A systematic review of comparative evidence of aspect-oriented programming

Muhammad Sarmad Ali, Muhammad Ali Babar, Lianping Chen, Klaas-Jan Stol

Publication date

01-01-2010

Published in

Information and Software Technology;52/ 9/ pp. 871–887

A Systematic Review of Comparative Evidence of Aspect-Oriented Programming

Muhammad Sarmad Ali[1], Muhammad Ali Babar[2], Lianping Chen[1], Klaas-Jan Stol[1]
[1]Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland
[2]IT University of Copenhagen, Denmark
[1]{sarmad.ali, lianping.chen, klaas-jan.stol}@lero.ie, [2]malibaba@itu.dk

Abstract

Context: Aspect-Oriented Programming (AOP) promises to improve many facets of software quality by providing better modularization and separation of concerns, which may have system wide affect. There have been numerous claims in favour and against AOP compared with traditional programming languages such as Objective Oriented and Structured Programming Languages. However, there has been no attempt to systematically review and report the available evidence in the literature to support the claims made in favor or against AOP compared with non-AOP approaches.
Objective: This research aimed to systematically identify, analyze, and report the evidence published in the literature to support the claims made in favor or against AOP compared with non-AOP approaches.
Method: We performed a systematic literature review of empirical studies of AOP based development, published in major software engineering journals and conference proceedings.
Results: Our search strategy identified 3,307 papers, of which 22 were identified as reporting empirical studies comparing AOP with non-AOP approaches. Based on the analysis of the data extracted from those 22 papers, our findings show that for performance, code size, modularity, and evolution related characteristics, a majority of the studies reported positive effects, a few studies reported insignificant effects, and no study reported negative effects; however, for cognition and language mechanism, negative effects were reported.
Conclusion: AOP is likely to have positive effect on performance, code size, modularity, and evolution. However its effect on cognition and language mechanism is less likely to be positive. Care should be taken using AOP outside the context in which it has been validated.

Keywords

Evidence-based software engineering, systematic literature review, aspect-oriented programming

1. Introduction

It has been more than a decade since the Aspect-Oriented Programming (AOP) paradigm was introduced by Kiczales et al. [1]. AOP was presented as an alternative approach to Object Oriented Programming (OOP) for better modularization and Separation of Concerns (SoC), especially for those concerns that cut across a system's functionality and hence, can result in redundant, scattered and tangled code. This relatively new paradigm has attracted a lot of interest from researchers and practitioners recently. A wide variety of AOP languages and tools have been developed. It has been argued that AOP and its related techniques can have a positive impact on the overall software development process and improve software quality [2].

Such claims are usually based on Dijkstra's idea that the more the concerns are separated, the easier it becomes to perform changes locally [3]. However, there have also been doubts about the applicability and effectiveness of AOP [4]. As compared to OOP, the most popular software development paradigm today, aspect-oriented paradigm can be considered to be in its infancy as it lacks well-defined rules to determine good design and implementation decisions [5]. There are also claims about the limitations of AOP techniques [6]. Furthermore, it has been reported that not many empirical studies have been conducted to investigate the effectiveness of AOP, which is why there is little empirical evidence available to support the claims made about AOP [7].

However, there has been no effort to systematically identify, analyze, and report the evidence reported in the literature to support the claims made in favor or against AOP. We believe that systematically carried out aggregation and synthesis of the reported evidence can help clarify the confusions and contradictions regarding AOP's benefits and limitations. Hence, we decided to conduct a Systematic Literature Review (SLR) of the evidence reported about the benefits and limitations of AOP compared with non-AOP. This paper reports the methodological details about and findings from our SLR.

1.1 Contribution of this Review

This review provides evidence-based insights that can help practitioners to gain a good understanding of the claimed benefits of AOP and the kinds of evidence provided to support those claims. We also believe that readers interested in the AOP paradigm can use this paper as a map for finding studies relevant to their situation and then analyze the study settings to decide about their applicability. For researchers, this SLR gives an overview of the reported empirical evaluation/ validation of the effectiveness of AOP and reveals those areas that are not addressed by the reported research or areas that need further research. At the same time, it also points out the limitations of the current practice of designing and reporting empirical studies of AOP. The information extraction scheme we used to characterize the study context and study findings can be used to guide the activities of designing and reporting future empirical studies of AOP.

The remainder of this paper is organized as follows. Section 2 provides a short introduction to AOP. Section 3 reports the details of our research methodology and logistics. Section 4 provides an overview of the selected studies. Main findings from our SLR are presented in Section 5. Section 6 reports the limitations of the review. Finally, Section 7 concludes this paper.

2. Background: Aspect Oriented Programming (AOP)

At the heart of aspect oriented development paradigm is the idea of *concern*. Though an abstract concept, a concern at implementation level is usually considered as a particular behavior or functionality in a program. Concerns can be very primitive, such as adding a variable. High level concerns are coarser, such as transaction management. Of particular interest in AOP is a concern whose implementation is *scattered* over various system modules, or when a particular module's implementation is *tangled* with different concerns. Scattering and tangling usually go hand in hand and result into what is termed as *crosscutting* of concerns. Crosscutting concerns are *homogeneous* when the same or almost similar behavior is replicated at multiple points in the implementation, such as logging or tracing. When the behavior at multiple points is different, the crosscutting concern is termed *heterogeneous*. With other programming paradigms (e.g., object oriented), even if a developer chooses a system's structure carefully, the implementation of such concerns may still end up being non-modular i.e., scattered and tangled across multiple modules. The proponents of AOP claim that AOP provides mechanisms to modularize and encapsulate crosscutting concerns which appear due to "*dominant decomposition*"[1] [8].

The core idea of AOP, separation of concerns, has been around for many years with different names. Earlier approaches such as adaptive programming [9], subject-oriented programming [10] and composition filters [11] shared the same idea. However, the model of AOP (as implemented in the AspectJ programming language) proposed by Kiczales et al. [12] proved to be a simpler extension to the popular OOP language Java. It is now well supported by the Eclipse project[2], and many different plugins have been developed. The remainder of this section briefly discusses different features of AOP in languages such as AspectJ.

In AOP languages, crosscutting concerns are encapsulated in an *aspect*, a class-like construct. This encapsulation is sometimes colloquially termed as *aspectization*. A single aspect can contribute to the behavior of a number of methods or objects through implicit invocation of additional behavior, which is composed at specific points of interest in the execution of a program. These points of interest are called *join points*. A *pointcut* is a language construct which defines a join point in the code. The additional behavior can execute *before*, *after* or *around* join point, and is defined in an *advice*. Programs written with aspects can be composed and compiled with the base code. Aspect-oriented code is either transformed into the base code language where it becomes indistinguishable for the interpreter, or modifies the interpreter/environment to understand aspect-oriented code. Since it is difficult to change a programming's runtime environment, a special program transformation process called *weaving* is used. The *weaving* converts aspect-oriented code into object-oriented code with the aspects integrated into the code.

3. Research Method

We conducted an SLR, which is a well-defined and rigorous method to identify, evaluate and interpret all relevant studies regarding a particular research question, topic area or phenomenon of interest [13]. The goal of an SLR is to give a fair, credible and unbiased evaluation of a research topic using a trustworthy, rigorous and auditable method. A common reason for undertaking an SLR is to summarize existing evidence concerning a technology [13]. Hence, an SLR was an appropriate research method for our research that aimed at identifying and evaluating the evidence regarding the benefits and limitations of the AOP paradigm. For our SLR, we followed the guidelines for performing SLRs as proposed by Kitchenham and Charters [13]. The remainder of Section 3 discusses our approach in more detail.

3.1 Development of review protocol

Prior to conducting our systematic review, we developed a review protocol. A pre-defined protocol reduces researcher bias and increases the rigor and repeatability of

[1] A good definition of "*dominant decomposition" can be found on this site: http://aosd.net/wiki/index.php.*

[2] http://www.eclipse.org/aspectj/

the review. An SLR protocol specifies the review plan and procedures by describing the details of various strategies for performing the systematic review. In particular, it defines the research questions, search strategy to identify the relevant literature, inclusion and exclusion criteria for selecting relevant studies, and the methodology for extracting and synthesizing information in order to address the research questions. The protocol was developed following the process shown in Figure 1.

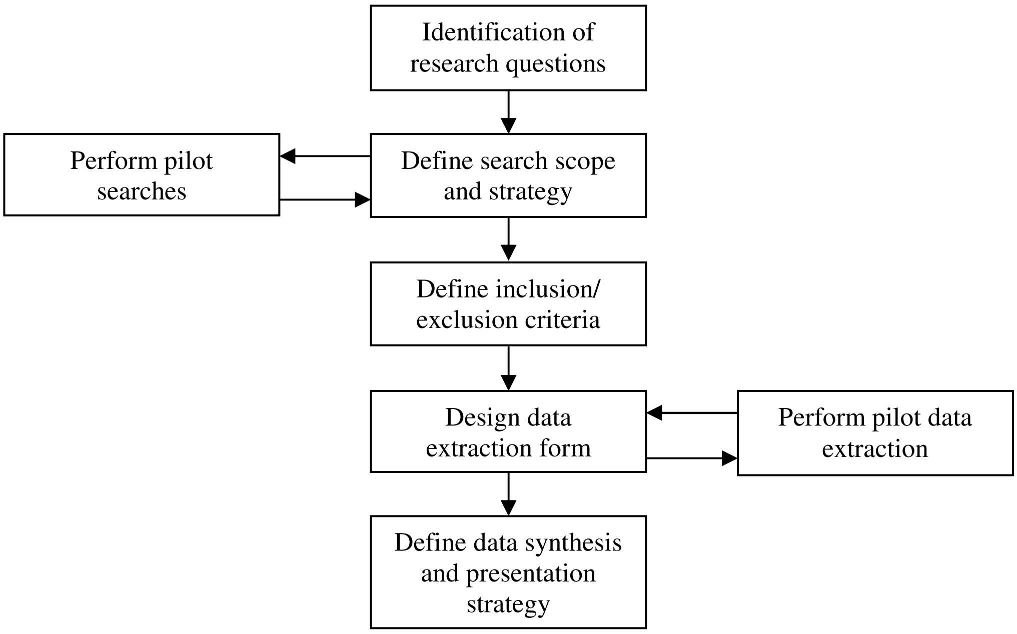
Identification of research questions
Perform pilot searches
Define search scope and strategy
Define inclusion/ exclusion criteria
Design data extraction form
Perform pilot data extraction
Define data synthesis and presentation strategy

Figure 1: Development process for the review protocol.

