How do Programmers Learn AOP?

An Exploratory Study of Recurring Mistakes

Péricles Alves, Alcemir Santos, Eduardo Figueiredo
Computer Science Department
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
{periclesrafael, alcemir, figueiredo}@dcc.ufmg.br

Fabiano Ferrari
Computing Department
Federal University of São Carlos (UFSCar)
São Carlos, Brazil
fabiano@dc.ufscar.br

Abstract — **Aspect-Oriented Programming (AOP) is a maturing technique that requires a good comprehension of which types of mistakes programmers, novice or experts, make during the development of applications. Unfortunately, the lack of such knowledge seems to represent one of the reasons for the cautious adoption of AOP in real software development projects. This paper reports the results of an experiment whose main goal is to observe and analyze mistakes made by students learning AOP. The experiment consists of a task that requires the aspectization of two crosscutting concerns in two object-oriented applications. Three rounds of the experiment in academic environment provided us with the data of 36 AOP applications upon which we develop our analysis. Mistakes are categorized and correlated with the developers' expertise in object-oriented programming (OOP) and their professional backgrounds. The results suggest that (i) in general, the more experience developers have the fewer mistakes they make, (ii) not always programmers working in pairs perform better than individually, and (iii) some specific kinds of mistakes are more frequently made by experienced older programmers.**

Keywords – Programming Mistakes, AOP, learning.

I. INTRODUCTION

Aspect-Oriented Programming (AOP) [12] is a maturing technique to enhance software modularity and reuse through the separation of crosscutting concerns into new modular units called aspects. However, to be widely adopted in practice, this technique requires a good comprehension of which mistakes programmers make during the development of AOP applications. It is known [17, 18] that novice programmers make several kinds of mistakes when learning how to program.

Novice programmers need special guidance to motivate and help them migrating from the current state-of-the-practice programming paradigm to AOP. For instance, these programmers can improve their performance when tools alert them to the existence of specific error-prone situations in the program at hand. Unfortunately, we still lack of a complete understanding of how programmers deal with the AOP mechanisms during maintenance tasks and how they make mistakes using these mechanisms. Such knowledge may represent one of the reasons for the cautious adoption of AOP in real software development projects.

A number of studies [1, 2, 6, 8, 19] have been performed aiming to identify faulty scenarios, bug patterns and fault taxonomies for the AOP paradigm. Such studies often take into account complex systems with numerous crosscutting concerns developed by experienced programmers in AOP. However, in such complicated scenarios, it is difficult to grasp the actual reasons which hamper the learning of basic AOP concepts, such as pointcut and advice.

This paper presents an experiment aiming to investigate the types of mistakes made by novice programmers learning AOP. Participants of this experiment have to factor out to aspects two existing crosscutting concerns in two OO applications. We performed the experiment in 3 rounds with 36 groups of students and junior professionals in software development. After refactoring the applications, participants delivered their AOP implementations of the concern and, then, we analyzed the types of recurring mistakes they made. Finally, we also investigate the possible correlation between such mistakes and the participants' backgrounds.

In this context, the contributions of this paper are threefold. First, we present in Section II the settings of this study which may serve as guidelines for further experiment replications with similar purposes. Second, based on the study settings, we perform an experiment to identify types of AOP-related mistakes recurrently made by novice programmers. A classification of these mistakes is presented in Section III. Finally, we analyze the data of the experiment and discuss our main results in Section IV. Matching the common sense, our results indicate that, in general, the more experience developers have the fewer mistakes they make. Interestingly, however, we found out that some kinds of mistakes, such as leaving concern code in the base, are more often made by experienced older programmers. Section V discusses related work and threats to the study validity. Section VI concludes this paper and points out directions for future research work.

II. STUDY SETTING AND GOALS

This section presents the goals of our study and the evaluated hypotheses (Section II.A). It also characterizes the study in terms of its participants' background (Section II.B) and the target applications (Section II.C).

