Lech Madeyski and Łukasz Szała, “Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study”, IET Software Journal, vol. 1, no. 5, pp. 180–187, 2007. http://dx.doi.org/10.1049/iet-sen:20060071

Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study

Lech Madeyski, Łukasz Szała

Institute of Applied Informatics, Wrocław University of Technology,
Wyb.Wyspiańskiego 27, 50370 Wrocław, POLAND
`Lech.Madeyski@pwr.wroc.pl, Lukasz.Szala@e-informatyka.pl`

Abstract. The aspect-oriented programming approach is supposed to enhance a system’s features, such as its modularity, readability and simplicity. Due to a better modularization of crosscutting concerns, the developed system implementation would be less complex, and more readable. Thus software development efficiency would increase, so the system would be created faster than its object-oriented equivalent. An empirical study of a web-based manuscript submission and a review system is carried out to examine aspect-oriented vs. object-oriented approach with regard to software development efficiency and design quality. The study reveals that the aspect-oriented programming approach appears to be a fullfledged alternative to the pure object-oriented approach. Nevertheless, the impact of aspect-oriented programming on software development efficiency and design quality was not confirmed. In particular, it appeared that design quality metrics were not significantly associated with using aspect-oriented programming, instead of object-oriented programming. It is possible that the benefits of aspect-oriented programming will exceed the results obtained in this study for experiments with larger number of subjects.

1 Introduction

The aspect oriented programming (AOP) is a relatively recent approach that has been argued to better enable modularization of crosscutting concerns [1] and consequently hasten the development process. The hypotheses are that well separated concerns are more easily maintained, changed and developed, so the total programmer’s working time should be shorter than the development time of analogous system, realized without mechanisms offered by AOP. The validation of these hypotheses requires empirical studies.
The authors present the results of preliminary empirical evaluation of the impact of AOP on software development efficiency and design quality. Three experienced programmers, with recent industrial experience, were acquired to develop a web-based manuscript submission and review system, using AOP and the Object-Oriented Programming (OOP) approach.

The paper includes a comparison of developed AOP and OOP systems, based on software metrics proposed by Chidamber and Kemerer (hereafter CK) [2], Distance from the Main Sequence metric proposed by Martin [3], external code quality metric (defined as a number of acceptance tests passed) [4,5,6], and programmers' productivity metric. CK software metrics [2] were adapted to new properties of aspect-oriented software [7,8].
Subramanyam and Krishnan state that research on metrics for object-oriented software development is limited, and empirical evidence, linking the object-oriented methodology and project outcomes, is scarce [9]. Even more scarce is empirical evidence of the effect of aspect-oriented programming on software design quality, or development efficiency metrics. Therefore, the aim of this paper is to fill this gap and provide empirical evidence of the impact of aspect-oriented programming on software development efficiency and design quality metrics, as design aspects are extremely important to produce high quality software [9]. The hypothesis that design quality metrics are good predictors of the fault proneness is supported in [10] and [11].
The paper starts with the presentation of the related work in the field in Section 2. Section 3 contains a description of the examined system in terms of user requirements. Section 4 defines goals of the study, questions to be answered and metrics to be collected, in accordance with the Goal-Question-Metric (GQM) approach [12]. The results of the study are presented in Section 5, with the validity of the study in Section 6. The conclusions of the paper are presented in Section 7.

2 Related work

There is little related work focusing on the quantitative assessment of aspect-oriented solutions [13]. Kersten and Murphy [14] described the effect of aspects on object-oriented development practices, as well as some rules and policies that were employed to achieve maintainability and modifiability. Walker et al. [15] provided initial insights into the usefulness and usability of aspect-oriented programming. Soares et al. [16] reported that the AspectJ implementation of the Web-based information system has significant advantages over the corresponding pure Java implementation. Garcia et al. [17] presented a quantitative study, designed to compare the maintenance and reuse support of a pattern-oriented approach, and an aspect-oriented approach for a multi-agent system. It turned out that the aspect-oriented approach allowed the construction of the investigated system with improved modularization of the crosscutting agent-specific concerns. The use of aspects resulted in superior separation of the agent-related concerns, lower coupling (although less cohesive) and fewer lines of code. Tsang et al. [18] evaluated the effectiveness of AOP for separation of concerns. They applied the CK metrics suite to assess and compare an aspect-oriented and object-oriented real-time system in terms of system properties. They found improved modularity of aspect-oriented system over object-oriented system, indicated by the reduction in coupling and lack of cohesion values of the CK metrics. Hannemann and Kiczales [19], as well as Garcia et al. [20], have developed systematic studies that investigated the use of aspect-oriented programming to implement classical design patterns. It is worth mentioning that Tonella and Ceccato [21] performed an empirical assessment of refactoring the aspectizable interfaces. This study indicates that migration of the aspectizable interfaces has a limited impact on the principal decomposition size, but, at the same time, it produces an improvement of the code modularity. From the point of view of the external quality attributes, modularization of the implementation of the crosscutting interfaces clearly simplifies the comprehension of the source code. Unfortunately, most empirical studies involving aspects have been based on subjective criteria and qualitative investigation [13].