After identifying the research questions (discussed in Section 3.2), we defined the search scope and decided on a search strategy (discussed in Section 3.3). At this stage we designed the search string to be used to search on various electronic sources (see Table 1 for the list of venues that we searched). As part of this step, we conducted a number of pilot searches to test the search string. Defining a good search string is important to get a high recall rate as well as a high precision. Once the search scope and strategy were defined, we developed a number of study selection criteria (discussed in Section 3.4). Specifically, we defined explicit criteria to include and exclude studies that were identified through the search phase. The next step was to decide on the data elements to be extracted which can provide important information in answering the research questions. We initially designed a preliminary data extraction form based on our initial understanding. To evolve and subsequently improve the data extraction form, we performed a small pilot study on eight relevant studies that we had identified during the pilot search phase. Pilot data extraction step helped us to finalize the data elements that need to be extracted during the data extraction phase of the review. As a final step in designing the protocol, we decided our strategy to synthesize the extracted data and how to present the results of this synthesis.

3.2 Research Questions

There have been many conflicting claims regarding the benefits of AOP. Thus, our main goal is to summarize evidence related to those claims. Hence our first research question is:

RQ-1: *What empirical evidence has been presented in the research literature regarding the benefits and limitations of aspect-oriented programming in comparison to non-aspect oriented programming approaches?*

It has been reported several times that the results of a systematic literature review can only be as good as the evidences available [14]. The overall strength of a body of evidence is usually referred to as *strength of evidence* [15]. An analysis of the strength of evidence is very important for readers of an SLR to know how much confidence they can place in the conclusions and recommendations arising from such reviews. Hence, the second research question explored in this SLR is:

RQ-2: *What has been the strength of evidence in support of the stated findings?*

3.3 Search Strategy

For a systematic review, a well-planned search strategy is very important so that every relevant piece of work can be expected to appear in the search results (high recall [15]) without being cluttered by irrelevant studies (high precision [15]) [16].

We describe our search strategy from the following dimensions: search scope (both time and space), search method (i.e., automatic search or manual search), search strings, and electronic data sources used. By electronic data sources, we mean both index engines (e.g., web of science and EI Compendex) and publishers' sites (e.g., ScienceDirect and IEEEXplore).

Search Scope: We limited our literature search over two dimensions: publication period (time) and publication venues (space). In terms of publication period, we limited our search to papers published over the period of July 1997 and July 2008. We chose the start date as July 1997 because the first paper on AOP (i.e., Kiczales et al. [1]) appeared in ECOOP'97. The end time is July 2008, because we performed our search at this time. Hence, any paper published after July 2008 is not included. In terms of publication venues, we selected 15 venues (6 conferences and 9 journals). These venues were enlisted in first column of Table 1. AOSD, TAOSD, ECOOP, and OOPSLA were selected because they are well-known venues where AOP researchers are likely to publish their research results. ICSE, ASE, FSE, IEEE Software, IET Software, TSE, TOSEM, JSS, IST, SPE, and SQJ were selected because they are known for publishing high quality software engineering papers in general.

Search Method: We used two search methods, automatic search and manual search. Automatic search refers to the search performed by executing search strings on search engines of electronic data sources. Manual search refers to a search performed by manually browsing journals or conference proceedings. We tried to use manual search whenever the effort required was affordable because manual search can avoid missing relevant literature compared to automatic search. However, for some journals, the number of papers published in them can be over several thousands; and manually browsing all the papers of such a journal is too time-consuming. Thus we used automatic search for those journals that were expected to contain several thousands of papers. The column "Search Method" of Table 1 shows which search method was used for each venue. All conference venues were searched manually; and all journals (except TAOSD) were searched by automatic search.

Search String and Electronic Data Sources: For the automatic searches, we used the following search string:

((aspect AND oriented) OR aspect-oriented OR ((crosscut OR crosscutting OR cross-cutting) AND concern) OR pointcut OR joinpoint OR 'join point' OR aspectj)

This search string was constructed after performing a number of pilot searches (see Figure 1). Different electronic data sources provide different features (e.g., different field codes and syntax of search strings). When executing the search string on each electronic data source, we constructed a semantically equivalent search string for each electronic data source. Although the exact search strings executed on those electronic data sources are different, all of them are semantically equivalent to the above search string. The electronic data source used for each venue is shown in the "Electronic Data Sources" column of Table 1. It can be seen that the electronic data sources we used do not include index engines, such as Web of Science, EI Compendex, and Google Scholar. This is because we selected the electronic data source provided by the publisher if multiple electronic data sources were available for a venue.

Table 1: Overview of search results and study selection

Venues	Papers retrieved	Papers full-text Read	Papers selected	Search Method	Electronic Data Sources
AOSD	125	39	6	Manual	NA
TAOSD	32	12	2	Manual	NA
ECOOP	265	6	3	Manual	NA
OOPSLA	334	10	2	Manual	NA
ICSE	568	19	3	Manual	NA
ASE	543	4	0	Manual	NA
FSE	335	6	1	Manual	NA
IEEE Software	124	5	0	Automatic	IEEEXplore
IET Software	335	7	2	Automatic	IEEEXplore
TSE	195	9	1	Automatic	IEEEXplore
TOSEM	148	4	0	Manual	IEEEXplore
JSS	33	0	0	Automatic	ScienceDirect
IST	154	5	0	Automatic	ScienceDirect
SPE	25	5	1	Automatic	Wiley InterSc.
SQJ	91	2	1	Automatic	Springer
	3307	133	22		

AOSD	- International Conf. on Aspect-Oriented Software Development
TAOSD	- Transactions on Aspect-Oriented Software Development
ECOOP	- European Conference on Object-Oriented Programming
OOPSLA	- Int. Conf. on Object Oriented Prog., Systems, Lang., & App.
ICSE	- International Conference on Software Engineering
ASE	- International Conference on Automated Software Engineering
FSE	- International Symposium on Foundations of Software Engineering
TSE	- IEEE Transactions on Software Engineering
TOSEM	- ACM Transactions on Software Engineering Methodology
JSS	- Journal of Systems and Software
IST	- Information and Software Technology
SPE	- Software – Practice and Experience
SQJ	- Software Quality Journal

Table 1 also shows the count of the studies in each step of the study selection process (see Figure 2). For automatic search, the number in the column "Papers retrieved" indicates the number of papers returned by the electronic data source after running the search. For manual searches, the number indicates the number of papers browsed (i.e., the number of papers in each venue). The column "Papers full-text Read" indicates the number of papers left for each venue after primary study selection on the basis of title and abstract. We read the full texts of all these papers. The column "Papers selected" indicates the number of papers finally selected from each venue. All these papers were critically analyzed.

Figure 2 below shows the process for selecting papers, divided into three stages. The first stage was the literature search (both manual and automatic) identified 3,307 papers. In the second stage, we excluded irrelevant papers based on their title and/or abstract. This step resulted in a set of 133 papers. In stage three, we read the full text of these papers in order to select relevant studies based on our selection criteria. This is discussed in further detail in Section 3.4.

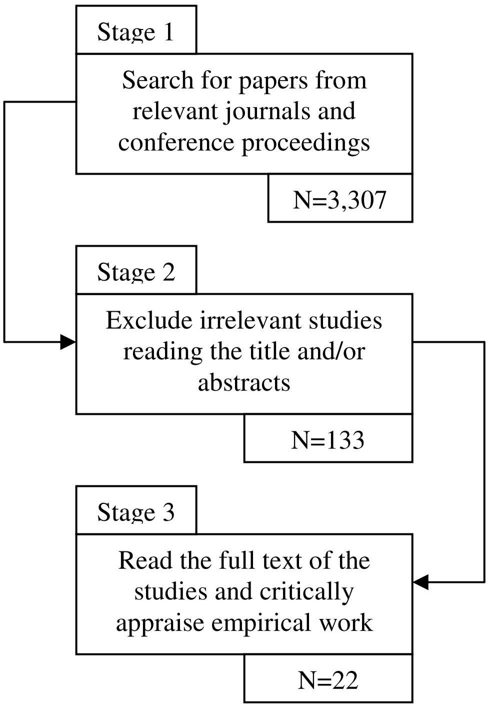
Stage 1
Search for papers from relevant journals and conference proceedings
N=3,307
Stage 2
Exclude irrelevant studies reading the title and/or abstracts
N=133
Stage 3
Read the full text of the studies and critically appraise empirical work
N=22

Figure 2: Paper selection process

3.4 Selection Criteria

From the papers published in the venues enlisted in Table 1, we identified 133 papers reporting empirical evidence on AOP. However, we included only those papers that reported empirical studies which compared AOP based solutions with non-aspect-oriented solutions. The following are the exclusion criteria:

- Editorials, position papers, keynotes, reviews, tutorial summaries and panel discussions;
- Papers reporting lessons learned, expert judgments or anecdotal reports, and observations;
- Papers that compare different AOP techniques or models (rather than AOP versus non-AOP techniques);

- Studies that provide empirical evidence of the claimed benefits of a specific AOP model, framework, or technique but do not provide a comparison with any non-AO counterpart.

One of the included papers ([E]) was reported in more than one venue. We selected the one with a more thorough account of the empirical work. We kept the record of paper citations and inclusion/exclusion decisions using a citation management software and a spreadsheet application.

3.5 Quality Assessment

We performed the quality assessment checks on the selected studies during this step. Table 2 lists the set of questions against which each of the selected studies was assessed for the quality of the used method and the quality of the reporting. The quality assessment instrument was adopted from the instrument used by Dybå and Dingsøyr [17]. This instrument was informed by those proposed for the Critical Appraisal Skills Programme (CASP) (in particular, those for assessing the quality of qualitative research [18]) and by principles of good practice for conducting empirical research in software engineering [19]. As confirmed by the authors (via personal communication), the instrument has general applicability to assess the quality of empirical studies in software engineering [20]. Hence, we adopted this instrument in our study. However, instead of using a dichotomous scale during the assessment, as used by Dybå and Dingsøyr [17], we decided to use a three point scale to answer each question, either as 'Yes, 'To Some Extent' or 'No'. By using a three point scale, we did not neglect the statements where authors provided only limited information to answer the assessment questions. Answers to every quality assessment question were assigned numerical values (i.e., Yes = 1, No = 0, and To Some Extent = 0.5). A quality assessment score for a study was given by summing up the scores for all the questions for a study. This overall score was considered as the study's quality assessment score. Results of the quality assessments are provided in Table 10 (Appendix-II). We discuss the results of the quality evaluation in Section 4.2.

