [20], middleware [3], and software product lines [11]. AOSD aims to simplify software maintenance through the modularization of otherwise crosscutting concerns. However, the systematic evaluation of the benefits and drawbacks of AOSD in terms of maintainability is an often neglected, albeit critical, task. As AOSD is a relatively young paradigm, the preparation and replication of maintainability studies is challenging and time-consuming. In fact, empirical assessments are scarce in the context of aspect-oriented (AO) software maintenance [11; 13].

A fundamental stumbling block is the difficulty of enacting one of the first steps of any empirical assessment: the systematic selection, design, or adaptation of representative applications. The AO community in particular does not have a well-accepted group of exemplars or benchmarks [9; 19] for AO software maintenance. More fundamentally, researchers and practitioners do not have methodological frameworks to guide the construction of such benchmarks. Benchmark applications [25] are the basic empirical mechanisms to advance research and technology transfer in software engineering fields. In fact, there are some examples of their success within the software maintenance community [19].

Notwithstanding, the design of benchmarks for AO software maintenance is difficult for many reasons. First, the number of application domains where AOSD can be applied is rapidly increasing. AO programming was initially used to improve the maintainability of classical widely-scoped concerns in distributed applications [17; 20], such as concurrency control and error handling. Later, it has been used for very different purposes, such as improving middleware adaptability [3] and enhancing design stability of software product lines [11]. Second, the mechanisms that can be classified under the umbrella of AOSD and their hybrid incarnations in emerging programming languages [23] are consistently growing. Finally, such maintainability studies require the investigation of many factors typical in software maintenance tasks, such as different types of changes.

In spite of all the aforementioned challenges, there is a pressing need for methodologically supporting the generation of empirical studies and the construction of benchmark applications, rather than relying on a few "universal cases", which probably do not encompass several possible characteristics essential to different stakeholders goals and domains. In this context, this paper presents a framework that supports the assessment of AOSD techniques in terms of maintainability. The framework is an idealized scheme with specific guidelines and criteria to be realized by benchmark applications for assessing maintainability attributes of AO techniques. The framework guides researchers and practitioners in selecting or adapting applications and their releases that best fit the specific experimental goals. It can also be used to support the design, replication, and evaluation of empirical studies. Sec-

[*] Partially supported by CNPq, grants 309234/2007-7 and 480489/2007-6.
[†] Partially supported by CNPq, grants 481147/2007-1 and 550895/2007-8.

tion 2 introduces the proposed framework. Its effectiveness is assessed in Section 3. In Section 4 we reflect upon the many potential framework applications and discuss related work. Finally, Section 5 presents some concluding remarks.

2. The framework

This section presents a framework for AO software maintenance studies, which supports the systematic: (i) elaboration, evaluation and replication of empirical studies, and (ii) selection, generation and adaptation of benchmark applications. Based on the framework guidelines, their stakeholders can determine the extent to which applications, and its maintenance scenarios, are effective to cover study goals and assess AO techniques. It defines guidelines that can be tailored or extended to: (i) fit the specific goals of a maintenance study, and (ii) stimulate the evaluation of AO composition mechanisms. These guidelines are based on our experience from conducting a family of AO software maintenance studies over the last few years [3; 11; 13; 12], as well as analyzing others studies [1; 10].

The framework is structured according to three major components: Process (Section 2.1), Product (Section 2.2), and Maintenance Scenarios (Section 2.3). The first one is related to the framework workflow and how it works, and the other two address complementary assessment issues relevant to AO software maintenance. A concrete example on the application of the framework components is presented in Section 3.

2.1 The process component

This sections details the process and workflow related to the framework usage. The framework can be analyzed as a process to transform inputs into outputs, which may also serve as feedback to improve the framework. First, we classify framework *stakeholders* into two categories: the *designer of empirical studies* and the *benchmark designer*. The first group is interested in conducting a maintainability study involving one or more AO techniques. In this case, the input is the set of experimental requirements and the output is the initial configuration of the experiment. In contrast, the second group includes people who change or add new artifacts to the benchmark application. This group needs to analyze applications and maintenance scenarios to determine if they are suitable to conduct a variety of studies. The output of this process is one or more applications and change scenarios that are appropriate to benchmark AO techniques. Figure 1 shows a schematic representation of the Process component with the framework stakeholders, inputs and outputs.