3 Empirical study

3.1 User requirements

The project was led with the eXtreme Programming methodology. Although some practices (such as pair programming practice) were neglected, user stories were used for the specification of requirements concerning the developed system. The whole set of 30 user stories was prepared to describe the complete, coherent system. The outlined application is a web-based manuscript submission and review system. It defines different user roles such as *Article Author*, *Reviewer*, *Conference Chair* and *Content Manager* and specifies multi-level authentication functionality. The workflow system involves the management of articles and their reviews on each step in their life cycle. Additionally, the application provides access to accepted and published articles to all registered and unregistered users.

3.2 Participants

Three experienced programmers took part in the study. Two of them (labelled as OO1 and OO2) developed the system using pure object-oriented programming approach, whilst one (labelled as AO1), implemented the system using aspect-oriented approach (he was the only programmer with about one year of AOP experience). The subjects are students taking the final MSc in Software Engineering diploma course concerning empirical assessment of novel approaches to software engineering. The additional selection criterion is based on Höst et al. [22] classification scheme. In particular it means that all participants can be classified as E4 according to Höst et al. classification scheme (i.e. recent industrial experience, between 3 months and 2 years). Moreover, the design and programming skills of participants were observed by the corresponding author in 18-months period during ambitious Java projects in industrial as well as academic settings (e.g. e-Informatyka portal, LaTeX2XML conversion service). It turned out that all of the participants possess similar and high design and programming skills. Nevertheless, OO2 developer has the richest programming experience (in years), and has developed the biggest software project among participants. Table 1 contains information gained from questionnaires and the differences in the programmers' experience. Although this information is given by programmers and represents their subjective estimates, the differences are substantial enough to take them into consideration during analysis. All programmers were subscribed to a mailing list, where misconceptions concerning user stories were explained.

Table 1. Comparison of programmers' experience in projects AO1, OO1 and OO2

	AO1	OO1	OO2
Experience in programming [years]	8	5	10
Programming in Java [years]	2	3	2
Size of the biggest project [$NCLOC$]	5000	2500	20000
Size of the biggest project in Java [$NCLOC$]	5000	1500	4500

3.3 Restrictions

In order to compare the projects' metrics, all the participants must obey the same rules. The technology used to develop the application were: `Java` and `AspectJ` programming languages with Java Servlets. The presentation tier was provided by JSP pages and all the persistent entities were kept in XML files. Each participant used the Eclipse Integrated Development Environment (IDE) with the ActivitySensor plugin installed. The source code was committed to an SVN repository after each completed user story.

4 Measurements definition

The empirical study was conducted using the GQM method, described in [12], to define measurements on the software projects. Two goals were defined. The first one relates to code quality (in terms of its modularity and size), and the other concerns software development efficiency.

Goal 1: *The evaluation of the AOP impact on code quality.*
Questions:

- **Question 1:** *How does AOP affect a system's modularity?*
  Metrics: The answer to this question is given by metrics Dn (Distance from the Main Sequence), CBM (Coupling Between Modules), RFM (Response For a Module) and LCO (Lack of Cohesion in Operations). The detailed description of Dn metric may be found in [3], whilst description of CBM, RFM and LCO metrics may be found in [7,8]. The lower the distance from the ideal line (Dn metric) is

measured, the better the system has been developed. The same rule is valid for CBM – modules that are not coupled with each other are better modularized and thus better manageable and easier to change. The RFM metric calculates the size of a potential communication between modules, and expresses a module modularity and complexity. The LCO metric points out the modules that are not cohesive, which means that they realize a functionality that should be divided, and thus the LCO metric is an indirect modularity indicator.

– **Question 2:** *How does AOP affect a system's size?*
  Metrics: Metrics that indicate a system's size are: $NCLOC$ (Non-Comment Lines Of Code), NOM (Number Of Modules) and WOM (Weighted Operations in Module). If the system is smaller (in terms of number of source code lines and modules), it is easier to grasp the programmer's intentions and comprehend the whole functionality. The WOM metric counts operations in a module (class, interface or aspect) and together with NOM metric indicates its size.

All the specified metrics were calculated by aopmetrics tool [8] developed at the Wroclaw University of Technology and used before to analyse software metrics [23]. The tool provides calculations of the same set of metrics for object-oriented, as well as aspect-oriented systems [7]. This allowed us to compare the data gathered from applications developed using both programming approaches.

Goal 2: *The evaluation of the AOP impact on software development efficiency.*
Questions:

– **Question 1:** *How does AOP affect software development efficiency in terms of the number of acceptance tests passed?*
  Metrics: A comprehensive set of 87 acceptance tests validated the degree to which requirement specifications were implemented. User stories have different sizes and different number of associated acceptance tests. This is why the $NATP$ (Number of Acceptance Tests Passed) metric is a more accurate indicator of a system completion than the number of implemented user stories. While an absolute measure is useful, it is beneficial to report a normalized measure $NATP/T$ (number of acceptance tests passed per development time) as well.