Table 2: Quality assessment checks (adopted from [17])

No.	Question
Q1	Is there a rationale for why the study was undertaken?
Q2	Is there an adequate description of the context (e.g. industry, laboratory setting, products used etc.) in which the research was carried out?
Q3.	Is there a justification and description for the research design?
Q4.	Has the researcher explained how the study sample (participants or cases) were identified and selected, and what was the justification for such selection?
Q5.	Is it clear how the data was collected (e.g. through interviews, forms, observation, tools etc.)?
Q6.	Does the study provide description and justification of the data analysis approaches?
Q7.	Has 'sufficient' data been presented to support the findings?
Q8.	Is there a clear statement of the findings?
Q9.	Did the researcher critically examine their own role, potential bias and influence during the formulation of research questions, sample recruitment, data collection, and analysis and selection of data for presentation?
Q10.	Do the authors discuss the credibility of their findings?
Q11.	Are limitations of the study discussed explicitly?

3.6 Data Extraction

The selected primary studies were read in depth in order to extract the data needed to answer the research questions. Three researchers read the selected papers in parallel. Data were extracted based on a detailed set of questions. Some of the fields of our data extraction form included: study ID, targeted domain, study aim, crosscutting concerns aspectized, studied characteristics, metrics used, study findings, assessment approach, type/size of the system, AOP/non-AOP languages used, research method used, and type of subjects. We kept a record of the extracted information in a spreadsheet for subsequent analysis. We noted the lines and/or paragraphs of the paper where the information was located. This approach helped us to quickly locate and validate the extracted information, and resolve disagreements. This helped to increase our confidence that the extraction process was consistent and minimally biased.

3.7 Data Synthesis and Aggregation

During an SLR, the extracted data should be synthesized in a manner suitable for answering the questions that an SLR seeks to answer [13]. For the reported SLR, we decided to perform descriptive synthesis of the extracted data and to present the results in tabular form. Analysis of the data revealed that each study investigated (either qualitatively or quantitatively) the effect of AOP on one or more *characteristic*, and concluded with some appraisal or critique of the use of the AOP in comparison with a non-AOP approach. For the purpose of our review, we consider a 'characteristic' as any property or feature of the software system, development aid, or process, which is affected by the use of the AOP paradigm, and where such an effect is supported by evidence. In cases where the effect on a characteristic is studied indirectly while presenting intuitive analytical remarks or expert judgment with no direct support through any empirical data (such as tabular or graphical presentation of measurements), the results are not considered in our review.

In order to synthesize the extract data to answer RQ-1 (Section 3.1), we model the question 'what are the benefits and limitations of AOP?' as 'what *effect* AOP has had on the studied characteristics?' In empirical research, effect of a treatment is signified by *effect size,* which is an indicator of its magnitude. Effect size measures can be standardized or un-standardized [21-22]. Standardized effect size measures are independent of the scale since these are defined in terms of variability in data. Un-standardized measures are expressed in original scale, and are thus easier to interpret than standardized measures. Studies in this review mostly present effect size in terms of un-standardized measures involving both quantitative and qualitative investigations. Combining effect size results from these studies is not possible since studies not only examine different characteristics but also employ different metrics to study the same characteristic. The reported data in most cases is also very limited. Hence, a Meta-analysis cannot be performed in such situations [23]. A possible option could be the *vote counting* method, which does not depend on actual effect size values and metrics. Though we are aware that the use of vote counting as an aggregation approach has been discouraged [24], researchers in empirical software engineering have also argued that in situations like ours, it is the only viable method [23, 25].

4. Overview of the Reviewed Studies

This section presents information related to the method, type and setting of the 22 selected primary studies[3]. Year-wise distribution of the studies revealed that over the past decade there has been a rise in the number of studies reported (see Figure 3). This depicts an increased interest from the community in investigating the usefulness and usability of modularizing crosscutting concerns. A majority of the selected studies (63.6%) were published in highly ranked software engineering conferences such as ICSE, OOPSLA, ECOOP, and AOSD.

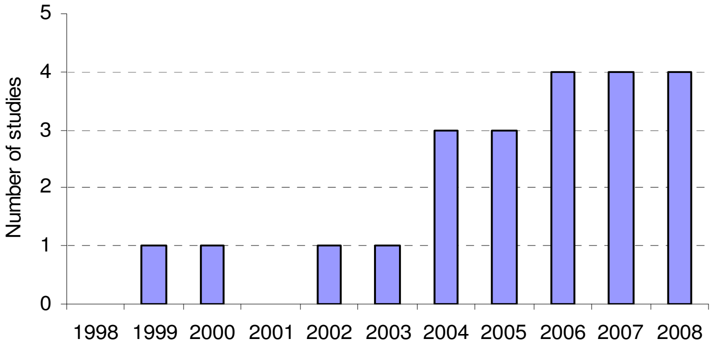

Figure 3: Year-wise distribution of studies.

4.1 Reported Research Methods

It is important that a suitably designed and rigorously conducted empirical study follows a well defined research methodology to ensure the reliability and validity of the findings. An empirical study is expected to explicitly report and justify the used research methodology and its related logistics. Table 3 provides information about the type of research methods reported by the authors in the reviewed studies. It is evident that 'case study' and 'experiment' research methods are the dominant approaches used by researchers to evaluate and compare their AO-solutions. Out of 22 selected studies, only one study reported the use of 'simulation' method, while two studies utilized benchmarks to evaluate the performance of aspectized code. Three studies, however, do not state their selected research method.

Table 3: Research methods reported

Method	Studies	Number	Percent
Case Study	[A],[B],[D],[F],[G],[J],[Q],[S],[T]	9	40.9 %
Experiment	[C],[I],[M],[O],[P],[R],[U]	7	31.8 %
Benchmarking	[H],[L]	2	9.09 %
Simulation	[K]	1	4.54 %
Not mentioned	[E],[N],[V]	3	13.6 %

4.2 Methodological Quality

While assessing a study's quality, it is usually difficult to separate the methodological rigor used for the research reported in a paper from the quality of reporting the

[3] Appendix-I lists the references to these studies.

research in that paper [14]. We therefore grouped the first eight questions as assessments checks for the quality of reporting and rigor, the results of which are listed in Table 4. Table 5 shows the results based on the scores for the last three questions, which are related to the evaluation credibility.

Table 6 summarizes the overall rating of the reviewed studies. It should be noted that our rating scale is non-linear. We refer the reader to Table 10 (in Appendix II) for the detailed quality profile of a specific study. Below, we briefly discuss the observations from the information provided in these tables.

- Table 4 shows that a majority of the reviewed studies are good in terms of presentation. However, one can observe the sparse entries (almost one third) in the columns Q3 and Q6 in Table 10 which are related to the description and justification of research design and data analysis. Three studies ([G, M, U]) appeared to be the most rigorous and well documented.
- It is clear from Table 10 that, except for two studies ([O, R]), the majority of the studies do not explicitly discuss the issues of bias, validity and reliability of their findings. Only 4 studies ([M, O, R, U]) discuss validity issues, while only half of the studies discuss the limitations of their approaches. This renders the overall strength of the evidence reported in these studies very low.
- From Table 10, it should be noted that none of the studies got a full score on the overall quality assessment criteria. Six studies [D, G, M, O, R, U] scored 9 or higher on the overall quality rating and are considered high quality studies. Five studies [B, H, N, P, V] are considered poor in our quality rating. Studies with high overall quality score are referenced in bold while those with poor quality rating appear in italics in Table 8, 10 and 11.

Table 4: Rating quality of reporting and rigor of studies

Quality Score	Poor (0 – 5)	Fair (5.5 – 6.5)	Good (7 – 8)
Number of studies	4	9	9
Percentage of papers	18.2%	40.9%	40.9%

Table 5: Rating credibility of evidence of included studies

Quality Score	Poor (0 – 1)	Fair (1.5 – 2)	Good (2.5 – 3)
Number of studies	18	2	2
Percentage of papers	81.8%	9.1%	9.1%

Table 6: Overall quality rating of included studies

Quality Score	Poor (0 – 5.5)	Fair (6 – 7)	Good (7.5 – 8.5)	High (9 – 11)
Number of studies	5	6	5	6
Percentage of papers	22.7%	27.2%	22.7%	27.2%

While reading the papers, we found it quite difficult to extract the information regarding the data collection and analysis approach as many authors did not report how the data were gathered and analyzed. We also found that a majority of the reviewed studies referred to their approach as ‘case study’ or ‘experiment’ (Table 3). However, most of the studies did not provide any justification for the adopted methodology. Hence, it may not always be clear to a reader why a particular research method was adopted. For both practitioners and researchers, it may be helpful to know the motivation for the research design, in order to assess the relevance and the reliability of the results. These results suggest that there is a lack of knowledge and understanding regarding the selection, design and conduct of an empirical study.

4.3 Study Settings

An overview of the contexts and settings in which empirical evaluations are performed can reveal the level of empirical research practice in a discipline. However, it is difficult to delineate what constitutes the context or settings of an empirical study. We have also observed that studies provide limited information regarding their experimental setup, and in most of the cases it was not explicitly reported in the reviewed studies for this systematic review. Although we encountered studies conducted in different settings, we found three dimensions of the study settings common among all the studies: scope, studied objects and system type. Below, these are discussed in more detail. Table 7 provides an overview.

4.3.1 Scope

We found that none of the reviewed studies was conducted within an industrial environment. Rather, all of the studies were conducted in an academic/laboratory environment where application of the aspect-oriented solution was assessed within a limited scope. This situation is not surprising as it is generally not feasible in an industrial setting to develop same application (same feature set) with AOP and non-AOP approaches and then compare the results. However, it would be very useful to study a migration from OO to AOP in an industrial setting.

4.3.2 Studied objects

We observed that only four studies involved human subjects where the primary objective was to consider the cognitive impact of the aspect-oriented development process on the developers. The remaining studies only discussed software applications as the studied objects without any explicit account of the involvement of human subjects.

4.3.3 System type

Every reviewed study utilized some software system to study the effect of AOP. Based on the usage of the systems studied we categorized these into three types:

- ‘real world applications’ that had been or were being used in a real world practice,
- ‘sample applications’ that were or had largely been used in research studies
- ‘toy systems’ that were mostly pedagogical applications developed or used specifically for evaluation purposes in the study.

Only six studies (27.3%) considered mature real-world applications, while a majority of the studies focused on sample applications which were mostly medium sized systems. It is also evident from Table 7 that the studies evaluating the cognitive impact of AOP versus non-AOP on developers were carried out using ‘toy systems’ for manageability and simplicity of the reported studies.