A. *Study Goals and Research Hypothesis*

This study aims at investigating the types of mistakes made by students and junior professionals in AOP. In particular, its main goal is to identify and classify the most recurring categories of mistakes made by programmers learning this programming technique. The study relies on the aspectization of crosscutting concerns using the AspectJ language [13]. The target applications consist in two small-size Java applications.

To achieve the goal of this study, we analyze whether and how the programmer skills impact on the mistakes made by them. In this context, we define the null and alternative hypotheses as follows.

H_0 *The background of programmers learning AOP impacts on the mistakes they make.*

H_1 *The background of programmers learning AOP does not impact on the mistakes they make.*

To evaluate these hypotheses, we first investigate what are the categories of AOP-specific mistakes the programmers make. We then classify the participants according to three criteria: OOP-specific background, work experience, and age group. In total, 47 undergraduate students and junior professionals with different background levels took part in this experiment.

B. Participants' Backgrounds

The study was performed in three rounds with 47 participants. Prior to the experiment, all participants were already familiar with at least one programming language. All participants also attended a 2-hour training session in which they were presented the basic concepts of AOP and the AspectJ language.

In the first and second rounds, 13 groups of undergraduate students and 14 junior professionals, respectively, factored out to aspects a crosscutting concern of an OO ATM application (Section II.C). The participants worked in pairs in the first round and individually in the second one. In the third round, the experiment was replicated with 9 groups of undergraduate students. At this time, we relied on a different crosscutting concern and application, namely Chess (Section II.C), to achieve greater generality. In two rounds (the first and third ones), we decided to organize the participants in groups of two members. The goal was to verify possible variations in the concern aspectization when performed by multiple participants working together. From now on, we call the groups and individuals as *subjects of* the experiment.

With respect to the participants' backgrounds, each participant filled in a questionnaire asking for three pieces of information: their age, their level of professional experience, and their level of knowledge in OOP technology. In the second question, participants should indicate for how long they have worked with software development by choosing one of the following options: (i) never worked, (ii) worked less than six months, (iii) worked between six months and one year, (iv) worked between one and three years, and (v) worked for more than three years. Finally, the options for level of knowledge in OOP are: (i) never programmed in OOP, (ii) programmed a few times, (iii) programmed several times, and (iv) very familiar with OOP.

C. The Target Crosscutting Concerns and Applications

This study relies on two small Java applications named ATM and Chess. Participants were asked to aspectize one crosscutting concern of each application: *Logging* and *ErrorMessages*, respectively. Table I presents data about the sizes of the applications and their respective concerns. As presented in this table, both applications share similar sizes and characteristics. For instance, the Number of Modules (NOM) is 12 and 13 in ATM and Chess, respectively. Their respective numbers of Lines of Code (LOC) are 606 and 1011. These applications were carefully chosen to allow the experiment to be completed in a 1.5-hour class.

TABLE I. SIZE METRICS OF THE APPLICATIONS AND CONCERNS

	Application Size		***Concern Size***	
	NOM	***LOC***	***CDC***	***LOCC***
ATM / Logging	12	606	4	19
Chess / ErrorMessages	13	1011	8	36

Participants of the first and second rounds of our experiment had to aspectize *Logging* in ATM while participants of the third round had to aspectize *ErrorMessages* in Chess. Table I shows that the *Logging* concern is scattered across 4 classes (Concern Diffusion over Components - CDC) [10, 16]. It requires 19 lines of code (Lines of Concern Code - LOCC) [7] in the OO ATM implementation. On the other hand, *ErrorMessages* is spread across 8 classes (CDC) and 36 lines of code (LOCC).

Figure 1 presents a simplified class diagram of the OO ATM application. This application simulates the activities of a conventional cash machine, such as withdrawing money, making a deposit, and issuing the account balance. Classes, methods and attributes related to the *Logging* concern appear shadowed in gray. This concern records every action performed by the user of the ATM.