– **Question 2:** *How does AOP affect software development efficiency in terms of the time needed to develop the system?*
  Metrics: The programming paradigm may simplify the development process, so the software product may be created faster. In this case, the helpful metric is a total development time labelled as T. While an absolute measure is useful, it is beneficial to report a normalized measure $NCLOC/T$ (non-commented lines of code per development time) as well.

– **Question 3:** *How does AOP affect software development efficiency in terms of the active time and the passive time needed to develop the system?*

Metrics: The total development time spent on programming may be divided into *active time* and *passive time*, labelled as T_A and T_P, accordingly. The active time is when a programmer types, and passive time is when a programmer performs other activities concerning a project. If a programmer knows what to do and which part of the source code is to be changed, the fraction of passive time in total time is smaller.

The $NATP$ metric was collected manually. Each test is a sentence written in natural language that depicts a piece of functionality. The testing procedure checks whether statements in acceptance tests are true or false. The metrics T (total programming time) and T_A (active programming time) were calculated by ActivitySensor Eclipse plugin developed at the Wroclaw University of Technology [24].

5 Results

The data are analysed with descriptive analysis (in Section 5.1) and statistical tests (in Section 5.2).

5.1 Descriptive Statistics

Depending on a metric properties, the average, minimum, maximum and median values were calculated either for packages or classes.

Modularity The Normalized Distance from the Main Sequence (Dn) metric is calculated for packages, and the third row of Table 2 contains its average value. However, this indicator may be misleading, because packages have different sizes. In order to present more accurate value, a weighted average value of Dn has been calculated and presented in the fourth row of Table 2. Depending on calculation method, either AO1 (arithmetical average) or OO2 (weighted average) system is characterized by lower average value of Dn. However, the median value rather advocates aspect-oriented solution – in both object-oriented systems the median value is about twice as high as in AO1.

Table 2. Average, weighted average and median values of Normalized Distance from the Main Sequence (Dn) in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
Dn metric	Value	Value	Ratio	Value	Ratio
Average	$0,31$	$0,50$	$160,13\%$	$0,43$	$135,83\%$
Weighted Average	$0,41$	$0,45$	$111,32\%$	$0,35$	$86,65\%$
Median	$0,25$	$0,55$	$220,00\%$	$0,47$	$186,67\%$

The CBM metric is presented in Table 3. The average value is lower in AO1 system than in both OO1 and OO2 systems. The difference is higher than 30 per cent in both cases. A plausible explanation might by that new modularisation possibilities, provided by aspect-oriented programming, decrease coupling between modules (CBM) and allow enclosing functionality pieces in separate units.

Table 3. Average, maximum, minimum and median values of Coupling Between Modules (CBM) in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
CBM metric	Value	Value	Ratio	Value	Ratio
Average	$3,22$	$4,26$	$132,24\%$	$4,22$	$130,80\%$
Max	15	29	$193,33\%$	16	$106,67\%$
Min	0	0	$100,00\%$	0	$100,00\%$
Median	2	3	$150,00\%$	1	$50,00\%$

The RFM metric values presented in Table 4 seem to be similar in each system. The comparison of median and maximal values does not bring any clarification.

Table 4. Average, maximum, minimum and median values of Response For a Module (RFM) in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
RFM metric	Value	Value	Ratio	Value	Ratio
Average	$9,43$	$9,53$	$101,08\%$	$9,24$	$98,05\%$
Max	49	56	$114,29\%$	50	$102,04\%$
Min	0	0	$100,00\%$	0	$100,00\%$
Median	8	6	$75,00\%$	8	$100,00\%$

The LCO metric values are depicted in Table 5. Again, the AO1 system is characterized by the lowest level of lack of cohesion in operations. The lowest cohesion level (the highest LCO) was observed in OO1 system, where the average value of LCO metric is over twice as big as in AO1 or OO2 system.
The gathered descriptive statistics do not allow judging which programming approach provides more modular implementation. To answer the question whether the impact of aspect-oriented programming on design modularity metrics is significant, or not, statistical tests are performed in Section 5.2.

Size The usual value of WOM metric ranges from 1 to 10. Table 6 presents metric values for all three systems. The lowest average value is represented by AO1 system, but the differences do not seem to be large. The comparison of median and maximal values does not bring any

Table 5. Average, maximum, minimum and median values of Lack of Cohesion in Operations (LCO) in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
LCO metric	Value	Value	Ratio	Value	Ratio
Average	$16,6$	$41,4$	$249,79\%$	$17,06$	$102,89\%$
Max	446	849	$190,36\%$	350	$78,48\%$
Min	0	0	$100,00\%$	0	$100,00\%$
Median	0	0	$100,00\%$	0	$100,00\%$

clarification, either, because the median and maximum values of WOM for AO1 are between the values for OO1 and OO2.