Table 7: Three dimensions of study settings

View	Settings	Studies	Number
Scope	Industrial study	None	0
	Academic/Lab study	All	22
Studied objects	Humans	[M],[O],[R],[U]	4
	Applications	Remaining reviewed studies	18
System type	'Toy' system	[M],[O],[R],[U]	4
	Real World application	[C],[D],[G],[I],[L],[T]	6
	Sample application	[A],[B],[D],[E],[F],[H],[J],[K],[N],[P],[Q],[S],[V]	14

4.4 Metrics

We also looked at the metrics used in the reviewed studies. Our observation is that the work related to aspect-oriented metrics has borrowed most of the ideas from OO metrics, especially the work of Chidamber and Kemerer [26]. However, it may not be possible to apply all existing OO metrics straightforwardly to aspect-oriented software since AOP introduces new abstractions. As a result, most of the empirical studies conducted during the first few years after the introduction of the AOP concepts are anchored in qualitative assessment. Researchers have recently introduced several AOP metrics suites such as reported by Ceccato and Tonella [27], Sant'Anna et al. [5] and Zhao [28]. Table 12 in Appendix-II provides a detailed summary of the metrics used in different studies and the properties they measure quantitatively. Some of these metrics have been adopted in many studies included in our review, especially a metric suite proposed in [5] (a collection of size, coupling, cohesion and separation of concern metrics) has been used in almost one third of the reviewed studies which include [A, B, E, G, Q, T]. However, the use of this metric suite appears to be more frequent within a group of researchers who developed the metrics suite.

4.5 Systems investigated

Table 11 (in Appendix-II) summarizes the size and type of the studied systems reported the reviewed studies. The types of the studied systems include operating system, virtual machine, embedded software, middleware, frameworks and applications, which provides a good coverage of the typical types of software in industry. Regarding the size of the software systems, most of them fall into medium to large sized categories (in terms of lines of code), which show that the systems under investigation are not trivial and are comparable to industrial software systems.

4.6 Implementation Languages

Each of the reviewed studies compares an aspect-oriented version of a software system with its non aspect-oriented counterpart, which in almost all the cases was the original implementation of a system. We term the language that is used to implement the original non-AOP version as 'comparison language'. Java is the most frequently used comparison language; 19 out of 22 studies used Java (see Table 11). Apart from Java, two studies used C++, while three studies used C programming language.

AspectJ is the most frequently used language for AOP implementation. Eighteen studies (81.8%) used AspectJ; out of which 16 studies used AspectJ exclusively, while two studies, [A] and [G], used EJFlow and CaesarJ respectively along with

AspectJ. Two studies [J, K] used AspectC++, one study [C] used AspectC, and one study [I] used GluonJ language for programming in aspect orientation. It is interesting to note that except one study [C], which compares AOP versus non-AOP implementations in a structured paradigm, all remaining studies (95.45%) compared AO solutions with their OO counterparts.

5. Results and Discussion

This section presents the findings our analysis of the data extracted from the reviewed papers in order to answer the research questions. We anchor our presentation on the characteristics studied in the selected papers.

Table 8 lists 16 identified characteristics with cited references. Since detailed presentation of each characteristic in the study context is not feasible, we prefer to discuss closely related characteristics in concert. Characteristics are closely related when there is some common goal of evaluation. For example, changeability, extensibility, sustainability, maintainability, design stability etc. can be considered related, since these are all facets of evolvability in a software system and a common assessment objective is to understand evolution under AO. Grouping related characteristics into high-level characteristics (6 grey boxes in the left hand side of Table 8) not only depicts the common goals of the studies, but also to provides leverage to better understand the effect due to increased sample size.

5.1 Benefits and Limitations of AOP

Our assessment of the effectiveness and limitations of AOP is collectively based on the findings of individual primary studies regarding the effect of AOP on various characteristics. Typically, studies present four types of conclusive statements:

- Positive - when authors note improvement with AOP use compared to non-AOP implementations.
- Negative - when the consequences of introducing aspects are not beneficial in the context.
- Insignificant - when AOP solution does not yield better results than earlier solutions, or there is no significant evidence of improvement.
- Mixed - when the study concludes with a mix of above three statement types and does not provide any aggregated statement about the effect that AOP had on the studied characteristic.

These conclusions (summarized in Table 8) provide important information in answering our main research question (RQ-1). Below, we discuss six high-level characteristics and briefly discuss our conclusions on the effect of AOP. Our conclusions, however, are only based on a limited number of primary evaluations found in the reviewed studies. We maintain that since every study discusses the effect of aspect-orientation with specific goals, settings, and limitations, it is difficult to draw precise conclusions.

5.1.1 Performance

There are mixed reports regarding the performance of aspect-oriented solutions. We found nine studies examining performance related characteristics. Kourai et al. [I] attempted to improve execution performance with aspect-oriented application level scheduling by separation of the scheduling code. Using a special weaver for performance tuning of the selected application, they note improvements in average response time as compared to the original implementation. Lohmann et al. [J] conducted a practical case study of an embedded weather station product line, comparing C-based, OOP-based, and AOP-based implementations. They found that the C-based and AOP-based versions are comparable in performance and using AOP instead of OOP led to significant reduction in memory and hardware costs. The AOP-based implementation achieved good separation of concerns (SoC). Based on an experience of re-factoring major middleware functionalities into aspects, Zhang et al. [L] reported around 8% performance improvements on third party benchmarks with less overhead. Pratap et al. [P] also report improvements in throughput while using AOP to selectively enable and disable middleware functionality. They utilized aspects to subset middleware in order to reduce code bloat and configuration complexity. In their effort, they developed a framework (FACET) for aspect composition where aspects encapsulate optional features, enabling the user to select only those features that are necessary. They report an experiment with various event channel configurations in the presence and absence of CORBA. Their results indicate that CORBA-disabled FACET configurations significantly reduce the memory footprint to almost half. Footprint reductions by individual features (enabling one feature at a time), however, was not significant. It is important to mention that their study does not consider the size of the Object Request Broker (ORB) in their results. Coady and Kiczales [C] evaluated the runtime cost of introduced aspects in a Unix-based operating system kernel. They found that the AO implementation had negligible impact on performance with minor overheads. However, Siadat et al. [K] did not experience any performance improvement in their case study of applying optimizations to a network simulator. Harbulot and Gurd [H] considered the problem of tangling code in high performance scientific software, and treated parallelism as a separate concern. They conclude that the underlying design of the application is crucial for allowing aspect-oriented re-factoring. Hence, it can be concluded that it is still an open question whether or not the AOP paradigm can be successfully used for high performance scientific applications.

5.1.2 Code Size

Kiczales et al. [1] mentioned that AOP can significantly affect code size of an application by eliminating scattering and tangling in the code. Since then it has been a general expectation that aspects reduce a program's size by improving reuse and minimizing code duplication. Many studies have focused on this aspect of aspect-orientation. Study of Coady and Kiczales [C] observed a reduction in redundant code. Zhang et al. [L] report around 10KLOC or 40% reduction in the size of a middleware core by factoring out major middleware (ORBacus) functionality as aspects. They achieved around 35% fewer methods and 17% simplification in terms of the control flow. Cacho et al. [A] also experience reduction in lines of code (LOC) and number of exception classes while evaluating an AO model for exception handling implementation. Lippert and Lopes [N] specifically studied size of exception handling code for different aspect designs while partially reengineering a Java-based framework. They found large reductions in the amount of exception handling code present in the application – from 11% of the total code in the OOP version to 2.9% in the AOP version. They concluded that in the best-case scenario, exception detection and handling code was reduced by a factor of four, by using aspects and in the worst case LOC with aspects was of the same order of magnitude as the original implementation.

Besides the positive remarks above, certain situations might not yield lesser LOC in aspect-oriented implementations. Tonella and Ceccato's [M] observed that the code size was not significantly affected when re-factoring aspectizable interfaces. They conclude that one reason for this could be the relatively small size of code devoted to implementation of aspectizable interface methods. Madeyski and Szala [R] did not notice any significant effect on code size in an experiment to assess the impact of AO on design quality at source code level. Filho et al. [T] also did not find any major improvements in size measures while modularizing exceptional handling concerns in four software applications. Unlike the study reported in [N], they observed slightly higher numbers (0-13%) for various size metrics, although a few instances showed improvements (0.5%-6.5%). Having analyzed the findings reported about the affects of using AOP on the code size of an application, we can conclude that aspect-oriented implementations result in lesser number of lines of code, or at worst maintain the same application code size as non-aspect-oriented implementations. We also found that when crosscutting concerns are homogeneous, aspectization significantly reduce redundant code fragments.

5.1.3 Modularity

Modularity is one of the characteristics considered to be directly affected by the idea of AOP. Though many researchers have advocated the positive effect of AOP on modularity, only a few have attempted to empirically validate it. Each of the reviewed studies in some respect discusses modularity. However, only five (22.7%) of them presented evidential data. Hannemann and Kiczales [V] conducted a qualitative investigation to explore the effect of AOP techniques on the implementation modularity of design patterns. They compared Java and AspectJ implementations of the 23 Gang of Four (GoF) patterns [29] and found modularity improvements (with textual localization) in 17 cases where there was some form of crosscutting between a pattern's roles and its structure. Their assessment was based on modularity related properties: locality, reusability, composition transparency, (un)pluggability. However, they did not provide any justification for their choice of properties. Tonella and Ceccato [M] report improvements in modularity after migrating aspectizable interfaces to aspects. In their experiment such migration resulted in an increased cohesion of operations in each class, and a significant decrease in coupling with the interfaces. Lobato et al. [S] proposed an aspect-oriented architecture (ArchM) for code mobility, and studied its usability and usefulness while comparing with a non-AO architecture. Their assessment revealed that an aspect-oriented architectural solution promoted better modularity as it reduced overall architectural coupling by making inter-component relationships uni-directional. In another experiment, Madeyski and Szala [R] gathered various statistics to measure the effect of AOP on systems' modularity. Their study concludes that the effect was not confirmed, and points out the need for more detailed evaluation. Garcia et al. [E] conducted a quantitative replication of the same study reported in [V] based on coupling, cohesion and separation of concern metrics. Although their results show that most aspect-oriented implementations provided better separation of concern, in some cases it resulted in higher coupling and more complex operations. No general conclusion was drawn in their study, rather they discussed the situations when aspect introduction provides benefits and when it does not. We found that none of the reviewed studies reported any negative effect of AOP on modularity under a specific situation.

Madeyski and Szala [R] studied design quality of AOP versus OOP implementations in terms of modularity and size measures. They studied design quality at the source code level, and referred it as 'code quality' in their work. For the same project developed during the study, they gathered different statistics to measure which implementation, Java or AspectJ, was better with respect to modularity and size. They applied various statistical tests to analyze the data, and pointed out that the impact of AOP on design quality metrics (e.g. package level, and class/aspect level) was not significant. They conclude that varying software development skills to effectively make use of the features and limitations of AspectJ might have rendered AOP's effect on design quality insignificant. Their study, however, highlights the need for further assessment of this characteristic.