2.2 The product component

This section lists several characteristics that can guide and support the various decisions that have to be made, pertaining to the product (target systems), when designing maintainability studies and selecting candidates benchmark applications. We consider a wide range of possible characteristics that a

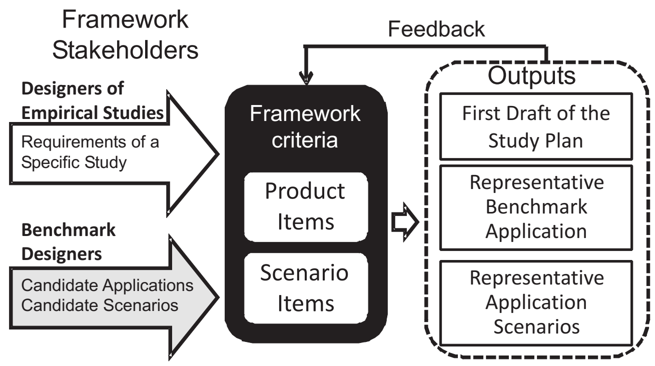
Framework Stakeholders
Feedback
Designers of Empirical Studies
Requirements of a Specific Study
Benchmark Designers
Candidate Applications
Candidate Scenarios
Framework criteria
Product Items
Scenario Items
Outputs
First Draft of the Study Plan
Representative Benchmark Application
Representative Application Scenarios

Figure 1. The inputs and outputs of framework process.

candidate application might exhibit, such as its domain, the development techniques that were employed in its construction, and the types of the crosscutting concerns that appear in its implementation. It is important to stress that it is impossible to define a set of characteristics that applies to every possible empirical study or application domain. Therefore, the proposed list of application characteristics should be constantly evolved and extended to more closely meet the stakeholder's goals. Although, each stakeholder does not need to consider all criteria elements, they should focus on a representative subset that cover their experimental objectives. To ease this identification process and improve the organization of the different criteria, the Product component is divided into two groups: General Attributes, which encompass common application characteristics, and AO Attributes, which consist of specific characteristics of AO applications.

2.2.1 General attributes

A variety of general application attributes must be considered by framework users, such as *System Domain*, *Versions Available*, and *Packaging*. The domain of a system is often an important aspect to consider, so as ensuring that the system exhibits some expected properties (e.g. embedded systems are often of a resource constrained nature). Another important information, in order to evaluate the so mentioned and not so much demonstrated AO improvements, is the availability of OO versions of the same application, which is described in the *Versions Available* attributes. Related to the System Domain and the Version Available, the *Packaging* attribute specifies the technologies and programming languages used in an application, in addition to target platforms/operating systems. When considering an application for use in an empirical study it is important to consider the development artifacts (e.g. requirements specification, architecture documentation, and the design diagrams) available to ensure the goals of the study can be realized. The software *Life-Cycle Documentation* attribute lists all the documentation artifacts available for assessment. In addition to listing the documentation, the *Development Techniques* that have been used to create the system (e.g. design patterns, application toolkits, etc.) should also be described. Different techniques can influence the quality of the final product and it is therefore important to take this into account when selecting an application.

2.2.2 Aspect-Oriented attributes

A variety of previous work has attempted to classify the different types of crosscutting concerns (CCCs) [1; 7]. The *Crosscutting Concern Classification* attribute provides classification of CCC types that are implemented with aspects according to different dimensions. For instance, the most popular classification of this nature has identified heterogeneous and homogeneous CCCs and also attempted to classify the invasiveness of the concern (i.e. if it alters the control flow of the system). For AO-related empirical studies it is important to have a range of different types of CCC implemented as aspects to guarantee that a variety of AO language constructs are exercised. Our CCC classification is based on a representative subset of the aforementioned existing classifications.

The first dimension of our CCC classification categorizes the concern as being *Functional* or *Non-functional*. A functional concern relates to business functionality, whereas a non-functional concern relates to the quality of the services provided by the system (e.g. security, reliability, distribution, etc.). Both functional and non-functional concerns can further be classified as either being *Homogeneous* or *Heterogeneous*. A homogeneous concern extends a program at multiple joinpoints by adding the same code at each joinpoint. A heterogeneous concern again extends multiple joinpoints but with a unique piece of code at each joinpoint. The final dimension of the CCC classification identifies if a concern affects a single component or multiple components, by either being an *Intra-Component* or *Inter-Component* concern.

