Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Avoiding code pitfalls in Aspect-Oriented Programming

Adriano Santos [a,*], Péricles Alves [a], Eduardo Figueiredo [a], Fabiano Ferrari [b]

[a] Computer Science Department, Federal University of Minas Gerais, Belo Horizonte, Brazil
[b] Computing Department, Federal University of São Carlos, São Carlos, Brazil

ARTICLE INFO

Article history:
Received 15 March 2015
Received in revised form 1 December 2015
Accepted 2 December 2015
Available online xxxx

Keywords:
Aspect-Oriented Programming
Mistakes
Code pitfalls
Empirical study

ABSTRACT

Aspect-Oriented Programming (AOP) is a maturing technique that requires a good comprehension of which types of mistakes programmers 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 on the results of a series of experiments whose main goal is to analyze and catalogue code pitfalls that are likely to lead programmers to make mistakes in AOP refactoring. Each experiment consists of a task that requires the aspectization of a crosscutting concern in one object-oriented application. Eight rounds of the experiment provided us with data of 98 AOP implementations from four crosscutting concerns in four different applications. Each participant of the experiment produced one implementation. Based on the analysis of these implementations, we (i) document six categories of recurring mistakes made by programmers, (ii) correlate these mistakes with the programmer expertise in object-oriented programming, years of software development, and pair programming, and (iii) derive a catalogue of code pitfalls which are likely to lead programmers to make the documented mistakes. We apply significance tests in order to statistically evaluate our results. We also present a prototype tool to warn programmers of the code pitfalls during refactoring activities.

1. Introduction

Aspect-Oriented Programming (AOP) [25] is a software development technique that aims to improve software modularity through the separation of crosscutting concerns into modular units called aspects. A concern is any consideration that can impact on the design and maintenance of program modules [38]. It is well known that novice programmers need special guidance while learning how to program in a new programming language [41,42,44]. Therefore, to be widely adopted in practice, we need a good comprehension of the kinds of mistakes made by AOP programmers when learning this development technique and the situations that lead to these mistakes.

The IEEE Standard Glossary for Software Engineering [22] defines a *mistake* as "*a human action that produces an incorrect result*". In the context of this paper, a refactoring mistake occurs when a developer performs an inconsistent code refactoring, which may have as a consequence, the introduction of a fault into the application code. A *fault*, on the other hand, consists in an incorrect step, process, or data definition in a computer program [22]. In other words, a fault occurs when there is

* Corresponding author.
E-mail addresses: adrianolages@dcc.ufmg.br (A. Santos), periclesrafael@dcc.ufmg.br (P. Alves), figueiredo@dcc.ufmg.br (E. Figueiredo), fabiano@dc.ufscar.br (F. Ferrari).

http://dx.doi.org/10.1016/j.scico.2015.12.003
0167-6423/

a difference between the actually implemented software product and the product that is assumed to be correct. A mistake may lead to one or more faults being inserted into the software, although it may not necessarily do so.

Previous research [3,6,11,14] has investigated mistakes that are likely to be made by AOP programmers either to build systems from scratch or to refactor existing ones. Other studies have identified aspect-oriented code smells [15,28,31,35] and fault-prone scenarios [46]. However, these studies target at complex software systems developed by experienced programmers. In such complex scenarios, it is difficult to reveal the factors that hinder the learning of basic AOP concepts, such as pointcut and advice, by novice programmers. This situation gives us a lack of understanding of the scenarios that could lead novice programmers to make mistakes while refactoring existing code to AOP. We call these scenarios *code pitfalls*. In turn, code pitfalls may represent one of the reasons for the cautious adoption of AOP in real software development projects [26,33,36].

This paper extends our previous study [4] that investigated a preliminary set of recurring mistakes made by novice AOP programmers. Previously, we derived mistakes by running six rounds of an experiment with 80 fresh AOP programmers [4]. This paper extends our previous work by reporting two additional rounds of the same experiment (resulting in eight rounds in total) with a different application and 25 additional participants. Therefore, this paper relies on an extended dataset of 98 refactored code samples of four applications (Section 2). With the new replications, we identified additional instances of mistakes and code pitfalls.