The study performed by Cacho et al. [B] considered the problem of applying design patterns in real world software where pattern roles are composed for required quality behavior. They argue that composing multiple patterns affects various concerns and pattern roles crosscut several business classes. In order to investigate how well AOP can improve separation of pattern roles and the consequences of aspectizing pattern compositions, they assessed 62 pair-wise compositions in three different systems. They noticed that aspectization results depend on the patterns involved, composition intricacies, and the application requirements. They found that there are situations in which aspectization is not straightforward and developers need to select among available design options. However, they did not draw a general conclusion. Hence, it can be concluded that this work needs to be extended in future research by using other AO languages and metrics.

5.1.4 Evolvability

Real-world software needs to evolve continually in order to cope with imperfections and changes in user requirements and operational environment. The nature of change actions can be corrective, adaptive and perfective (sometimes also preventive) [30]. Good software should thus be flexible enough to absorb required changes with minimum effort. Developing software where individual concerns are well modularized significantly aids in achieving desirable characteristics like stability, maintainability, changeability, and extensibility. Programming in aspect-oriented languages has been suggested a way to realize these characteristics. We found 11 studies in eight papers investigating the ability of aspect-oriented software to accommodate change.

Coady and Kiczales [C] conducted a longitudinal case study. They tracked the evolution of the FreeBSD operating system across three different versions. They introduced several aspects into version 2 code, and then rolled them forward into their subsequent releases in the next two versions. They focused on the evolution of specific crosscutting concerns in isolation. They found that in the AO implementation of each concern, changes to the concern itself were better localized due to textual locality, configuration changes mapped directly to modifications to pointcuts and/or makefile options, and aspectization solutions provided extensibility due to improved modularization. Gibbs et al. [F] have presented the results from a longitudinal study aimed at testing the sustainability of aspects under large scale evolution. During the study of restructuring the memory management subsystem (MMTk) of a virtual machine (RVM), they investigated whether the introduction of aspects had positive, negative or neutral impacts in different situations. Positive impacts of each aspect resulted from localization of the implementation of the crosscutting concerns, while negative impact resulted from weak representation of invariants, which led to new code being unintentionally encompassed by the introduced aspects. However, they reported that aspects in their study did no harm in terms of a coarse-grained assessment of change tasks and half of them did better than the original implementation. They concluded that aspects keep pace with changes and provide a means of better sustaining separation of concerns in system infrastructure software.

Two studies present empirical work to assess design stability (resistance to potential ripple effects under modifications [31]) of aspect-oriented implementations. Greenwood et al. [G] conducted a quantitative case study comparing OOP and AOP implementations. They implemented nine changes and assessed the overall maintenance effects on fundamental modularity properties and change impact analysis. Design stability was found to be stable, particularly when the change targeted crosscutting concern. Such changes tended to be more simple to apply and less intrusive. Figueiredo et al. [Q] reported similar observations from their study focusing on the evolution scenarios of two heterogeneous product lines. According to them, AO implementations for Software Product Lines (SPLs) tended to have a more stable design particularly when the required changes targeted optional and alternative features. Both studies conclude that aspectual decompositions are superior when considering the Open-Closed principle [32]. However, they did mention that AO mechanisms do not cope well when introducing new mandatory features, or changing a mandatory feature to an alternative one. Aspectual decomposition appeared to narrow down the boundaries of concern interaction.

Zhang and Jacobson [L] applied the principle of horizontal decomposition to the original monolithic implementation of ORBcus middleware (an implementation of CORBA specification [33]) and re-factored major middleware functionalities into aspects. To them, such modularization and isolation from the core architecture enabled better customization and configuration of the middleware. It resulted in around 17% simplification in terms of control flow and 22% reduction in coupling.

Bartsch and Harrison [U] conducted an exploratory study with 11 software professionals to study the effect of AOP on maintainability. They defined maintainability as understandability and modifiability and used corresponding metrics to measure the effect. Although the results of their experiment suggested that the object-oriented system under investigation may be more maintainable than the AO system, they could not find any statistically significant evidence for the effects of aspect-orientation. Tonella and Ceccato [M], however, report that overall maintenance time in case of AOP is likely to decrease. Although their study revealed lesser overall maintenance time in case of AOP (as compared to OOP), the maintenance task chosen in their study might have impacted the results. There is thus a clear need for more empirical evidence about the potential impact of AOP on this issue.

It is also important to note that only one study examining the process of change found an insignificant effect. Thus, the evidence suggests that AOP provides better support for evolution and maintenance than non-aspectized solutions.

5.1.5 Cognition

AOP offers new language constructs and mechanisms. It is thus important to explore how the new paradigm affects the cognitive dimensions of software development. We found two relevant characteristics investigated in four primary studies. One of these is *understandability*, which is considered as the degree to which the purpose of a system or component is clear to the evaluator/developer. Second is the *development efficiency* measured in terms of the time and effort spent to program AO code.

Walker et al. [O] conducted one of the first experiments to understand how the separation of concerns provided by AOP affects a programmer's ability to accomplish different kinds of tasks. They found that programmers might be better able to understand aspect-oriented programs when the effect of the aspect code has a well-defined scope. Although they considered the time taken for understanding and coding a few implementation tasks by the participants, they did not provide any conclusive statement on the effect of AOP on development efficiency. In another experiment, Bartsch and Harrison [U] asked 11 software professionals to implement three changes in both OOP and AOP versions of an online shopping system. They noticed that understanding and applying changes to AOP code took more time than the same activity for OOP code. It is clear that both studies have reported contradictory results. However, the experimental format in both the studies did not take into account a detailed investigation of how the system structure and programmers' experience might have impacted on their understanding of AOP code compared with OOP code. In another experiment, Tonella and Ceccato [M] observed that aspect-oriented refactoring of aspectizable interfaces resulted in a significantly lower understanding time as compared to OOP while performing maintenance tasks. They conclude that refactoring into aspectizable interfaces improves code understanding as it separates the implementation from primary class responsibility.

Madeyski and Szala [R] considered the hypothesis that well-separated concerns are easier to maintain and develop, and hence the development time using AOP should be less than for a non-AOP implementation. They conducted an empirical study with three students to evaluate software development efficiency and design quality. The participants were asked to develop OO and AO implementations of a web-based manuscript submission and review system following Extreme Programming (XP) practices. Based on the analysis of the data gathered using pre-defined metrics (e.g. number of acceptance tests passed, total development time, active and passive programming time), they found that AOP had no significant effect on development efficiency as all the developers took almost the same amount of active time to finish the project. One of the significant limitations of this study was the very small sample size. The authors themselves recommend the use of a larger sample size to reach conclusive findings.

Due to the contradictory results, AOP's effectiveness for improved understandability is questionable. We did not find any evidence that AOP has been successful in any setting in significantly improving development efficiency. The effect of AOP on cognition has been studied in three papers and the results are not encouraging.

5.1.6 Language Mechanism

With new language constructs, AOP offers new ways to implement traditional mechanisms. Exception handling is an important mechanism which has been investigated empirically by AOP research community. We identified four studies addressing this issue (see Table 8).

Lippert and Lopes [N] applied aspect constructs to modularize the exception handling code in a Java-based OO framework (JWAM). They observed that AOP offers better support for different configurations of exceptional behavior since the contract aspects could be (un)plugged at compile time. Filho et al. [T] performed a similar study but they report on the contrary that the reuse of exception handlers, is not straightforward as advocated by Lippert and Lopes; rather, it depends on a set of factors such as the type of exceptions, handler behavior and contextual information. They suggest that when exception handling is non-uniform, strongly context dependent and complex, the use of aspects cannot bring any benefits. Coelho et al. [D] conducted a detailed quantitative study on the effect of AOP on exception control flows. They evaluated how exception handling aspects interact with aspects implementing other concerns. Based on a comparison of OO versus AO versions of three applications, they discuss the results on the number of undetected exceptions, exceptions caught by subsumption and specialized handlers. They conclude that AO mechanisms negatively affect the robustness of exception aware software systems. Cacho et al. [A] recently presented an AO model for exception handling implementation (EJFlow) to address some of the limitations. Results of their quantitative comparisons among Java and AspectJ implementations indicate that AO implementation reduces the amount of code necessary to define exception interface, the effort to manage exception flows, and improves separation between normal and error-handling code.

Although two of the mentioned studies favor aspect-orientation, our analysis reveals that existing AO languages do not provide sufficient benefit over object-orientation in managing exception handling behavior. Effective joinpoint models are yet to be devised for robust exception handling mechanisms.

Table 8: Effect of AOP on studied characteristics – High quality studies are shown in bold, low quality studies are shown in italic.

High-level Characteristics		Characteristic	Positive	Negative	Insignificant	Mixed	Study Count
Product related (external)	Performance	Performance	[I], [J], [L], [*P*]	-	[C], [K]	[*H*]	7
		Memory consumption	[J], [*P*]				2
Product related (internal)	Code size	Code size	[A], [L], [*N*]	-	[**M**], [**R**], [T]	-	6
		Redundancy	[C]		-	-	1
	Modularity	Modularity	[**M**], [S], [*V*]	-	[**R**]	[E]	5
		Design quality	-	-	[**R**]	-	1
		Pattern composability	-		-	[*B*]	1
	Evolvability	Changeability	[C]	-	[**U**]	-	2
		Extensibility	[C]	-	-		1
		Sustainability	[F]	-	-	-	1
		Design Stability	[**G**], [Q]	-	-	-	2
		Maintainability	[**M**]	-	[**U**]		2
		Configurability	[C], [L]	-	-	-	2
	Cognition	Understandability	[**M**], [**O**]	[**U**]	-	-	3
		Development efficiency	-	-	[**R**]	[**O**]	2
	Language Mechanism	Exception Handling	[A], [*N*]	[**D**]	-	[T]	4
			24	2	10	5	Total: 42

5.1.7 Summary

This section provides a summary of the reported evidence (Table 8) on the effect of AOP on studied characteristics in relation to the assessed study quality (Appendix II-Table 10). It is evident that some of the reviewed studies have evaluated multiple characteristics. When one study evaluates multiple characteristics, we term evaluation of each characteristic as an *instance*. Our SLR identified 42 instances in the 22 reviewed studies. Table 8 shows that overall AOP provides improvement over non-AOP based solutions in 24 instances. There were only 2 instances where AOP was not found to be an appropriate approach in the studied context. However, it was not possible to assess the extent of a positive or negative effect as no common measures were used by different studies reviewed in this SLR; nor was the reported data uniformly quantitative or qualitative. We found 10 reports in which AOP did not provide much improvement or where the effect was insignificant. In five cases we were unable to categorize the effect.