Concerns in a system may be composed in a variety of ways. These compositions cause interactions between concerns which may affect system maintainability. Therefore, it is important to understand and classify these different types of composition. The *Concern Compositions* attribute allows the concern composition to be defined in some ways [3]. The simplest form of composition is *Invocation-Based Composition*. This arises if two concerns, C1 and C2, have no classes or aspects in common, i.e., they only communicate via method calls. *Component-Level Interlacing* occurs if multiple concerns have one or more components (classes or aspects) in common but no operation (i.e. method, advice, etc.) in common. As a result, the concerns are interlaced and tangled at the component level only. In contrast, *Operation-Level Interlacing* occurs if multiple concerns have one or more operations in common. In this case, concerns are interlaced at the operation level. Finally, *Overlapping* identifies points where multiple concerns share one or more statements, operations or components. In contrast to interlacing interactions that have disjoint common parts, overlapping concerns share elements (i.e. an entire component operation or statement contributes to multiple concerns).

A previous study reported that the maintainability of the code pertaining to a concern is directly related to the language constructs employed in its implementation [13]. Therefore, it is important to identify the various language features used to implement a particular concern. The *AO Language Constructs* attribute identifies the language elements used to implement a crosscutting concern, such as intertype declarations, pointcuts (the various different kinds of), advices, etc. The rationale in this case is that an application that leverages a large number of language constructs may be a good benchmark application. Language constructs can also be classified into other dimensions, such as, static (e.g. intertype declarations) or dynamic (e.g. cflow pointcut designator), according to the time when they are handled by the compiler (compile-time, load-time, run-time). This information could help the designers to evaluate in another way the performing of constructs at applications.

2.3 The maintenance scenarios component

A framework for empirical studies in software maintenance must also include a catalog of representative real software changes scenarios. This is necessary to ensure the framework can evaluate a variety of recurring maintenance issues. These scenarios are assessed through the attributes presented in the rest of this section, which aims to support decisions of both benchmark designers and empirical studies designers.

The *Scenario Description* attribute provides the name, description, and other general information about the scenario. Another provided information is the *Versions Available* attribute that reports if there are OO versions of each scenario. It is also necessary to specify the *Change Type*, which is partially based on previous works [2; 4; 24]. This involves classifying the change according to the effect it has on the base application. In this work, we consider that changes can be of one amongst three types: corrective, adaptive, or perfective [21]. This attribute also requires the *Goal of the Change* to be specified, which describes the desired effect the change has on the system.

The *Nature of the Change* must also be listed and it encompasses two types: structural and behavioral changes. These change types are orthogonal; therefore, a change scenario can affect both the software structure and its behavior at the same time. When *Structural Changes* are made, these can involve: *additions* (e.g. introduction of a new method), *subtractions* (deletion of an attribute), or *alterations* (to modify an existing element). In contrast, *Behavioral Changes* are changes that modify the software behavior and can be classified as either: (i) *behavior-modifying*, changes that alter the software behavior; or (ii) *behavior-preserving*, changes that preserve the behavior of the software (refactorings).

Finally, it is necessary to actually document the changes that are made. This involves specifying the *Changes at the Requirements Level*, *Changes at the Analysis and Design Level*, and *Changes to the Implementation* attributes. The changes performed in the requirements can influence artifacts at later development stages. These effects can differ depending on the applied requirements engineering approaches [21]. Almost all changes made to the analysis and design artifacts are critical due to these artifacts being the core of software development activities. Analysis and design changes are likely to affect the rest of the application and can, consequently, impact software modularity. Finally, changes to implementation artifacts often have less broad-scoped effects.

3. Framework evaluation

This section presents an evaluation of the framework involving a real application, the HealthWatcher system [20]. The goal is to assess the efficacy of the proposed framework when planning a new empirical study in AO software maintainability. Our analysis aimed at examining (i) whether circumstances the framework can assist in identifying omissions and inaccuracies in the planning or execution of empirical studies, and (ii) whether characteristics of the target applications adhere to the framework attributes.