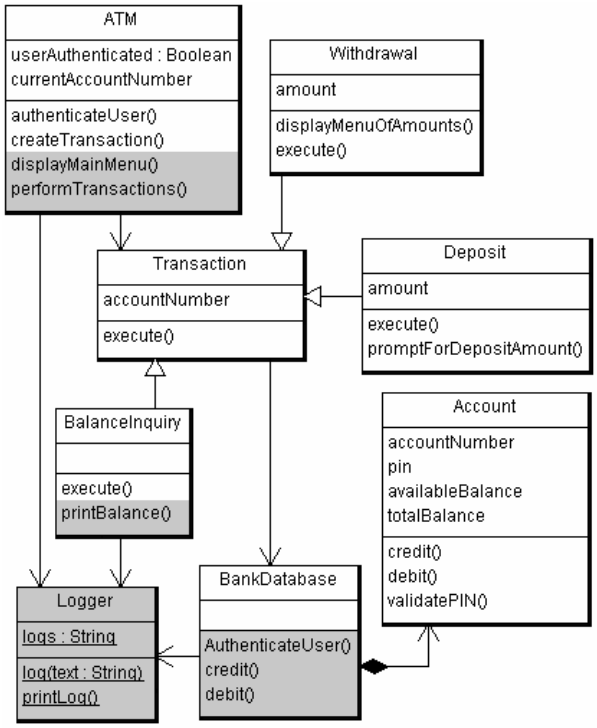

Figure 1. A simplified class diagram of the OO ATM application.

Figure 2 presents a simplified class diagram of the OO Chess application. This application implements a chess game with a Graphical User Interface. Methods and attributes which realize *ErrorMessages* are shadowed in Figure 2. This concern represents error messages which are shown to the players when they try to move a piece in a wrong way, i.e. which does not follow the chess' rules.

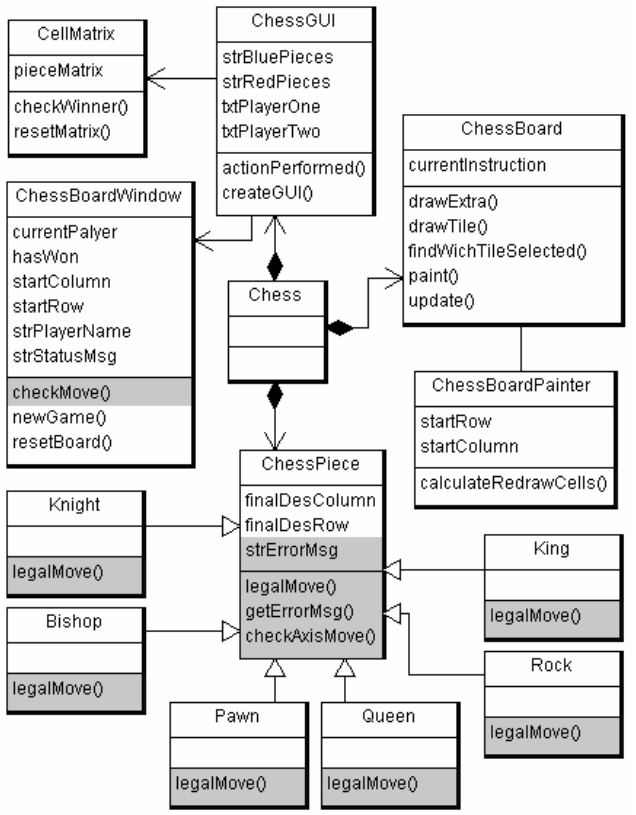

Figure 2. A simplified class diagram of the OO Chess application.

III. CLASSIFICATION OF RECURRING MISTAKES

A number of studies of AOP-specific faults and bug patterns are available in the literature [1, 2, 6, 8, 9, 19]. These studies range from the most basic concepts of AOP to advanced constructions of AspectJ-like languages, such as intertype declarations and other declare-like expressions. As this study was conducted only with novice programmers in AOP, the classification we propose in this section focuses on the mistakes they made when they use the two most basic features of AOP languages: advices and pointcuts.