In this paper, we not only further investigate and confirm our previous findings, but also seek to better understand the categories of common mistakes made by AOP programmers (Section 3). An interesting result found in our study is that more experienced programmers in OOP made more mistakes than novice programmers in some categories, such as incorrect advice type and duplicated crosscutting code. In addition, we also perform further statistical analysis of the data in this follow up study, for instance, to understand how skills of programmers impact on the mistakes they make (Section 4). We rely on a full factorial experiment design with application of one-way Analysis of Variance (ANOVA) [23,45]. Based on the new statistical analysis, this paper extends the catalogue of code pitfalls (Section 5); i.e., situations that appear to lead programmers to make the identified categories of recurring mistakes. Code pitfalls were observed by inspecting the original source code focusing on code fragments that were incorrectly or inconsistently refactored to AOP.

To support the automatic detection of the documented code pitfalls, we developed a prototype tool called ConcernReCS (Section 6). ConcernReCS is an Eclipse plug-in that aims to warn AOP programmers of situations that may lead to refactoring mistakes. It is based on static analysis of Java code and builds up on knowledge of our experiment results. Section 7 discusses some limitations of our study and Section 8 summarizes related work. Section 9 concludes this paper and points out directions for future work.

2. Experimental setup

This section presents the configurations of the experimental study we conducted. Section 2.1 presents and discusses the research questions that we aim to answer in this paper. Section 2.2 describes the background of all participants. Section 2.3 introduces the design of the target applications and the basic characteristics of the refactored crosscutting concerns. Section 2.4 explains the experimental tasks.

2.1. Research questions and hypotheses

This study aims at investigating the types of mistakes made by students and junior professionals when using AOP. Additionally, we target at uncovering and documenting code pitfalls that lead programmers to make these mistakes. Based on these aims, we formulate the following three research questions.

R1. What kinds of mistakes do junior AO programmers often make while refactoring crosscutting concerns?
R2. What are the code pitfalls in the source code that lead to these mistakes?
R3. Which skills should a programmer have to make fewer mistakes in AOP?

To answer R1, we first identify and classify the recurring categories of mistakes made by AOP programmers. Then, we document error-prone situations as a catalogue of code pitfalls to address R2. Finally, we analyze whether and how the programmer skills impact on the observed mistakes to answer R3. In this case, the common sense is that the background of programmers impacts on the number of mistakes they make. Therefore, we decomposed the last research question (RQ3) in the set of hypotheses below. To validate our hypotheses, we rely on a full factorial experiment design, including all possible combinations between the levels of the experiment factors. Statistical significance was tested by the analysis of variance (ANOVA) of experimental data.

H1.0: Level of experience in OOP does not impact the number of mistakes in AOP.
H1.1: Level of experience in OOP impacts the number of mistakes in AOP.

H2.0: Work experience does not impact on the number of mistakes in AOP.
H2.1: Work experience impacts on the number of mistakes in AOP.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

Table 1
Number of subjects in each round.

System	Round	Number of subjects	Grouping
ATM	First	11	Pairs
	Second	16	Individually
Chess	Third	15	Pairs
	Fourth	15	Individually
Telecom	Fifth	3	Pairs
	Sixth	19	Individually
Task Manager	Seventh	6	Pairs
	Eighth	13	Individually
Total number of subjects		98	

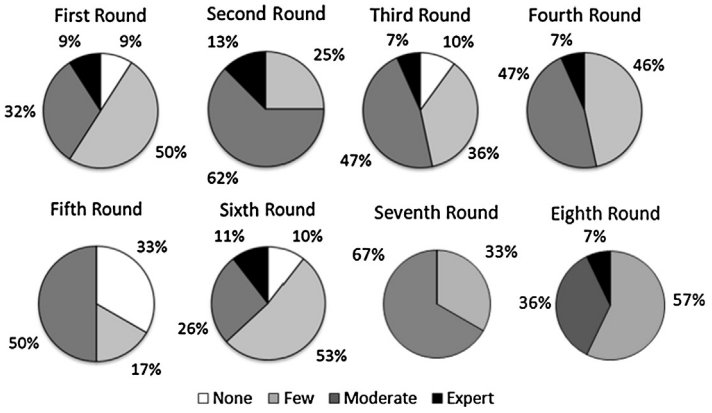

Fig. 1. OOP knowledge of the participants.

H3.0: Pair programming does not impact on the number of mistakes in AOP.
H3.1: Pair programming impacts on the number of mistakes in AOP.