In Table 8, references to studies which scored 'high' (9 or more, see Table 6) in the overall quality assessment appear in bold, and references which scored less than 6 ('poor' quality) are italicized. It can be noted that high quality studies report a range of effects – positive, negative, insignificant and mixed – whereas studies rated poor in quality assessment only report either positive or mixed effects. Also, except for 1 instance, all studies reporting negative or insignificant effects are 'good' or 'high' quality studies. It should also be noted that 4 of the 5 high quality studies were based on toy applications rather than real systems suggesting limitations to the extent that their results can be generalized. This is an important observation since it bears implications on how the overall effect on characteristics is interpreted.

We categorized sixteen individual characteristics into six related high-level characteristics: performance, code size, modularity, evolvability, cognition, and language mechanism. Performance related characteristics show improvements in AOP-based solutions. There were six positive reports (66.7%) out of 9 instances, and none of the studies found any significantly negative impact.. We did not find any report where effect of AOP on performance related characteristics was negative. We therefore conclude that AOP can enhance a system's performance where the context is similar as described in [I, J, L, P]. None of the instances were associated with high quality studies. Hence, there is a need for some high quality studies on real applications to confirm these effects.

This SLR found that code size related characteristics are studied in 7 instances. There were four studies, which reported significant reduction in code size and redundancy. According to three studies, code size change was either insignificant or it slightly increased. However, studies reporting positive effect ´were rated low in quality assessment as compared with studies reporting an insignificant effect. However, the high quality studies were performed on "toy" systems, so the negligible results may be restricted to small systems. Thus we conclude that in larger systems where concern scattering and tangling is expected to be widespread, introducing aspects is likely to significantly reduce number of lines of code.

Modularity related characteristics were studied in 7 instances. Three studies reported improvement in modularity, one of which ([M]) is a high quality study. One high quality study [R], however, found the effect on modularity and design quality to be insignificant. In two studies the effect was largely dependent on the problem context and the authors could not draw overall conclusions based on their findings. Our conclusion is that although AOP can result in modularized structure of a system but the context in which AOP is used should be carefully assessed.

Evolvability related characteristics are the most studied characteristics found in this review. There are 8 positive reports, 6 of which are of ‘good’ or ‘high’ quality; there are no reports of a negative effect of AOP. Although only one study [U] finds insignificant improvement, it is a high quality study. Improvements have been reported in various facets of evolvability which include changeability, extensibility, sustainability, design stability, maintainability and configurability. We believe that AOP has the potential to develop evolvable and maintainable software.

An often raised suspicion regarding AOP is its effect on cognitive process in software development [4]. It should be noted that all four studies examining this characteristic are rated high quality in our quality assessment. All of these studies used human subjects (see Table 7). However, the experimental setup and the objects of study appeared to be too small to observe a statistically significant difference. We did not find any large scale study. Kiczales et al. [1] have mentioned that "it is extremely difficult to quantify the benefits of using AOP without a large experimental study, involving multiple programmers using both AOP and traditional techniques to develop and maintain different applications". Our observations gained through this SLR corroborate their remarks. Overall the effect of AOP on cognitive is not encouraging but is an area that would benefit from some high quality studies in the context of real applications.

Exception handling was the only language mechanism studied comparatively. The only instance arising from a high quality study showed a significant negative impact in the context of a real application. Two instances of lesser quality showed a positive effect and one instance of lesser quality showed mixed effects. These results are unexpected given that exception handling was one area where AOP was expected to have a significant impact. Here, we conclude that AOP does not behave better than its non-AOP counterparts. In our opinion, this is certainly the area which needs further research.

Table 8 shows that, except for two (development efficiency and exception handling), all of the studied characteristics are attributes of the software product. Product attributes can be internal or external [34]. External product attributes are characteristics that a user of a software system experiences during execution. Performance and memory consumption are external attributes. The remaining characteristics are internal product attributes. These are the characteristics which are only visible to a developer during the development or maintenance process. Among 24 improvement reports, 16 are related to internal product attributes, while six are related to external attributes. It is also obvious that, except for ‘code size’ and ‘memory consumption’, all product attributes are *quality attributes* mentioned in different quality models. This strongly implies that the research community regards AOP as a technology that ought to improve product quality. The readers should note that we have only reported on quality characteristics studied in the literature. Product quality models usually include many other characteristics such as reusability, verifiability, security, reliability, and a number of their related characteristics. It would be useful to investigate the impact of AOP on these quality characteristics.

An implementation paradigm can affect some of the process related characteristics as well. Currently, aspect orientation is not just an implementation concept but spans all other phases of the development, is now known as aspect oriented software development (AOSD). From implementation perspective, we need to know whether or not task allocation in AOP is different from non-AOP and the potential consequences. For example, can AOP help in ensuring that the application complies with the application specific standards, conventions and policy regulations in the domain? These and a number of other potential process related questions make understanding AOP's effectiveness an open are for research.

5.2 Strength of Evidence

We have already mentioned in Section 3.2 that it is very important for a reader of an SLR to know how much confidence he/she can have in the conclusions and recommendations arising from that SLR. Hence, this was the second research question of this SLR to address which we analyzed the overall strength of the body of evidence based on the reviewed studies. There are several systems exist for grading the strength of evidence [35]. We used the definitions from the Grading of Recommendations Assessment, Development and Evaluation (GRADE) working group, because the GRADE definitions addressed the weakness of most evidence hierarchy-based grading systems [13]. The GRADE definitions were also used by other software engineering researchers for grading the strength of evidence [14, 17].

GRADE defines four grades of strength of evidences: high, moderate, low, and very low (see Table 9). The strength of evidences is determined by the combination of four elements: study design, study quality, consistency, and directness. We will discuss the strength of evidences in the context of our study along the line of these four elements.

Table 9: Definitions used for grading the strength of evidence (adopted from [35])

Grade	Definitions
High	Further research is very unlikely to change our confidence in the estimate of effect.
Moderate	Further research is likely to have an important impact on our confidence in the estimate of effect and may change the estimate.
Low	Further research is very likely to have an important impact on our confidence in the estimate of effect and is likely to change the estimate.
Very low	Any estimate of effect is very uncertain.

With respect to study design, the majority of the primary studies were observational. Only seven (31.8%) primary studies are experiments (see Section 4.1). Thus, according to GRADE [35], our initial categorization of the total evidence in this review from the perspective of study design is low.

Regarding study quality, approaches of data analysis were not, in general, explained well; issues of bias, validity, and limitations were poorly addressed. Only three studies partially examined the possibility of bias introduced by researchers. In only four studies, the credibility of the study findings was discussed. As many as 11 out of 22 studies (50%) did not discuss the limitations explicitly (See Table 10). Based on these findings, we conclude that there are serious limitations in the quality of the studies.

Regarding consistency, which refers to the similarity of estimates of effects across studies [39], we found that the estimates of effects from different studies represent significant inconsistency. There are inconsistent results regarding effects on code size, modularity, changeability, understandability, maintainability, performance, and exception handling. These inconsistencies might be caused by insufficient control of various influencing factors and confounding factors. Because the confounding factors were not described in the report, we could not trace the cause of these inconsistencies. The results regarding effects on design stability, configurability, design quality, extensibility, sustainability, and memory consumption are consistent. However, each of these properties has been investigated by no more than two studies. Based on these findings, we conclude that in general the results lack consistency.

Directness refers to the extent to which the people, interventions, and outcome measures are similar to those of interest [35]. In the context of our study, people refer to subjects (e.g., students or software professionals) of the study; interventions refer to AOP approaches and non-AOP approaches, which are often embodied by the programming languages used (e.g., AspectJ, AspectC, Aspect C++ for AOP approaches, and Java, C, C++ for non-AOP approaches); outcome measures refer to the measures used to measure the properties (see Table 12 for details of these measures). With respect to people, only four studies (i.e., [M],[O],[R], and [U]) used human subjects. The subjects in [U] were software professionals. The subjects in [R] were graduate students; however, they were all experienced programmers. The subjects in [M] were a mixture of academics and programmers. The subjects in [O] were academics (i.e., graduate students and professors). In general, the characteristics of the subjects were close to (in the case of students) or representative (e.g., in the case of software professionals) software professionals. With respect to interventions, 18 out of 22 studies used AspectJ, which is the major AOP language available to practitioners. With respect to outcome measures, the measures used by the reviewed studies were also used in the real industrial settings. In addition, we also analyzed the settings (industrial or lab) where the studies were conducted and the systems investigated (as the objects of the studies) by these studies. None of the studies was conducted in an industrial setting (i.e., all in academic or lab settings). Most of the systems investigated are not trivial and are comparable to the software systems that the industrial practitioners deal with day-to-day (see Table 11). On the basis of these findings, our initial categorization of the total evidence in this review based on directness is between low to moderate.

Combining the four elements for grading the strength of evidence, we consider the strength of evidence in the current body of evidence regarding the benefits and limitations of AOP approaches compared to non-AOP approaches is low. Hence, any estimate of effect that is based on the body of evidence from current research cannot be considered very certain. Further research is definitely required to gain a reliable estimate of effects of AOP.

6. Limitations of the Review

The findings from this SLR could have suffered from following limitations (i.e., validity threats), which should be taken into account while interpreting or using the reported findings:

- Accuracy and consistency during the review process is based on a common understanding among the reviewers. Misunderstandings can result in biased results. One of the main limitations of the review can be the possibility of bias in the selection of studies. To help ensure that the selection process was as unbiased as possible, we developed detailed guidelines in the *review protocol* prior to the start of the review. During the paper screening phase, we documented the reasons for its inclusion/exclusion. Then we also rechecked the papers based on the inclusion/exclusion criteria.

- We found that many papers lacked sufficient details about the design and context of the reported studies. The findings were usually reported in a manner, which made it difficult to determine about the effect a study examined. Sometimes we had to infer certain pieces of information during the data extraction process. There is therefore a possibility that the data extraction process might have introduced some inaccuracy in the extracted data. To minimize this possibility, we decided to report such information based on the data presented in the reviewed studies For example, we have reported the research methods used for the reviewed studies as whatever authors of those studies claimed without any assessment of the research method against the available guidelines such as [23, 40-41]. Additionally, we held frequent discussions among the researchers involved in this review in order to clarify any ambiguity during the review process. This practice served as a way to recheck our results, ensure that there was consistency among individual researchers, and help resolve any disagreements. However, we were not able to recheck every piece of extracted information due to the limited time and resources. We selectively ran cross-checks during the different phases of this study.
- An account of evidence from the practice community could have been beneficial to compare findings with the research/academic community. However, following a detailed study selection and quality assessment criteria, we were left primarily with academic studies. We had to exclude studies that lacked scientific rigor. In our experience, research work reported by industry practitioners often falls into this category.