Table 6. Average, maximum, minimum and median values of Weighted Operations in Module (WOM) in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
WOM metric	Value	Value	Ratio	Value	Ratio
Average	$4,79$	$5,33$	$111,42\%$	$5,27$	$110,08\%$
Max	34	50	$147,96\%$	29	$85,29\%$
Min	0	0	$100,00\%$	0	$100,00\%$
Median	3	2	$66,67\%$	3,5	$116,67\%$

Table 7 presents the size metrics. For $NCLOC$ and NOM metrics two values have been presented. One contains values counted for production code (without unit tests), whilst the other contains values for production code with unit tests.

Table 7. $NCLOC$ and NOM metrics in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
Size metrics	Value	Value	Ratio	Value	Ratio
$NCLOC$(w/o tests)	3895	4378	$112,40\%$	4680	$120,15\%$
$NCLOC2$(with tests)	7260	8850	$121,90\%$	14176	$195,20\%$
NOM(w/o tests)	89	87	$97,75\%$	78	$87,64\%$
$NOM2$(with tests)	131	144	$109,92\%$	138	$105,34\%$

The OO2 system was built using smaller number of production code modules (NOM), which (for the systems with similar LCO level) is desirable. When comparing production code with unit tests, the AO1 system turns out to be developed with the minimal number of modules. The explanation of this fact may lie in aspects that facilitate writing unit tests. The AO1 system includes aspects that set up the appropriate database state before and after execution of test methods, and separates application persistent data from unit test entities, saved during testing procedure. According to the second size metric ($NCLOC$), aspect-oriented system AO1 is smaller than object-oriented solutions OO1 and OO2. A plausible explanation might be a shift of some (e.g. logging) functionality, spread over classes to cohesive aspects. This relation is retained when comparing production code with unit tests ($NCLOC2$). In this case the difference is even bigger. It may suggest that object-oriented programmers were more thorough when writing unit tests, or the aspect-oriented mechanisms make tests even more concise.

Table 8 containing code coverage values may be helpful in resolving this issue. Code coverage indicates how much of the code has been covered by the tests. A number of code coverage measures have been proposed e.g. line, branch, or method coverage. The line, method, or branch coverage is a ratio of lines, methods, or branches executed during the testing procedure to the total number of lines, methods, or branches in a system. The main disadvantage of line coverage is that it is insensitive to some control structures. To avoid this problem, branch coverage has been devised. Branch coverage is a measure based on whether decision points, such as `if` and `while` statements, evaluate to both true and false during test execution, thus exercising both execution paths [25]. The calculated code coverage metrics[1] reveal the fact that line coverage and method coverage are slightly higher in the case of AO1 whilst branch coverage is slightly higher in the case of OO1.

Table 8. Code coverage metrics in AO1, OO1 and OO2 systems

	AO1	OO1		OO2	
Coverage metrics	Value	Value	Ratio	Value	Ratio
Line coverage %	88	85	95,45%	80	90,91%
Branch coverage %	93	95	102,15%	83	89,25%
Method coverage %	91	76	83,52%	91	100,00%

It is worth mentioning that unit tests were written by means of JUnit framework [26] by all developers, contrary to acceptance tests designed by requirements' giver and used to measure the development efficiency. Test-last and test-first programming strategies were applied by all developers during the course of development, as both strategies are used in professional software development. Test-last programming phase was followed by test-driven phase and then test-last phase occurred again. The points at which test-first programming was introduced and withdrawn were randomly determined in all projects. Test-first programming is a practice based on specifying piece of functionality as a low level unit test, before writing production code, implementing the functionality, so that the test passes, refactoring (e.g. removing duplication) and iterating

[1] Code coverage was measured by Cobertura http://cobertura.sourceforge.net/ as the tool provides code coverage metrics for object-oriented and aspect-oriented systems.

the process [27]. In the case of test-last programming, tests are specified after writing production code of the selected user story.
To answer the question whether the impact of aspect-oriented programming on size metrics is significant, or not, statistical tests are performed in Section 5.2.

Efficiency The results concerning software development efficiency are presented in Table 9. The ActivitySensor Eclipse plugin [24] allows to automatically collect development time and to divide total development time into active and passive times. The active time may be spent on typing and producing code, whilst the passive time is spent on reading the source code, looking for bugs etc. A switch from active to passive time happens after 15 seconds of a programmer's inactivity (the threshold was proposed by the activity sensor authors). After 15 minutes of inactivity, the passive time counter is stopped until a programmer hits a key.
The highest level of implemented functionality (in terms of acceptance tests passed) was achieved by AO1 project, although the OO2 project had only 1 acceptance test passed less. The OO1 managed to implement 90% of its competitors' functionality. Looking at the development time, one can conclude that OO2 system was implemented faster than OO1 and AO1. This observation is supported by normalized measures (non-commented lines of code per development time $NCLOC/T$ and number of acceptance tests passed per development time $NATP/T$). However, there is no correlation between the programming approach and $NATP/T$. It is interesting that all three developers needed nearly the same amount of active time to finish the project (similar level of the T_A metric value).