We are using the term *mistakes* to refer to the errors made by programmers when they refactor an OO application. Therefore, our classification partially overlaps previously defined fault taxonomies [2, 9] and adds new categories, such as *compilation errors or warning* and *incomplete refactoring*. As far as we are concerned, these categories have not appeared in previous classifications. The defined categories are presented in the sequence. Each of them brings together an example extracted from the two applications described in the previous section.

A. *Advices-Related Mistakes*

A.L. – *Incorrect implementation logic*. This mistake occurs when part of the base code is factored out to an advice. However, the advice produces an unexpected behavior. For instance, in the original ATM application (i.e. its OO version), a report with the log of all operations performed by the user is only displayed when the user option is 0 (zero). This option is a parameter of the `performTransactions` method in the `ATM` class. Once this behavior is factored out to an advice, due to a mistake during the aspectization, whenever the `performTransactions` method is invoked, the log report is displayed on the screen.

A.I. – *Incorrect advice type*. In this mistake category, the programmer uses an incorrect type of advice to implement part of the concern. For example, in the OO version of the ATM application, the logged message which records information about deposits is performed after the `credit` method of the `BankDatabase` class is concluded. That is, a call to `log` in the `Logger` class (Figure 1) appear at the end of the `credit` method. When *Logging* is refactored to an aspect, the advice should be implemented as *after*. However, some subjects of this study wrongly use a *before* advice instead.

B. *Mistakes Related to Compilation*

C.E. – *Compilation error or warning*. This mistake refers to either a syntactic fault or a potential fault signaled by the AspectJ compiler through an error or warning message[1]. For example, in the OO version of the Chess application, the `King` class contains a method named `legalMove` (Figure 2). Due to a refactoring mistake, a pointcut that should select join points when the `legalMove` method is invoked was written as `call(public boolean legalMove(..))`. That is, the pointcut did not specify the method's class neither uses any wildcard. Consequently, a warning message notifies the user that no join point matched this specific pointcut.

C. *Other Mistakes*

R.D. – *Duplicated crosscutting code*. In this mistake, part of the code that implements a crosscutting concern appears replicated in both aspects and the base code. For instance, in the Chess application (Figure 2), the `getErrorMsg` method of the `ChessPiece` class implements part of the *ErrorMessages* concern. When the concern is aspectized, the `getErrorMsg` method is moved to an aspect. However, due to a mistake during the refactoring, the method is also left in the original class (i.e., `ChessPiece`) class and, so, resulting in duplicated code.

R.I. – *Incomplete refactoring*. This mistake occurs when the developer fails to refactor part of the crosscutting concern. Consequently, part of concern still remains in the base code. Example: In the OO version of ATM application, the `performTransactions` method of the `ATM` class (Figure 1)

[1] *Note that some times syntactic problems in AspectJ code are not promptly signaled by the compiler; programmers can only notice such problems when they perform a full weaving of the system. Considering that the subjects had to complete the planned tasks in a short period of time, several implementations could not be double-checked with respect to these types of faults. Section IV presents and discusses the observed results.*

implements part of the *Logging* concern. When the programmer refactored the concern, neither did s/he transfer the behavior of this method to the aspect nor excluded it from the `ATM` class.

IV. Data Collection and Analysis

This section presents and analyzes the results of our experiment in terms of each piece of background information: OOP experience (Section IV.B), work experience (Section IV.C), and age group (Section IV.D). Section IV.A presents the overall data for all subjects regardless of their experience.

A. Overall Number of Mistakes

We collected 36 different AO implementations in this study (Section II). The mistakes made by each subject when refactoring a concern were collected and classified according to the categories defined in Section III. Figure 3 presents the percentage of subjects that made mistakes in each category per round. Note that, we do not specify the number of mistakes per category in our analysis. That is, if a subject made several mistakes in the same category, we count them just once. Therefore, a column reaching 100% in Figure 3 (e.g., R.I for the 2nd and 3rd rounds) means that all subjects of the round made at least one mistake in that specific category.