7. Conclusion

Several studies have been conducted to investigate the effects of AOP compared with non-AOP on characteristics of software development process and the developed software since 1997, when the term AOP was coined. To the best of our knowledge, there has been no effort to systematically identify, analyze, and synthesize the findings of the reported empirical studies. This paper presents the methodological details and results of our systematic review of the empirical studies reporting the benefits and limitations of AOP compared with non-AOP.

We identified 3,307 papers from searching the literature, of which only 22 were finally found to be the relevant primary studies reporting comparative empirical evidence. We identified a number of reported benefits and limitations of aspect-oriented software development from the perspective of the effect of AOP on certain characteristics. We observed that most of the reviewed studies have reported either positive or no significant effect of AOP compared with non-AOP approaches. The effect of AOP on performance, code size, modularity and evolvability related characteristics appear to be promising in the context similar to the reviewed studies. A few studies reported negative effect on certain characteristics. According to the findings, language mechanism, specifically exception handling is less likely to improve under current AOP models. AOP also appears to have performed poorly on cognitive dimension of software development during the reviewed studies. In many study instances, we noticed diverse findings. This highlights the perception of controversy regarding the applicability of aspect-oriented paradigm. There is a need to increase the number, quality and diversity of empirical studies on AOP. We have also observed that overall reporting and (or) conduct of the primary empirical studies lack methodological rigor in the sense that researchers rarely discuss the validity and limitations of their studies. It was also observed that a majority of the studies do not explicitly state the hypothesis which is being evaluated either qualitatively or quantitatively. We believe that there is a general lack of appreciation and understanding of designing, conduct, and reporting high quality empirical studies.

The findings of this SLR also enabled us to conclude that there is a significant potential for empirical software engineering researchers to systematically investigate the effect of AOP on various perspectives of software development. We found only a few of the product related characteristics which were examined. There are many other characteristics where comparative evidence can help in understanding not only the true potential of the approach, but also in pinpointing areas where improvement in technology should be sought.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero and by IRCSET grant no. RS/2008/134 to M. Ali Babar and Klaas.Jan Stol. We sincerely thank our colleagues for helping us during the search and study selection process. We are also thankful to the IST area editor, Barbara Kitchenham, and anonymous reviewers for providing useful comments to improve the paper.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Springer, 1997, pp.220-242

[2] R. Laddad, Aspect-oriented programming will improve quality, IEEE Software, 20 (2003), 90-91.

[3] E. W. Dijkstra, On the Role of Scientific Thought, in Selected Writings on Computing: A Personal Perspective: Springer-Verlag, 1982, pp. 60-66.

[4] A. Colyer, R. Harrop, R. Johnson, A. Vasseur, D. Beuche, C. Beust, Point/Counterpoint, IEEE Software, 23 (2006), 72-75.

[5] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, A. von Staa, On the reuse and maintenance of aspect-oriented software: an assessment framework, in: Proceedings of the 17th Brazilian Symposium on Software Engineering, 2003, pp.19-34

[6] F. Steimann, The paradoxical success of aspect-oriented programming, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), ACM, 2006, pp.481-497

[7] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, M. A. Kersten, Does aspect-oriented programming work?, Communications of the ACM, 44 (2001), 75-77.

[8] P. Tarr, H. Ossher, W. Harrison, J. Stanley M. Sutton, N Degrees of Separation: Multi-Dimensional Separation of Concerns, in: International Conference on Software Engineering (ICSE '99), ACM, 1999, pp.107-119

[9] K. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, USA, 1995

[10] W. Harrison, H. Ossher, Subject-oriented programming: a critique of pure objects, in: Proceedings of 8th International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), ACM, 1993, pp.411-428

[11] L. M. J. Bergmans, M. Akşit, J. Bosch, Composition filters: extended expressiveness for OOPLs, in: Proceedings of OOPSLA Workshop on Object-Oriented Programming Languages: the Next Generation, 1992

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Springer, 2001, pp.327-353

[13] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, Software Engineering Group, School of Computer Science and Mathematics, Keele University, EBSE Technical Report Version 2.3, Jul 2007.

[14] T. Dybå, T. Dingsøyr, Strength of evidence in systematic reviews in software engineering, in: Proceedings of the 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'08), ACM, 2008, pp.178-187

[15] D. L. Olson, D. Delen, Advanced Data Mining Techniques, Springer, 2008

[16] O. Dieste, A. G. Padua, Developing search strategies for detecting relevant experiments for systematic reviews, in: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM '07), IEEE Computer Society, 2007, pp.215-224

[17] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: a systematic review, Information and Software Technology, 50 (2008), 833-859.

[18] T. Greenhalgh, How to Read a Paper, BMJ Publishing Group, London, 2001

[19] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, J. Rosenberg, Preliminary guidelines for empirical research in software engineering, IEEE Transactions on Software Engineering, 28 (2002), 721-734.

[20] T. Dybå, T. Dingsøyr, G. K. Hanssen, Applying systematic reviews to diverse study types: an experience report, in: Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM'07), IEEE Computer Society, 2007, pp.225-234

[21] V. B. Kampenes, T. Dybå, J. E. Hannay, D. I. K. Sjøberg, A Systematic Review of Effect Size in Software Engineering Experiments, Information and Software Technology, 49 (2007), 1073--1086.

[22] V. B. Kampenes, T. Dyba, J. E. Hannay, D. I. K. Sjberg, A systematic review of effect size in software engineering experiments, Information and Software Technology, 49 (2007), 1073-1086.

[23] L. M. Pickarda, B. A. Kitchenham, P. W. Jones, Combining empirical results in software engineering Information and Software Technology, 40 (1998), 811-821.

[24] M. Ciolkowski, Aggregation of empirical evidence, in Empirical Software Engineering Issues: Critical Assessment and Future Directions, V. R. Basili, D.

Rombach, K. Schneider, B. Kitchenham, D. Pfahl, and R. W. Selby, Eds.: Springer, 2007, p. 20.

[25] J. Miller, Applying meta-analytical procedures to software engineering experiments, Journal of Systems and Software, 54 (2000), 29-39.

[26] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 20 (1994), 476-493.

[27] M. Ceccato, P. Tonella, Measuring the effects of software aspectization, in: Electronic Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE), 2004

[28] J. Zhao, Measuring coupling in aspect-oriented systems, in: Proceedings of the 10th International Software Metrics Symposium (METRICS'04), 2004

[29] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[30] L. Hatton, How accurately do engineers predict software maintenance tasks?, IEEE Computer, 40 (2007), 64-69.

[31] S. S. Yau, J. S. Collofello, Design stability measures for software maintenance, IEEE Transactions on Software Engineering, 11 (1985), 849-856.

[32] B. Meyer, Object-Oriented Software Construction, Prentice Hall, 1988

[33] OMG, The Common Object Request Broker: Architecture and Specification, Framingham, MA, USA: Object Management Group, 1999.

[34] S. McConnell, Code Complete, Microsoft Press, 2004

[35] D. Atkins, D. Best, P. A. Briss, M. Eccles, Y. Falck-Ytter, S. Flottorp, G. H. Guyatt, R. T. Harbour, M. C. Haugh, D. Henry, S. Hill, R. Jaeschke, G. Leng, A. Liberati, N. Magrini, J. Mason, P. Middleton, J. Mrukowicz, D. O'Connell, A. D. Oxman, B. Phillips, H. J. Schünemann, T. T.-T. Edejer, H. Varonen, G. E. Vist, J. W. Williams, S. Zaza, Grading quality of evidence and strength of recommendations, BMJ (British Medical Journal), 328 (2004), 1490.

Appendix-I: Selected Studies

[A] N. Cacho, F. C. Filho, A. Garcia, E. Figueiredo, EJFlow: Taming exceptional control flows in aspect-oriented programming, in: Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'07), ACM, 2008, pp.72-83

[B] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, C. Lucena, Composing design patterns: a scalability study of aspect-oriented programming, in: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), ACM, 2006, pp.109-121

[C] Y. Coady, G. Kiczales, Back to the future: a retroactive study of aspect evolution in operating system code, in: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, ACM, 2003, pp.50-59

[D] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, C. Lucena, Assessing the impact of aspects on exception flows: an exploratory study, in: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), Springer, 2008, pp.207-234

[E] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. v. Staa, Modularizing design patterns with aspects: a quantitative study, in: Proceedings

of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05), ACM, 2005, pp.3-14

[F] C. Gibbs, C. R. Liu, Y. Coady, Sustainable system infrastructure and big bang evolution: can aspects keep pace?, in: Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05), Springer, 2005, pp.241-261

[G] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, A. Rashid, On the impact of aspectual decompositions on design stability: an empirical study, in: Proceedings of 21st European Conference on Object-Oriented Programming (ECOOP'07), Springer, 2007, pp.176-200

[H] B. Harbulot, J. R. Gurd, Using AspectJ to separate concerns in parallel scientific Java code, in: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04), ACM, 2004, pp.122-131

[I] K. Kourai, H. Hibino, S. Chiba, Aspect-oriented application-level scheduling for J2EE servers, in: Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), ACM, 2007, pp.1-13

[J] D. Lohmann, O. Spinczyk, W. Schröder-Preikschat, Lean and efficient system software product lines: where aspects beat objects, Transactions on Aspect-Oriented Software Development II, (2006), 227-255.

[K] J. Siadat, R. J. Walker, C. Kiddle, Optimization aspects in network simulation, in: Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), ACM, 2006, pp.122-133

[L] C. Zhang, H.-A. Jacobsen, Resolving feature convolution in middleware systems, in: Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM, 2004, pp.188-205

[M] P. Tonella, M. Ceccato, Refactoring the aspectizable interfaces: an empirical assessment, IEEE Transactions on Software Engineering, 31 (2005), 819-832

[N] M. Lippert, C. V. Lopes, A study on exception detection and handling using aspect-oriented programming, in: Proceedings of the 22nd International Conference on Software Engineering (ICSE'00), ACM, 2000, pp.418-427

[O] R. J. Walker, E. L. A. Baniassad, G. C. Murphy, An initial assessment of aspect-oriented programming, in: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), IEEE Computer Society Press, 1999, pp.120-130

[P] R. M. Pratap, F. Hunleth, R. K. Cytron, Building fully customisable middleware using an aspect-oriented approach, IEE Proceedings-Software, 151 (2004), 199-216.