Table 9. Software development efficiency in AO1, OO1 and OO2 projects

	AO1	OO1		OO2	
Development efficiency results	Value	Value	Ratio	Value	Ratio
$NATP$	86	78	$90,70\%$	85	$98,84\%$
T Total Time [h]	$95,93$	$101,36$	$105,66\%$	$72,61$	$75,69\%$
T_A Active Time [h]	$42,79$	$46,94$	$109,69\%$	$44,29$	$103,50\%$
T_P Passive Time [h]	$53,13$	$54,42$	$102,42\%$	$28,32$	$53,30\%$
T_P / T [%]	$55,39$	$53,69$	$96,93\%$	$39,00$	$70,42\%$
$NCLOC$(w/o tests)/T	$40,60$	$43,19$	$106,38\%$	$64,45$	$158,74\%$
$NATP/T$	$0,90$	$0,77$	$85,84\%$	$1,17$	$130,58\%$

5.2 Statistical tests

The number of subjects who have taken part in the study is limited. However, since internal metrics are computed for each class, aspect or package in the system, the number of data appeared to be sufficient, as suggested by one of the reviewers, for the reliable execution of statistical tests. Statistical tests are performed to answer the question whether the impact of aspect-oriented programming on design quality metrics is significant, or not. Packages and classes developed according to object-oriented programming approach are labelled OO. The following null hypotheses are to be tested:

- $H_{0\ Dn,\ OO/AO}$ — There is no difference in the Dn metric values between packages developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ WOM,\ OO/AO}$ — There is no difference in the WOM metric values between modules (classes or aspects) developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ CBM,\ OO/AO}$ — There is no difference in the CBM metric values between modules developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ RFM,\ OO/AO}$ — There is no difference in the RFM metric values between modules developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ LCO,\ OO/AO}$ — There is no difference in the LCO metric values between modules developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ NCLOC,\ OO/AO}$ — There is no difference in the $NCLOC$ metric values between modules developed using object-oriented and aspect-oriented programming approach (OO and AO).
- $H_{0\ NCLOC2,\ OO/AO}$ — There is no difference in the $NCLOC2$ metric values between modules developed using object-oriented and aspect-oriented programming approach (OO and AO).

We start with exploratory analyses on the collected data to check whether they follow the assumptions of the parametric tests (i.e. normal distribution, interval or ratio scale, homogeneity of variance). The first assumption of parametric tests is that our data have come from a population that has normal distribution. Objective tests of the distribution are Kolmogorov-Smirnov and Shapiro-Wilk tests. These tests compare the set of scores in the tested sample to a normally distributed set of scores with the same mean and standard deviation. If the test is non-significant ($p > .05$) it tells us that the distribution of the sample is not significantly different from a normal distribution (i.e. it is probably normal). If the test is significant ($p < .05$) then the distribution in question is significantly different from a normal distribution (i.e. it is non-normal). It is worth noting that the Shapiro-Wilk test yields exact significance values and is thus more accurate (though less widely used) than the Kolmogorov-Smirnov test. In fact, the Shapiro-Wilk test results (see Table 10) are highly significant ($p < .05$), as the column labelled $Significance$ is less than .05 for both, OO and AO development methods in all cases.

The data distribution for OO and AO appears to be non-normal. This finding alerts us to the fact that non-parametric tests should be used. Therefore the hypotheses are evaluated using the Mann-Whitney one way analysis of variance by ranks. Tables 11 and 12 show test statistics and significances.

Table 10. Tests of Normality

Metric	Development	Kolmogorov-Smirnov[1]			Shapiro-Wilk		
	Method	Statistic	df	Significance	Statistic	df[2]	Significance
Dn	OO	.151	35	.041	.901	35	.004
	AO	.181	19	.102	.852	19	.007
WOM	OO	.255	165	.000	.594	165	.000
	AO	.227	89	.000	.665	89	.000
CBM	OO	.237	165	.000	.808	165	.000
	AO	.168	89	.000	.842	89	.000
RFM	OO	.188	165	.000	.810	165	.000
	AO	.163	89	.000	.831	89	.000
LCO	OO	.410	165	.000	.267	165	.000
	AO	.404	89	.000	.247	89	.000
$NCLOC$	OO	.221	165	.000	.675	165	.000
	AO	.224	89	.000	.744	89	.000
$NCLOC2$	OO	.233	282	.000	.637	282	.000
	AO	.177	131	.000	.863	131	.000

[1] Lilliefors Significance Correction.
[2] Degrees of freedom.