2.2. Background of subjects

Participants of all replications were organized in groups of one or two members; each group we call a *subject* of the experiment. We organized some participants in pairs in order to investigate possible impact of pair programming on the mistakes made. Participants of four rounds worked in pairs, while participants of the other four rounds worked individually. Table 1 summarizes the number of subjects (i.e., groups of participants) and how they were organized in each round. Note that in the first, third, fifth, and seventh rounds, the numbers of participants are twice the number of subjects since each subject includes two participants. That is, the total number of participants is 133 divided into 98 subjects.

Table 1 also shows the application each subject worked with. That is, subjects of the first two rounds refactored a concern of an ATM application, subjects of the following two rounds refactored a Chess game, subjects of the fifth and sixth rounds refactored a telephony simulation system (Telecom), and the subjects of seventh and eighth rounds addressed a Task Manager system. We present the refactored concerns of each software system in Section 2.3.

The participants were asked to fill in a questionnaire with their background information. This questionnaire asked participants about their level of knowledge in Object-Oriented Programming (OOP) and their work experience in software development. For the former, participants had one of the following options to choose: (i) never programmed in OOP, (ii) programmed a few times, (iii) programmed several times, and (iv) I am very familiar with OOP. For the question concerning work experience, the options were: (i) never worked in software development industry, (ii) worked less than one year, (iii) worked between one and three years, and (iv) worked for more than three years. This questionnaire aims both to characterize the background of the study participants and to help us identifying some key factors that may impact on AOP-related mistakes [1].

Fig. 1 summarizes the answers of all participants per study round with respect to their knowledge in OOP. Based on answers of this questionnaire, we verified that, except in fifth round, almost all participants are familiar with basic OOP concepts. Moreover, when participants worked in pairs (odd rounds), at least one participant member in each subject is familiar with OOP. The exception is one subject in the fifth round where both participants claimed to have no experience in OOP (details can be found at the project website [1]).

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

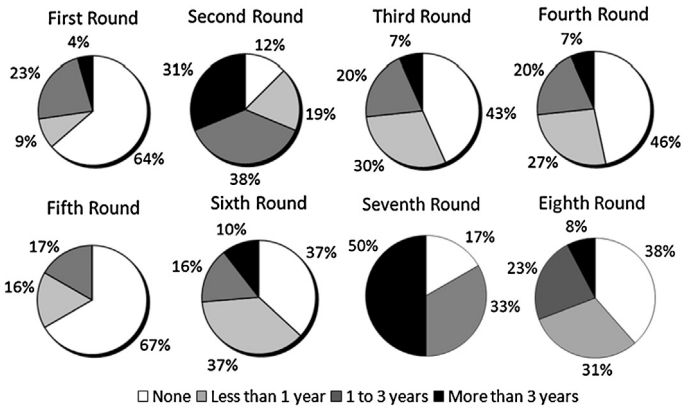

Fig. 2. Work experience of the participants.

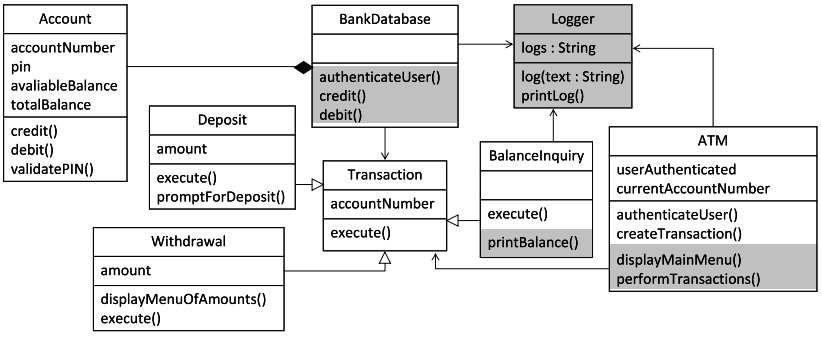

Fig. 3. Class Diagram of the ATM application.

Fig. 2 presents the answers of all participants with respect to their work experience. In this case, we observe that about 37% to 67% of the participants in all rounds, except for the second and seventh rounds, have never worked in a software development company. This result is somehow expected since most participants are undergraduate or post-graduate students. It is important to highlight that our study focuses on junior programmers (Section 2.1), and hence, the selected participants match our goals. Participants in the second and seventh rounds are professionals (or recently graduated people) taking an advanced Software Engineering course. This fact explains why these participants have more experience in software development.