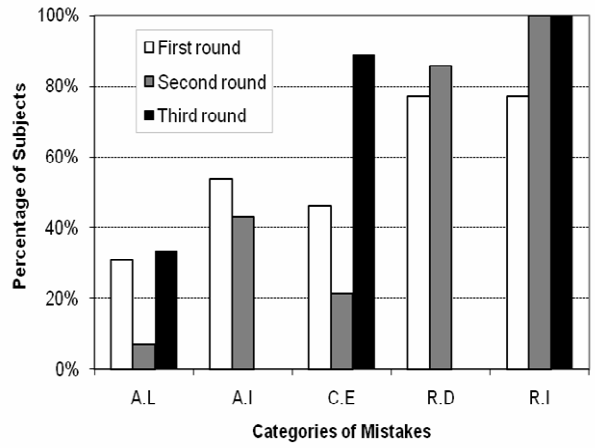

Figure 3. Percentage of subjects that made mistakes per category.

An analysis of data in Figure 3 indicates interesting behavior across the three rounds. For instance, although subjects of the first and second rounds aspectized the same application (ATM), undergraduate students working in pairs made more mistakes related to advice and compilation errors (A.I, A.L, and C.E) than individual graduate participants. This result suggests that undergrads cannot easily understand how advice in AspectJ works. Furthermore, due to their lack of experience, they do not pay attention – or do not observe – the language compilation error and warning messages. On the other hand, graduated participants performed worse than undergrads with respect to refactoring-related mistakes. The result may be due to pair programming performed in the first round. In fact, refactoring and pair programming are key practices of agile methods, such as Extreme Programming (XP) [4]. Therefore, our results may confirm that these practices are also valuable in the AOP context.

Subject of the third round made several mistakes of some categories, such C.E and R.I, and barely any mistake in others. We verified that this situation occurs due to the nature of the *ErrorMessages* concern. For instance, this concern only requires one type of advice (after advice) in its implementation. Therefore, subjects almost did not have the "opportunity" to make mistakes of incorrect advice type (A.I), for instance.

B. Level of OOP Experience

We divided the subjects into two groups according to the levels of experience in OOP of their members: (i) subjects with little or no experience and (ii) subjects with moderate to high experience. In case a subject represents a pair of students, we considered it into the second group if at least one student has moderate to high experience in OOP. The first group (*Beginners in OOP*) has 12 subjects and the second one (*Experts in OOP*) has 24 subjects.

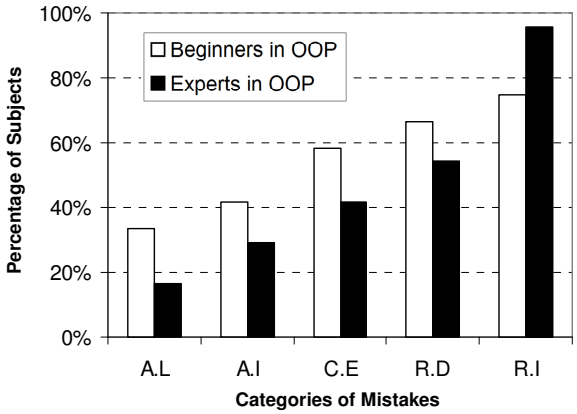

Figure 4. OOP experience and mistakes made by subjects.

Figure 4 presents the percentage of subjects which made mistake in each category. As expected, the most experienced in OOP the subjects are, the less mistakes they make. This is easily observed in Figure 4, since experienced subjects usually make fewer mistakes than beginners in all categories, except Incomplete Refactoring (R.I). The explanation to this result could be the fact that experts in OOP tend to heavily use the OOP abstractions which they are comfortable with. That is, they do not factor out to aspects several pieces of code in favor of using their best knowledge in OOP. The good OOP solution is considered a mistake in this paper because (i) we explicitly asked subjects to use aspects to modularize the specific crosscutting concerns and (ii) there is no way to modularize the crosscutting concern using only OOP.