Table 11. Mann-Whitney Test Statistics (grouping variable: Development Method)

	Dn	WOM	CBM	RFM	LCO
Mann-Whitney U	259.000	6963.500	7302.500	6926.500	6828.500
Wilcoxon W	449.000	20658.500	20997.500	20621.500	20523.500
Z	-1.337	-.684	-.073	-.748	-1.066
Asymp.Sig.(2-tailed)	.181	.494	.941	.455	.286

Table 12. Mann-Whitney Test Statistics continued

	$NCLOC$	$NCLOC2$
Mann-Whitney U	7193.000	17347.000
Wilcoxon W	11198.000	25993.000
Z	-.268	-.996
Asymp.Sig.(2-tailed)	.789	.319

It turned out that Dn, package level design quality indicator, was not significantly affected by aspect-oriented programming approach (the Mann-Whitney test statistics: $U = 259.0$, non-significant, $z = -1.337$). Class level design quality indicators were also not significantly affected by aspect-oriented programming approach: WOM (the Mann-Whitney test statistics: $U = 6963.5$, non-significant, $z = -.684$), CBM ($U = 7302.5$, non-significant, $z = -.073$), RFM ($U = 6926.5$, non-significant, $z = -.748$), LCO ($U = 6828.5$, non-significant, $z = -1.066$), $NCLOC$ ($U = 7193.0$, non-significant, $z = -.268$) and $NCLOC2$ ($U = 17347.0$, non-significant, $z = -.996$).
An effect size ($r = \frac{Z}{\sqrt{N}}$ where Z is the z-score in Tables 11 and 12, and N is the number of observations i.e. 54 packages and 254 classes) is an objective and standardized measure of the magnitude of observed effect. The effect size is rather small for Dn ($r = -.18$) and is even smaller for all class level metrics ($r < -.07$).
Why did not aspect-oriented programming approach result in a significant improvement of software design quality, measured by package level, as well as class level design quality metrics? The plausible explanation is that software development skill may, to a certain extent, compensate for the lack of useful features of aspect-oriented programming. Another possible explanation is that AspectJ exhibits some limitations as a Java language extension.
Additionally, we decided to analyse the relationships between the subjects' experience and the design quality metrics. We used a questionnaire to measure the subjects' experience in programming. The subjects were asked to indicate the number of years experience in programming. All of the subjects had between 2 and 3 years of experience in programming in Java, and therefore this variable was neglected. However, there were differences between subjects' programming experience in general. It turned out that the relationships between programmer experience and class level design quality metrics were weak (Spearman's and Kendall's tau correlation tests were non-significant and far from a high degree of correlation) with the exception of WOM metric where there was a significant relationship ($r_s = .15$, $\tau = .13$, $p < .05$). Although Spearman's statistic is more popular, Kendall's statistic is used as well, as it is actually a better estimate of the correlation in the population [28]. The relationship between programmer experience and package level design quality metric (Dn) was weak and non-significant as well. Lack of significant relationships in almost all cases may be due to the fact that all programmers were at the same experience level (E4 according to Höst et al. [22] classification scheme).

6 Validity evaluation

To enable an analysis of the validity of the study, possible threats are discussed, based on the schema presented in [29].
As a *statistical conclusion* validity threat, the authors see limitations of inferential statistics used in the analysis, caused by a small number of

professional programmers who have taken part in the study. Actually, it was impossible to conduct statistical tests for some metrics e.g. $NATP$. However, since internal metrics are computed for each class, aspect or package in the system, statistical tests for the internal metrics have been conducted in Section 5.2. The validity of the experiment is highly dependent on the reliability of the measures, which can be divided into two classes: objective measures, and subjective measures [29]. Measures used in the study were selected to be objective rather than subjective e.g. lines of code are more reliable than function points since it does not involve human judgement [29].

Internal validity of the experiment concerns the question whether the effect is caused by independent variables, or by other factors. A natural variation in human performance, as well as experience, presented in Table 1, is a threat. However, programmers were at the same experience level E4, and the relationships between programmer experience and almost all design quality metrics were weak and non-significant. Concerning the *internal* validity, the risk of compensatory rivalry, or demoralization of subjects receiving less desirable treatments must be considered. Programmers using the classical method (i.e. object-oriented programming) may do their very best to show that the old method is competitive. On the other hand, subjects receiving less desirable treatments may not perform as well as they generally do. However, the subjects were informed that the goal of the study was to measure different programming approaches, not the subjects' skills. Possible diffusion, or imitation of treatments were under control of the authors.

The mono-operation bias is a *construct* validity threat, as the study was conducted on a single requirements set. Using a single type of measures is a mono-method bias threat. To minimize this threat, different measures (e.g. package as well as class level measures) were used in the study. Interaction of different treatments is limited, due to the fact that the subjects were involved in only one study. Other threats to construct validity are social threats (e.g. hypothesis guessing and experimenter's expectancies). As neither the subjects nor the experimenters have any interest in favour of one approach or another, we do not expect it to be a large threat.

The main threat to the *external* validity is related to the fact that subject population may not be representative to the software developers population. This is not a major issue as long as we are interested in the evaluating the use of a technique by novice or nonexpert software engineers [30]. Moreover, Kitchenham et al.[30] state that students are the next generation of software professionals and thus, are relatively close to the population of interest. Actually, all the subjects were classified as E4 according to Höst et al. [22]. The programmers' experience, presented in Table 1, is typical of young programmers, with solid software engineering academic background and some industrial experience. Tichy argues why it is acceptable to use students as subjects [31]. Some indications on the similarities between student subjects and professionals are also given by Höst et al. [32]. The threats to external validity were further reduced by using standard development tools e.g. SVN, Eclipse or AspectJ.