We have selected the aforementioned application for several reasons. First and foremost, it needs to be examined for its exploitation in different empirical contexts. HealthWatcher was being considered by another research group as a target candidate in the planning of an upcoming controlled experiment. The goal of the new experiment was to assess the maintainability of AO Web-based systems. The framework was used in this study in order to guide the elaboration of the experimental design, one of the framework outputs (Figure 1). Second, the application has a number of unique characteristics that make it an appropriate representative of evolving AO software systems. It was developed using different programming languages, such as Java and AspectJ [17]. There are already some recent studies in the research community using it (e.g. [6; 13; 20]). In fact, the system was specially adapted to work as benchmarks for comparing the maintainability of AO and OO techniques [20]. Furthermore, there are multiple releases of HealthWatcher available and each release is the result of applying a number of heterogeneous changes to the previous one. Third, HealthWatcher is a Web-based information system based on a classical n-tier architecture. It is a real system developed by a team of developers in Recife, Brazil [20]. Finally, the system and its releases were originally designed without any access to the framework proposed here.

3.1 Planing a new experiment

As previously mentioned, this initial evaluation of the framework was carried out in the context of planning a new experiment. Considering one of the previous studies that generated several releases and scenarios of HealthWatcher, which focused on analyzing the stability of AO software design in the presence of heterogeneous types of changes [13] *(Exp. A)*. Designers of a new controlled experiment to evaluate the maintainability of AO Web-based systems *(Exp. B)*, performed an informal analysis to verify if the application release and the maintenance scenarios generated in *Exp. A* could be reused in the context of their new experiment. Thus, the framework was used to support the same analysis that occurred in *Exp. B*, guiding experiment designers to analyze and configure, if necessary, HealthWatcher and its scenarios in the light of the key experimental goals. We analyzed the efficacy of the framework by comparing two different drafts of the experimental plan that were devised independently, as described below.

Experimental Settings: Expert vs. Framework User. The first experiment plan was designed using the framework. The other one was designed by an expert that did not rely on the framework. The goal of this evaluation was twofold. First, we expected to be able to systematically verify how the framework can assist the planning of studies focusing on AO software maintenance. Moreover, by comparing the results of using the framework with the study design produced by an expert, we also expected to identify benefits and limitations of the framework. Both drafts of the experiment plan were generated based on the same set of goals. The design of the first plan assisted by the framework was carried out by a postgraduate student who was not expert in architectural analysis but had basic knowledge on experimental design. The other (without using the framework) was prepared by a researcher who was an expert in architectural analysis, had developed a number of empirical studies of AOSD in the last five years, but had no familiarity with the framework.

3.2 Evaluation results

Based on the specific experimental goals, we have examined if the framework criteria were effective in the analysis and adaptation of HealthWatcher. We have centered the assessment on the product and maintenance scenario components. After the execution of this evaluation, we have identified if there was any product element or change scenario that was not present in the framework. This provided a realization of the feedback loop illustrated in Figure 1. The results provided first evidence that the use of the framework might be appealing even for experts on empirical assessment of AOSD. As described in the following, the first draft, generated by the postgraduate student, encompassed some relevant considerations that were not addressed by the expert.

Scenarios for Observing AO Architecture Instabilities. While analyzing the appropriateness of HealthWatcher, the postgraduate student has considered possible architectural instabilities when different characteristics of AO software are present. In the expert plan, the only product characteristics taken into consideration were the diversity of functional and non-functional concerns. This was very limited when compared to the ones defined in Section 2.2. Table 1 shows all occurrences of the items described in Section 2.3.2. Notice that there is no functional crosscutting concern. This gap is a useful information for decision making in the experimental design. If no additional change scenario is included in the application, there is no guarantee that the experiment results will be representative to Web-based applications that do encompass such a characteristic (for example, to make the implementation of business rules more flexible).

Unified Classification of Maintenance Scenarios. Another point in our analysis of the two experiment plans was related

Functional	Non-Functional	Homogeneous	Heterogeneous	Intra-Component	Inter-Component	Invocation-based composition	Component-level interlacing	Operation-level interlacing	Overlapping
-	X	X	X	X	X	X	X	X	X

Table 1. Types of CCC and their Interactions.

to the classification criteria for the maintenance scenarios. We have observed that the expert (Case 2 in Table 2) categorized the changes using his own terminology, differently from the postgraduate student (Case 1), which used the framework categories. The expert included the use of terms for maintenance types that are not widely accepted yet by well-known classifications of software changes [21; 24]. In fact, there is no consensus on the categorization of software maintenance types [21] that are used in practice. This imprecision makes the existence of a framework fundamental to avoid misunderstandings when analyzing and replicating studies on AO software maintenance. Such ambiguity problems can be ameliorated through the use of the framework. It guides different experiment designers to analyze characteristics of their experiments in a coherent way. On the other hand, the draft generated by the expert covered some essential points not covered by the postgraduate student. The expert was more effective to anticipate specific architectural stability metrics to be applied to the change scenarios.