C. Types of Mistakes per Professional Experience

In this analysis, we split subjects into two groups according to their professional experience in software development: (i) subjects with little or no work experience and (ii) subject with some work experience. In case of pair-wise subjects, we consider little work experience when both participants has less than one year. Otherwise, the subject is classified in the second group. The first group, *less than one year experience*, has 21 subjects and the second group, *more than one year experience*, has 15 subjects.

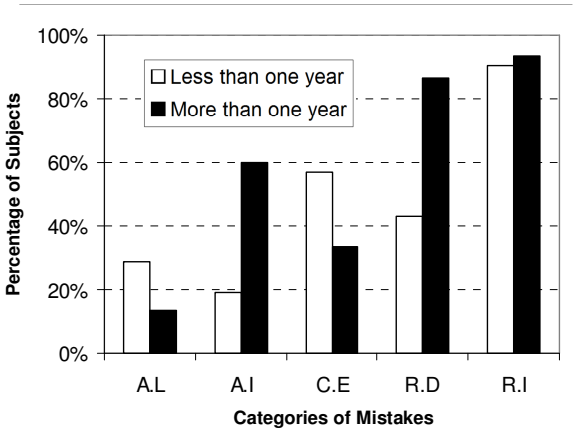

Figure 5. Work experience and mistakes made by subjects.

Figure 5 presents the overall percentage of mistakes made by subjects taking their work experience into account. Interestingly, the results presented in Figure 5 do not match the common sense that the more experience developers have, the fewer mistake they make. In fact, these results suggest that experienced developers make more mistakes than beginners in at least two categories: Incorrect Advice Type (A.I) and Duplicated Crosscutting Code (R.D). To further investigate this finding, we verified that all subjects of the third replication were classified in this group (i.e., less than one year experience). Furthermore, these subjects did not have the opportunity to make these two kinds of mistakes (A.I and R.D) due to the nature of the concern and application; as discussed in Section IV.A. If we consider just the first two rounds which rely on the same application (ATM), the difference becomes less significant (e.g., 75% vs. 87% for R.D). This smaller difference may be due to the fact that experienced developers worked alone (second round) while beginners worked in pairs (first round).

D. Types of Mistakes per Age Group

In Section IV.C we use the threshold of one year experience to split subjects aiming to balance the number of subjects in each groups (21 vs. 15). However, we recognize that a developer with one year experience cannot be compared to one with 10 years of experience. To mitigate this issue, we analyze the 15 subjects of the second group in Section IV.C by age group. Our hypothesis is that the older developers are, the more experience they have. As a result, we create two groups: (i) developers which are 25 years old or younger and (ii) developers older than 25. The first group has 7 subjects while the second group has 8 subjects.

Figure 6 shows the results per age group of developers with more than 1 year experience in software development. Data of this chart indicate that younger developers make more mistakes than older ones in the first three categories (A.L, A.I, and C.E). However, older developers make more refactoring-related mistakes (R.D and R.I). Even if we consider only subjects working individually (second round), similar trend is observed. Therefore, our conclusion in this case is that older developers more frequently create duplicated crosscutting code (R.D) or forget code which is supposed to be factored out (R.I). Since this observation is based only on 15 subjects, it is not conclusive. Further studies, such as controlled experiments, should be performed to confirm or refute our preliminary findings.

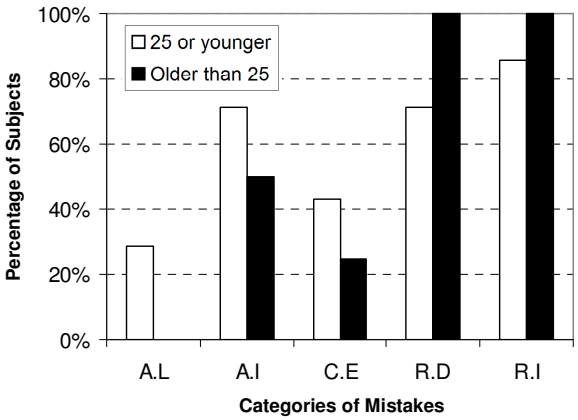

Figure 6. Age groups of developers with more tha 1 year work experience.