2.3. Target applications and crosscutting concerns

Four small Java applications were chosen to be used in this study. The authors of this paper implemented only one of them, namely Task Manager. Subjects were asked to refactor one crosscutting concern of each application using the AspectJ language [24]. This section briefly presents the design of each application – namely ATM, Chess, Telecom, and Task Manager - and explains the refactored concerns.

ATM. The ATM application simulates three basic functionalities of an ordinary cash machine: show balance, deposit, and withdraw. Its implementation comprises about 600 lines of code (LOC) and exactly 12 modules (classes and interfaces). Subjects of the first two rounds had to refactor out to aspects the Logging concern in this application. This concern is responsible for recording all operations performed by an ATM user. It is implemented by 19 LOC diffused over 4 application modules. Fig. 3 shows a simplified Class Diagram of the application. In this figure, methods and attributes related to the Logging concern are shadowed in grey. For instance, there is code related to Logging in three methods - `authenticateUser()`, `credit()`, and `debit()` - in the `BankDatabase` class.

Chess. This application implements a chess game with a graphical user interface. Its implementation comprises about 1000 LOC in 13 modules. Subjects of the third and fourth rounds refactored the ErrorMessages concern of this application. This concern is responsible for displaying messages when a player tries to break the chess rules. For instance, if a player tries to move the king piece more than one row or column, the "*Illegal move.*" error message is shown on screen. This concern is implemented by 36 LOC spread over 8 classes. Fig. 4 presents a simplified Class Diagram of the Chess application with the ErrorMessages concern shadowed in grey. As we can observe in this figure, ErrorMessages is implemented by the `ChessBoardWindow` class, and by `ChessPiece` and its child classes.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

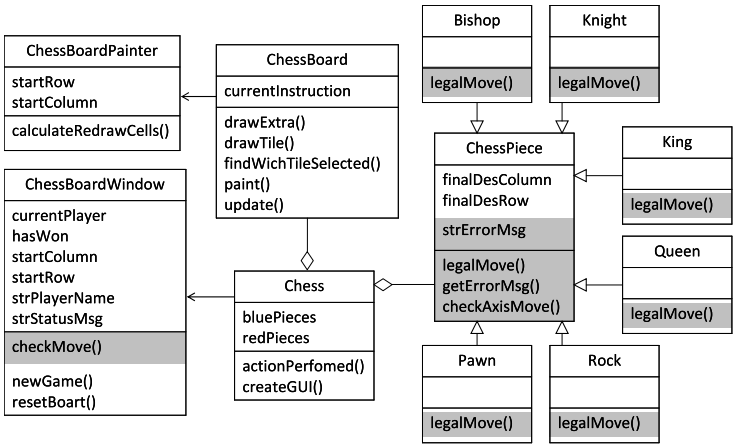

Fig. 4. Class Diagram of the Chess application.

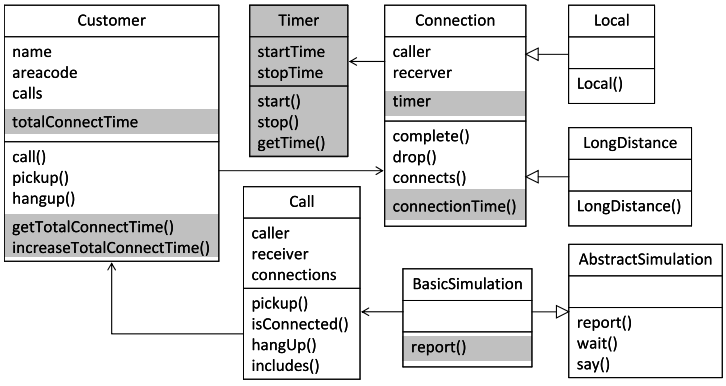

Fig. 5. Class Diagram of the Telecom application.