Release	Scenario Description	Case 1	Case 2
R1	Factor out multiple Servlets to improve extensibility.	P	R
R2	Ensure the complaint state cannot be updated once closed.	P	C
R3	Encapsulate update operations to improve maintainability.	P	R
R4	Improve the encapsulation of the distribution concern.	P	R
R5	Generalize the persistence	P	R
R6	Remove dependencies on Servlet response and request.	P	R
R7	Generalize distribution mechanism.	P	R
R8	New functionality added to support querying of more data types.	P	E
R9	Modularize exception handling and include more effective error recovery behavior into handlers.	P	P

Legend: P = Perfective; R = Refactoring;
C = Corrective; E = Evolutionary.

Table 2. HealthWatcher maintenance scenarios [13].

4. Discussion and related work

Based on the previously presented results, this section discusses the benefits and limitations of our framework and contrasts it with related work.

On the Symbiosis between Framework and Designers Experience. The framework is not a replacement for the creativity and experience of empirical studies and benchmarks designers. On the contrary, experts can be more effective to include certain change types (e.g. refactorings) that are not explicitly cataloged in popular classifications of software changes. Also, the strict use of the framework might constrain creativity in experimental designs if not applied properly. A good practice might be using the framework only after delivering a first plan draft of the empirical study.

Framework Coverage. It is virtually impossible to come up with a framework that covers all the relevant issues of benchmarking applications for AO software maintenance. Different maintainability studies of AOSD have specific goals that impose particular demands on the used applications. However, the idea is that the framework can assist on the time-effective generation of a first sketch on the experiment plan. As we have noticed in the HealthWatcher case, there are certain relevant benchmarking items that are easily skipped without using the framework. Besides, the framework provides a coherent and unified way of documenting important characteristics necessary to ease replication of empirical studies. The framework criteria can also be extended.

Absence of Benchmarking Frameworks for AOSD. A variety of related research has influenced the development of our framework as documented throughout the previous sections. However, there are very few instances which attempt to consolidate a methodology for designing benchmarks for AO software maintenance. Existing benchmarking frameworks in software engineering have been limited to provide decision support tools for creating such processes [5; 8]. Alternatively, the work presented by Demeyer [9] describes a software evolution benchmark that has very similar goals to ours. However, his framework is very generic and does not include important collaboration attributes.

In contrast, the work by Kienzle [18] derives a set of attributes for assessing language expressiveness based on aspects for ACID properties. The authors claim that these aspects and attributes naturally create a benchmark for assessing AO languages. However, a problem of this benchmark is that it is focused on a particular domain (transactional properties), which limits its applicability. Our proposed approach for benchmarking not only takes maintenance activities into account but is applicable to a variety of domains and does not focus on one particular assessment attribute. This categorization of aspects used by our framework and Kienzle is reminiscent of Aspect Categories [16], which attempts to classify aspects according to the affect they have on the base system. Some of these categories overlap with the attributes specified in our framework (e.g. concern compositions). However, the work by Katz is focused on proving correctness, rather than supporting assessment.

Improving the Body of Knowledge on AO Software Maintenance. Shull et al [22] highlight the need for information sharing and replication in empirical studies. The benchmark framework described in this paper contributes significantly towards these goals. Furthermore, from the work described in this section it is clear there is a need to document a set of guidelines for creating empirical studies of AOSD. Although a number of such studies have been performed, they are often conducted in isolation using different practices. By providing a framework, studies will promote easier information exchange and it will encourage more AOSD practitioners to conduct and replicate maintainability studies.

Interplay of Testbeds and Frameworks for AOSD. One of the most closely related initiatives to our framework is [14]. This initiative consists of developing an AOSD testbed with a long-term aim of providing a series of benchmark applications (and supporting artifacts). The testbed repository also contains a set of initial metrics and collected results with intention of other software developers reusing these resources to test their approaches. Our framework directly complements this initiative by i) providing a decision support framework for selecting and creating artifacts to instantiate appropriate empirical studies and ii) promote information sharing to allow the replication of studies.