The validity of the results must be considered within the context of the limitations discussed in this section. The study can benefit from several improvements before replication is attempted. The most significant one is conducting a study with a sample of size large enough to guarantee a high-power design to establish evidence-based recommendations for the impact of aspect-oriented programming on software design quality and development efficiency.

7 Conclusions

It was not possible to apply statistical tests to analyse the impact of aspect-oriented programming on software development efficiency metrics (related to the goal 2) due to limited number of subjects. However, execution of statistical tests was possible for internal metrics (computed for each class, aspect or package in the system) related to software design quality.

It turned out that aspect-oriented programming approach did not significantly affect software design quality metrics i.e. Dn, WOM, CBM, RFM, LCO, $NCLOC$ and $NCLOC2$. It means that the impact of aspect-oriented programming on software design modularity and size (related to the goal 1) was not confirmed. In fact, the impact of aspect-oriented programming on class-level software quality metrics (WOM, CBM, RFM, LCO) as well as on $NCLOC$ and $NCLOC2$ was extremely weak, as suggested by the obtained effect size (r) values, see Section 5.2. A bit different situation is in the case of package level quality indicator i.e. Dn. The effect size value is still rather small, but a bit higher (see Section 5.2). It may suggest the need of further investigation of the hypothesis that aspect-oriented programming has the impact on higher level software quality indicators.

As pointed out in Section 5.2, there are two plausible explanations of the obtained results. First, the software development skill may, to a certain extent, compensate for the lack of useful features of aspect-oriented programming (e.g. modularised implementation of crosscutting concerns). Second, the AspectJ exhibits some limitations as a Java language extension and therefore the effect of aspect-oriented programming on software design quality may not be significant. AspectJ draws on an asymmetric approach i.e. there is a base body of object-oriented Java code that is then augmented with aspects. Therefore, there is an important difference between the base and the aspects, so that an aspect cannot serve as the base of another composition (in contrast to languages based on a symmetric approach). Moreover, AspectJ as a Java language extension does not force the use of aspects. In fact, the ratio of aspects to plain Java classes and interfaces in AO1 system was 13%.

Future studies can build on this study and its key takeaways in different manners. Aspect-oriented programming mechanisms and tools appeared to be mature enough to build system in an alternative manner to the pure object-oriented approach. The impact of aspect-oriented programming on software development efficiency was not confirmed. Future studies can benefit from assuring a sample of size large enough to guarantee a high-power design to establish evidence-based recommendations for the impact of aspect-oriented programming on software development efficiency. The impact of aspect-oriented programming on software design quality was not confirmed either. However, since internal metrics are computed for each class, aspect or package in the system, the number of subjects needed for the reliable execution of statistical tests may be seriously reduced, assuming large enough number of classes or aspects are considered. Future studies concerning software design quality may take advantage of this idea.

Acknowledgements

We thank Adam Piechowiak for his invaluable help by means of ActivitySensor Eclipse plugin development and all subjects for their engagement. We also thank the anonymous reviewers for their careful review of the manuscript, valuable comments and suggestions. This work has been financially supported by the Ministry of Education and Science, as a research grant 3 T11C 061 30 (years 2006-2007).

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J.: 'Aspect-Oriented Programming', Proc. European Conf. Object-Oriented Programming (ECOOP 1997), vol. 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, June 1997, pp. 220–242
2. Chidamber, S. R., and Kemerer, C. F.: 'A Metrics Suite for Object Oriented Design', *IEEE Trans. Softw. Eng.*, 1994, **20**, (6), pp. 476–493
3. Martin, R. C.: 'OO Design Quality Metrics: An Analysis of Dependencies', http://www.objectmentor.com/resources/articles/oodmetrc.pdf, accessed September 2006
4. George, B., and Williams, L. A.: 'An Initial Investigation of Test Driven Development in Industry', Proc. ACM Symposium on Applied Compupting (SAC 2003), Melbourne, USA, March 2003, pp. 1135–1139
5. George, B., and Williams, L. A.: 'A structured experiment of test-driven development', *Inf. Softw. Tech.*, 2004, **46**, (5), pp. 337–342
6. Madeyski, L.: 'Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality', in Zieliński, K., and Szmuc, T. (Ed.): 'Software Engineering: Evolution and Emerging Technologies', vol. 130 of *Frontiers in Artificial Intelligence and Applications*, (IOS Press, 2005), pp. 113–123
7. Ceccato, M., and Tonella, P.: 'Measuring the Effects of Software Aspectization', Proc. Workshop on Aspect Reverse Engineering (WARE 2004), Delft, The Netherlands, November 2004 (Cd-rom)
8. Aopmetrics project, http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.AOPMetrics, accessed September 2006