[Q] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp.261-270

[R] L. Madeyski, L. Szala, Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study, IET Software, 1 (2007), 180-187.

[S] C. Lobato, A. Garcia, A. Romanovsky, C. Lucena, An aspect-oriented software architecture for code mobility, Software: Practice and Experience, 38 (2008), 1365-1392.

[T] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranh, A. Garcia, C. M. F. Rubira, Exceptions and aspects: the devil is in the details, in: Proceedings of the 14th

ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06), ACM, 2006, pp.152-162

[U] M. Bartsch, R. Harrison, An exploratory study of the effect of aspect-oriented programming on maintainability, Software Quality Journal, 16 (2008), 23-44

[V] J. Hannemann, G. Kiczales, Design pattern implementation in Java and AspectJ, in: Proceedings of the 17th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), ACM, 2002, pp.161-173

Appendix-II: Tables

Table 10: Quality assessment scores

	Quality of Reporting and Rigor									Credibility of Evidence				
	Rationale	Description of Context	Design	Sampling	Data collection	Analysis	Sufficient data	Clarity of findings		Author bias	Validity	Limitations		
Study	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	***Sum (1-8)***	**Q9**	**Q10**	**Q11**	***Sum (9-11)***	**Total**
[A]	1	0.5	0.5	1	1	0	1	1	*6*	0	0	0	*0*	6
[*B*]	1	0.5	0	1	0.5	0.5	0.5	1	*5*	0	0	0	*0*	5
[C]	1	1	0	1	1	0.5	1	1	*6.5*	0	0	1	*1*	7.5
[D]	1	1	1	1	1	0.5	1	1	***7.5***	0.5	0	1	*1.5*	**9**
[E]	1	0.5	1	1	1	0	1	1	*6.5*	0	0	1	*1*	7.5
[F]	1	0.5	0	1	1	0	0.5	1	*5*	0	0	1	*1*	6
[G]	1	1	1	1	1	1	1	1	***8***	0	0	1	*1*	**9**
[*H*]	1	1	0.5	1	1	0	0.5	0.5	*5.5*	0	0	0	*0*	5.5
[I]	1	1	0	1	0.5	0.5	1	1	*6*	0	0	0	*0*	6
[J]	1	1	0.5	1	1	0.5	1	1	*7*	0	0	0.5	*0.5*	7.5
[K]	1	0.5	0.5	1	1	0.5	0.5	1	*6*	0	0	0	*0*	6
[L]	1	0.5	0.5	1	0.5	0.5	1	1	*6*	0	0	0	*0*	6
[**M**]	1	1	1	1	1	1	1	1	***8***	0	1	0	*1*	**9**
[*N*]	1	0.5	0	0.5	1	0	0.5	1	*4.5*	0	0	0	*0*	4.5
[O]	1	1	1	0.5	1	1	1	1	***7.5***	0.5	1	1	*2.5*	**10**
[*P*]	1	0.5	0	1	1	0	0.5	1	*5*	0	0	0	*0*	5
[Q]	1	1	0.5	1	1	0.5	1	1	*7*	0	0	1	*1*	8
[R]	1	1	1	1	0.5	1	1	1	***7.5***	0.5	1	1	*2.5*	**10**
[S]	1	1	0	1	1	0	1	1	*6*	0	0	0	*0*	6
[T]	1	1	0.5	1	1	0.5	1	1	*7*	0	0	1	*1*	8
[U]	1	1	1	1	1	1	1	1	***8***	0	1	0.5	*1.5*	**9.5**
[*V*]	1	0.5	0.5	1	1	0	0.5	1	*5.5*	0	0	0	*0*	5.5

Table 11: Studied systems

Study	**System**	**Type**	**Size** (typically LOC of non-AOP code, unless specified)	**AOP Lang./System**	**Comparison Lang.**
[A]	MobileMedia	Application	almost 4,000 LOC	AspectJ, EJFlow	Java
[*B*]	OpenORB compliant middleware	Middleware	"medium-sized", LOC not mentioned	AspectJ	Java
	Measurement tool	Application	"medium-sized", LOC not mentioned	AspectJ	Java
	Agent-based application	Application	"medium-sized", LOC not mentioned	AspectJ	Java
[C]	FreeBSD	Operating system	v2: 212,000, v3: 357,000, v4: 474,000 LOC	AspectC	C
[**D**]	Health Watcher	Web-based IS	v1: 6,080, v9: 8,825 LOC	AspectJ	Java
	Mobile Photo	Application	v4: 2,540, v6: 1,571 LOC	AspectJ	Java
	JHotDraw	Framework	21,027 LOC	AspectJ	Java
[E]	23 GoF design patterns	--	Not specified	AspectJ	Java
[F]	Memory manager toolkit (MMTk) within Jikes RVM	Virtual machine	Not mentioned	AspectJ	Java
[**G**]	Health Watcher	Web-based IS	more than 4,000 LOC in both Java and AspectJ	AspectJ, CaesarJ	Java
[*H*]	Java Grande Forum benchmark suite	Benchmarks	Raytracer 3177 LOC; LUFact 1049 LOC; Crypt: unknown	AspectJ	Java
[I]	Kasendas, a river monitoring system.	Application	9,238 LOC Java, 1736 LOC JSP,	GluonJ	Java
[J]	An embedded software product line for weather stations	Embedded software	1392-5008 bytes of object code	AspectC++	C++
[K]	IP-TN network simulator	Simulator	27 KLOC at core	AspectC++	C++
[L]	ORBacus	Middleware	23 KLOC at core; 13 KLOC after aspectization	AspectJ	Java
[**M**]	JHotDraw	Framework	39,214 LOC	AspectJ	Java
	FreeTTS	Application	31,099 LOC	AspectJ	Java
	JGraph	Framework	18,373 LOC	AspectJ	Java
	All classes below java in the package hierarchy of JDK	Library	382,533 LOC	AspectJ	Java
[*N*]	JWAM	Framework	44,000 LOC	AspectJ	Java
[**O**]	Digital library system (2 versions)	Application	Not specified	AspectJ	Java, Emerald
[*P*]	FACET, an implementation of CORBA event channel	Middleware	Class file 55,250 - 342,226 bytes without CORBA feature; 166,921 - 475,100 bytes with CORBA feature.	AspectJ	Java
[Q]	MobileMedia	Software product line	more than 3,000 LOC	AspectJ	Java
	BestLap	Software product line	almost 10,000 LOC	AspectJ	Java
[**R**]	Web-based manuscript submission system (3 versions)	Application	OO version 1: 4,378; OO version 2: 4,680; AO version: 3,895	AspectJ	Java
[S]	Expert Committee	Application	10,000 LOC	AspectJ	Java
	MobiGrid	Framework	699 LOC	AspectJ	Java
[T]	Telestrada	Application	3,350 LOC	AspectJ	Java
	Pet Store	Application	17,500 LOC	AspectJ	Java
	CVS Core Plug-in	Component	20,000 LOC	AspectJ	Java
	Health Watcher	Web-based IS	6,630 LOC	AspectJ	Java
[**U**]	Online shopping system	Application	460 NCLOC in Java; 490 NCLOC in AspectJ	AspectJ	Java
[*V*]	23 GoF design patterns	--	Not specified	AspectJ	Java

Table 12: Measured properties and metrics used

Property	Metric	Studies
Size	Lines of code (LOC)	[A],[B],[E],[G],[L],[N],[P],[Q],[T]
	Non-commented lines of code (NCLOC)	[R],[M]
	Number of modules (NOM)	[R]
	Number of attributes (NOA)	[A],[B],[E],[G],[T]
	Weighted op. in modules/components	[A],[B],[E],[G],[R],[T]
	Number of class operations (OP)	[M]
	Vocabulary size (VS)	[A],[G],[Q],[T]
Coupling	Coupling btw. Comp./modules (CBM)	[A],[B],[E],[G],[Q],[R],[T]
	Depth of inheritance tree (DIT)	[A],[B],[E],[G],[Q],[T]
	Efferent coupling (EC)	[L]
Cohesion	Lack of cohesion in operations (LCOO)	[A],[B],[E],[G],[Q],[R],[T]
Separation of concerns	Concern diffusion over comp. (CDC)	[A],[B],[E],[G],[Q],[T]
	Concern diffusion over operations (CDO)	[A],[B],[E],[G],[Q],[T]
	Concern diffusion over LOC (CDLOC)	[A],[B],[E],[G],[Q],[T]
Change impact	Num. of added/changed/removed comp.	[G],[Q]
	Num. of added/changed/removed op.	[G],[Q]
	Num. of added/changed/removed pointcuts	[G],[Q]
	Number of added/changed/removed LOC	[G],[Q]
Ease of Change	Time to complete the change	[O]
	Time spent on coding	[O]
	Time spent on analysis	[O]
	Lines of code written	[O]
Ease of debugging	Time required to correct each fault	[O]
	Number of file switches	[O]
	Number of instances of semantic analysis	[O]
	Number of builds per fault	[O]
Performance	Round-trip message invocation cost	[L]
	Data sending cost (min., avg., max.)	[L]
	Number of running threads per unit time	[I],[L]
	Cost of forking a process	[C]
	Time to switch b/w user and kernel mode	[C]
	Number of event per unit time	[P]
	Avg. real/profiled interval btw. Join points	[I]
	Time for thread suspension	[I]
Modularity	Response for module (RFM)	[R]
	Distance from main sequence (Dn)	[R]
	Operation cohesion	[M]
	Attribute cohesion	[M]
	Interface coupling	[M]
Memory consumption	Memory footprint (in bytes)	[J],[P]
Maintainability	Maintenance time	[M]
Understandability	Understanding time	[M]
Exception handling	Num. of 'catch' stat. per type of exception	[N]
	Number of exception paths	[D]
	Number of uncaught exceptions	[D]
	Num. of exceptions caught by subsumptions	[D]
	Number of specialized handlers	[D]
Architectural SoC	Concern diffusion over arch. components	[S]
	Concern diffusion over arch. Interfaces	[S]
	Concern diffusion over arch. operations	[S]
Code Coverage	Line coverage (percentage)	[R]
	Branch coverage	[R]
	Method coverage	[R]
Architectural Coupling	Architectural fan-in	[S]
	Architectural fan-out	[S]
Interface Complexity	Number of interfaces	[S]
	Number of operations	[S]
Software development efficiency	Number of acceptance tests passed	[R]
	Development time	[R]
	Active programming time, Passive time	[R]

List of Captions to Illustrations