5. Conclusions and Ongoing Work

The proposed framework constitutes a significant stepping stone towards the creation of a compressible methodology for designing AO software benchmarks. Successful technology transfer usually takes a long time to be implemented, often around 18 years [15]. Supported by our first evaluation outcomes, we believe that our framework is an effective contribution to decrease this delay in many ways. First, it has shown to provide systematic ways for facilitating the effective design of maintainability studies on the AOSD arena. Moreover, the framework can also guide the selection, design, or adaptation of representative evolving applications and releases to be used in such studies and their replications. Third, this will help the community to accelerate the improvement of our body of knowledge on AOSD and better support the judgment of industrial decision makers. Finally, studies extensions and replications are time-consuming; in order to ameliorate this problem, we believe that our framework helps to save time on the quality assessment of existing empirical studies. However, the framework is not intended to guide all empirical decision that have to be done when planning, replicating, or evaluating studies. For example, the empirical study designers must evaluate if the relation of the selected set of scenarios with each other can bias the results. As ongoing work, we are planning to conduct a number of applications of the framework.

References

[1] Apel, S., et al. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?. AOPLE at GPCE.06, 2006.

[2] Buckley, J., et al. Towards a Taxonomy of Software Change. Journal on Software Maintenance and Evolution: Research and Practice, p. 309-332, 2005.

[3] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. AOSD.06, 2006.

[4] Chapin, N., et al. Types of software evolution and software maintenance. Journal on Software Maintenance and Evolution: Research and Practice, 13:330, 2001.

[5] Chiew, V., et al. Software Engineering Process Benchmarking. PROFES.02, p. 519-531. LNCS, v. 2559, 2002.

[6] Chitchyan, R., et al. Early Aspects at ICSE 2007. ICSE.07 Companion, p.127-128. ACM Press, May 2007.

[7] Constantinides C., et al. Reasoning About a Classification of Crosscutting Concerns in Object-Oriented Systems. Workshop on AOSD. Bonn, February 2002.

[8] Daoudi, F., et al. A Benchmarking Framework for Methods to Design Flexible Business Processes. Software Process: Improvement and Practice, 12(1):51-63, 2006.

[9] Demeyer, S., et al. Towards a Software Evolution Benchmark. Workshop on Principles of Software Evolution, 2001.

[10] Eaddy, M., et al. Do Crosscutting Concerns Cause Defects? IEEE Transaction on Software Engineering, 2008 (to appear).

[11] Figueiredo, E., et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. ICSE.08, p. 261-270, May 2008.

[12] Garcia, A., et al. Modularizing Design Patterns with Aspects: A Quantitative Study. AOSD.05, March, 2005.

[13] Greenwood, P., et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP.07.

[14] Greenwood, P., et al. On the Contributions of an End-to-End AOSD Testbed. Early Aspects at ICSE.07, May 2007.

[15] Jedlitschka A., et al. Relevant Information Sources for Successful Technology Transfer: A Survey Using Inspections as an Example. ESEM 2007, p. 31-40. IEEE, September 2007.

[16] Katz, S. Aspect categories and classes of temporal properties. TAOSD, 3880, Springer, 2006.

[17] Kiczales, G., et al. Getting Started with AspectJ. Communications of the ACM 44(10), 5965 (2001).

[18] Kienzle, J., et al. AO challenge - implementing the ACID properties for transactional objects. AOSD.06, March 2006).

[19] Sim, S., et al. Using benchmarking to advance research: a challenge to software engineering. ICSE.03, IEEE, 2003.

[20] Soares, S., et al. Implementing Distribution and Persistence Aspects with AspectJ. OOPSLA.02, p. 174-190, 2002.

[21] Sommerville, I. Software Engineering. Addison-Wesley, 2006.

[22] Shull, F., et al. Knowledge-Sharing Issues in Experimental Software Engineering. Empirical Software Engineering, 9:111-137, 2004.

[23] Sullivan, K. et al. Information Hiding Interfaces for Aspect-Oriented Design. FSE-13, Sept 2005, Lisbon, Portugal.

[24] Swanson, E. The Dimensions of Software Maintenance. ICSE.76, p. 492-297. IEEE, 1976.

[25] Tichy, W. Should Computer Scientists Experiment More? IEEE Computer, 31(5):3240, May 1998.