9. Subramanyam, R., and Krishnan, M. S.: 'Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects', *IEEE Trans. Softw. Eng.*, 2003, **29**, (4), pp. 297–310
10. Briand, L. C., Wüst, J., Ikonomovski, S. V., and Lounis, H.: 'Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study', Proc. Int. Conf. on Software Engineering (ICSE 1999), Los Alamitos, USA, May 1999, pp. 345–354
11. Emam, K. E., Melo, W. L., and Machado, J. C.: 'The Prediction of Faulty Classes Using Object-Oriented Design Metrics', *J. Syst. Softw.*, 2001, **56**, (1), pp. 63–75
12. Basili, V. R., Caldiera, G., and Rombach, H. D.: 'The Goal Question Metric Approach', in Marciniak J.J. (Ed.): Encyclopedia of Software Engineering, (Wiley, 1994), pp. 528–532
13. Garcia, A. F., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. J. P.de, and Staa, A.von: 'Modularizing Design Patterns with Aspects: A Quantitative Study', in Rashid, A., and Aksit, M. (Ed.): T. Aspect-Oriented Software Development I, 2006, vol. 3880 of *Lecture Notes in Computer Science*, pp. 36–74
14. Kersten, M., and Murphy, G. C.: 'Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999), New York, USA, November 1999, pp. 340–352
15. Walker, R. J., Baniassad, E. L. A., and Murphy, G. C.: 'An Initial Assessment of Aspect-oriented Programming', Proc. Int. Conf. on Software Engineering (ICSE 1999), Los Alamitos, USA, May 1999, pp. 120–130
16. Soares, S., Laureano, E., and Borba, P.: 'Implementing Distribution and Persistence Aspects with AspectJ', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), New York, USA, November 2002, pp. 174–190
17. Garcia, A. F., Sant'Anna, C., Chavez, C., Silva, V. T.da, Lucena, C. J. P.de, and Staa, A.von: 'Separation of Concerns in Multi-agent Systems: An Empirical Study', Proc. Int. Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003), vol. 2940 of *Lecture Notes in Computer Science*, pp. 49–72
18. Tsang, S. L., Clarke, S., and Baniassad, E. L. A.: 'An Evaluation of Aspect-Oriented Programming for Java-Based Real-Time Systems Development', Proc. Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), Vienna, Austria, May 2004, pp. 291–300
19. Hannemann, J., and Kiczales, G.: 'Design Pattern Implementation in Java and AspectJ', Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), New York, USA, November 2002, pp. 161–173
20. García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz-Sánchez, F., Piattini, M., and Genero, M.: 'Towards a consistent terminology for software measurement', *Inf. Softw. Tech.*, 2006, **48**, (8), pp. 631–644

21. Tonella, P., and Ceccato, M.: 'Refactoring the aspectizable interfaces: An empirical assessment.', *IEEE Trans. Softw. Eng.*, 2005, **31**, (10), pp. 819–832
22. Höst, M., , Wohlin, C., and Thelin, T.: 'Experimental Context Classification: Incentives and Experience of Subjects', Proc. Int. Conf. on Software Engineering (ICSE 2005), New York, USA, May 2005, pp. 470–478
23. Madeyski, L.: 'The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment', Proc. International Conf. Product Focused Software Process Improvement (PROFES 2006), vol. 4034 of *Lecture Notes in Computer Science*, Amsterdam, The Netherlands, June 2006, pp. 278–289
24. ActivitySensor project, http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.ActivitySensor, accessed March 2007
25. Cornett, S., 'Code Coverage Analysis', http://www.bullseye.com/coverage.html, accessed September 2006
26. Husted, T., and Massol, V.: 'JUnit in Action' (Manning Publications, 2003)
27. Erdogmus, H., Morisio, M., and Torchiano, M.: 'On the Effectiveness of the Test-First Approach to Programming', *IEEE Trans. Softw. Eng.*, 2005, **31**, (3), pp. 226–237
28. Howell, D. C.: 'Statistical Methods for Psychology' (Duxbury, Belmont, CA, 2002)
29. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A.: 'Experimentation in Software Engineering: An Introduction' (Kluwer Academic Publishers, Norwell, MA, USA, 2000)
30. Kitchenham, B., Pfleeger, S. L., Pickard, L., Jones, P., Hoaglin, D. C., Emam, K. E., and Rosenberg, J.: 'Preliminary Guidelines for Empirical Research in Software Engineering', *IEEE Trans. Softw. Eng.*, 2002, **28**, (8), pp. 721–734
31. Tichy, W. F.: 'Hints for Reviewing Empirical Work in Software Engineering', *Empir. Softw. Eng.*, 2000, **5**, (4), pp. 309–312
32. Höst, M., Regnell, B., and Wohlin, C.: 'Using Students as Subjects — A Comparative Study of Students and Professionals in Lead-Time Impact Assessment', *Empir. Softw. Eng.*, 2000, **5**, (3), pp. 201–214