V. RELATED WORK AND STUDY LIMITATIONS

We identified two research lines that are closely related to the study presented in this paper: (1) research that characterizes AOP-specific faults and (2) research that investigates how these faults occur in AO programs.

In regard to the categorization of faults in AO programs, as reported by Ferrari *et al.* [9], one can find a set of candidate fault taxonomies proposed in the literature, each one emphasizing different subsets of AOP-specific properties [1, 2, 6, 8, 19]. Ferrari *et al.* [9] analyzed the existing taxonomies and identified four main categories of faults that can be found in AO software. In this paper, our fault categorization relies on a different point of view: it focuses on mistakes programmers make during the refactoring of crosscutting concerns. The types of mistakes proposed herein do not necessarily result in faults in the application code (e.g. the *incomplete refactoring* category).

In regard to how faults are introduced into AO programs, recent studies [11, 15] have investigated mistakes in the projection of concerns on code – a key task in concern refactoring. For instance, Figueiredo *et al.* [11] noticed that programmers are conservative when projecting concerns, i.e. they make omissions of concern-to-elements assignments. Such false negatives may lead to mistakes like *incomplete refactoring* or even *incorrect implementation logic* herein described. Burrows at al. [5] investigated the introduction of faults during the evolution of a small AO system. Academic subjects performed a series of maintenance tasks in and, in a subsequent phase, the resulting implementations were tested and faults were reported. Differently from our study, Burrows *et al.* focused on analyzing the correlation between the number of faults and concern and code churn metrics.

One key limitation of this study concerns the size and representativeness of the target applications and the level of subjects' expertise. The target applications are indeed small, in turn limiting the generalization of the results. However, larger

systems could be an impediment for the execution of such an experiment in a predefined – and short – period of time, specially due to the time required to understand the system and perform changes on it. With respect to the level of expertise of subjects, one may argue that assigning students the tasks planned for an experiment does not reflect the state of the practice given that they might not have the required knowledge to do so. However, Basili *et al*. [3] and Kitchenham *et al*. [14] argue that students play an important role in experimentation in software engineering. In particular, they can help researchers to obtain preliminary, however still important evidence that should be further investigated in more robust experiments. Nevertheless, we are currently running additional rounds of the experiment in order to increase our data set and perform more solid statistical analysis.

VI. Conclusion and Future Work

Understanding how programmers deal with the AOP mechanisms during maintenance tasks and how they make mistakes using these mechanisms represents an important contribution towards the adoption of AOP in practice. This understanding may help, for example, in the development of tools that enhance system design and evaluation capabilities. With a growing body of knowledge with respect to common mistakes in AOP, researchers and practitioners can employ efforts to develop tools that automatically check the occurrence of imprecise refactoring or analyze specific parts of the code with the aim of localizing faults. When novice programmers are taken into account, understanding how they make mistakes can also improve the quality of programming education in general [17, 18], particularly in AOP.

This paper contributed in this sense by reporting on a series of small-sized experiments whose main goal was to characterize mistakes made by novice and junior programmers in typical crosscutting concern refactorings. Our results indicate recurring mistakes made by programmers with specific backgrounds and corroborate our research hypothesis (Section II.A), that the background of programmers learning AOP impacts the mistakes they make. For instance, pair programming may help programmers avoid some mistakes, such as incomplete refactoring, but does not help to avoid advice-related mistakes. The results of this study may be used for several purposes. For instance, they may help in the development and improvement of new AOP languages and tools. That is, developers of AOP languages and tools should enhance the error-prone mechanisms, constructs, and abstractions of current AOP approaches.

Future research can rely on our study settings to perform new replications of the experiment in order to enlarge our dataset. In fact, we have plans to replicate ourselves this study using different applications and subjects. This shall allow us to perform more comprehensive analysis with statistical significance. Moreover, based on the identified categories of mistakes, a supporting tool can be developed to advise programmers about typical AOP-related mistakes.

Acknowledgments

This work was supported by PIBITI/CNPq, FAPEMIG (grant APQ-02932-10) and FAPESP.

References

[1] R. T. Alexander, J. M. Bieman, and A. A. Andrews, “Towards the systematic testing of aspect-oriented programs,” Dept. of Comp, Science, Colorado State Univ., Report CS-04-105, 2004.

[2] J. Baekken and R. Alexander. "A Candidate Fault Model for AspectJ Pointcuts". In proceedings of the 17th Int’l Symposium on Software Reliability Engineering (ISSRE), pp. 169-178. Raleigh, USA, 2006.

[3] V. Basili, F. Shull, and F. Lanubile, “Building knowledge through families of experiments”. IEEE Transactions on Software Engineering. vol. 25, issue 4, pp. 456–473. 1999.

[4] K. Beck. “Extreme Programming Explained: Embrace Change”. Addison-Wesley. 1999.

[5] R. Burrows, F. Taïani, A. Garcia and F. Ferrari. “Reasoning about Faults in Aspect-Oriented Programs: A Metrics-based Evaluation”. To be presented at the 19th IEEE International Conference on Program Comprehension (ICPC), Kingston, Canada, June 2011.

[6] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa and 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), pp. 207-234. Paphos, Cyprus, July 2008.

[7] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan and A. Aho “Do Crosscutting Concerns Cause Defects?”, IEEE Trans. on Software Engineering (TSE), 34, 4, p. 497-515. 2008.

[8] F. Ferrari, J. Maldonado and A. Rashid. "Mutation Testing for Aspect-Oriented Programs". In proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST), pp. 52-61. Lillehammer, Norway, April 2008.

[9] F. Ferrari, R. Burrows, A. Garcia and J. Maldonado. "Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap Between Theory and Practice". In proceedings of the 24th Brazilian Symposium on Software Engineering (SBES), pp. 50-59. Salvador, Bahia, 2010.

[10] E. Figueiredo, C. Sant’Anna, A. Garcia, T. Bartolomei, W. Cazzola, and A. Marchetto. “On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework”. In proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR), pp. 183-192. Athens, Greece, 2008.

[11] E. Figueiredo, A. Garcia, M. Maia, G. Ferreira, C. Nunes, and J. Whittle. "On the Impact of Crosscutting Concern Projection on Code Measurement". In proc. of the 10th Int’l Conf. on Aspect-Oriented Software Development (AOSD), pp. 81-92. Porto de Galinhas, 2011.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. “Aspect-Oriented Programming”. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), vol. 1241, pp. 220-242. 1997.

[13] G. Kiczales, E. Hilsdale, j. Hugunin, M. Kersten, J. Palm and W. Griswold. "An overview of AspectJ". In proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), pp. 327-353. Budapest, Hungary, June 2001.

[14] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam and J. Rosenberg. “Preliminary guidelines for empirical research in software engineering”. IEEE Transactions on Software Engineering, vol. 28, issue 8, pp. 721–734. 2002.

[15] C. Nunes, A. Garcia, E. Figueiredo, and C. Lucena. "Revealing Mistakes on Concern Mapping Tasks: An Experimental Evaluation”. In proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 101-110. Oldenburg, Germany, 2011.

[16] C. Sant’Anna, A. Garcia and C. Chavez. “On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework”. In Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES), pp. 19-34. 2003.

[17] E. Soloway and K. Ehrlich. “Empirical Studies of Programming Knowledge”. IEEE Trans. on Softw. Eng. (TSE), 5, pp. 595-609. 1984.

[18] J. Spohrer and E. Soloway. “Novice mistakes: are the folk wisdoms correct?”. Communications of the ACM. vol. 29, issue 7. 1986.

[19] S. Zhang and J. Zhao. "On Identifying Bug Patterns in Aspect-Oriented Programs". In proc. of the 31st Int’l Computer Software and Applications Conference (COMPSAC), 431-438. Beijing, China, 2007.