Telecom. It is a connection management system for phone calls which simulates a call between two or more customers. Its implementation comprises nearly 200 LOC and 8 classes. In fact, this system is an example from AJDT plugin [2] for Eclipse IDE, originally implemented using Java and AspectJ languages. We only used the Java implementation and asked each subject of the fifth and sixth rounds to refactor out to aspects the Timing concern. This concern is responsible for measuring the elapsed time of a given call. It is implemented in 30 LOC spread over 4 modules. Fig. 5 presents a simplified Class Diagram of the Telecom application with the Timing concern shadowed in grey. This concern crosscuts the `Timer`, `Customer`, `Connection`, and `BasicSimulation` classes.

Task Manager. This application implements functionalities related to task management, like TO-DOs. Task Manager has 209 lines of code implemented in 6 classes. In this application, the user can define tasks and schedule their deadlines. The Observer design pattern [21] is the concern intended to be refactored by subjects. Fig. 6 presents a simplified Class Diagram of Task Manager. It highlights in gray the methods that implement Observer. This concern is responsible for notifying all classes that are observing the tasks, when this object is modified. Observer spreads over 3 classes in the Task Manager application: `Task`, `TaskHistoryPanel`, and `TaskSelectorPanel`.

Table 2 summarizes some size measurement for the target applications and their respective crosscutting concerns. The Number of Classes (NC) and Lines of Code (LOC) metrics indicate the size of each application in terms of classes/interfaces and lines of code, respectively. Additionally, Concern Diffusion over Classes (CDC) [16,18] and Lines of Concern Code (LOCC) [16,18] measure the concern size using the same units, i.e., classes/interfaces and lines of code implementing the target concern. All concerns and applications share similar complexity and were carefully chosen to allow subjects to finish the experimental task within 90 minutes (see more details in Section 2.4). For instance, although the Chess application is larger than the others, we verified that its source code is easier to be understood since each chess piece just implements a `legalMove()` method. The ErrorMessages concern is also very easy to be located and refactored in this application. Furthermore, we chose simple applications with distinct characteristics and from different domains to allow us assessing the complexity behind heterogeneous AOP constructs, such as inter-type declaration, pointcut, and advice.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

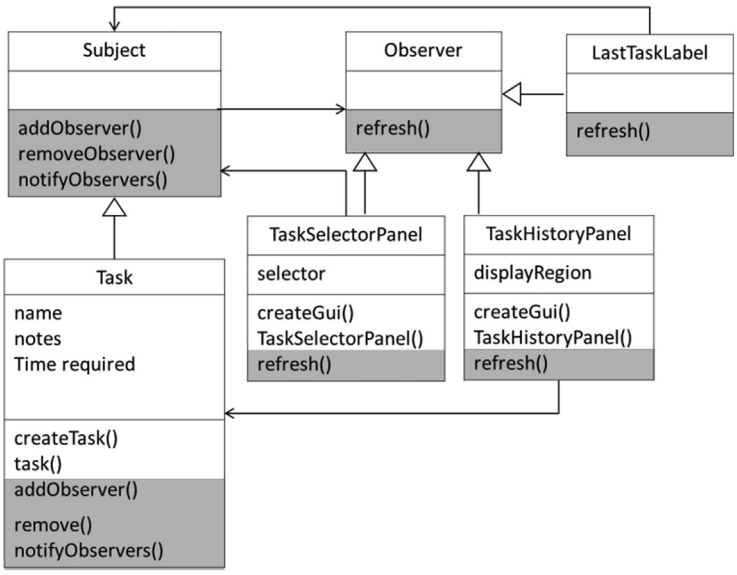

Fig. 6. Class Diagram of the Task Manager application.

Table 2
Size metrics of the target applications and concerns.

Application/concern	Application size		Concern size	
	NC	LOC	CDC	LOCC
ATM/Logging	12	606	4	19
Chess/ErrorMessages	13	1011	8	36
Telecom/Timing	8	213	4	32
Task Manager/Observer	6	209	3	22

2.4. Experimental tasks

In this experiment, each subject was asked to refactor to aspect a crosscutting concern existing in an OO application. Subjects performed their tasks using the AspectJ programming language [2,24,32]. We decide to adopt AspectJ because it is the most widely used AOP language and supports constructs available in other AOP languages, such as CaesarJ [30]. Before performing the experimental tasks, all participants attended a 2-hour training session about AOP and AspectJ. In the first part of the session, the participants learned about the basic concepts of AOP, such as the principle of separation of concerns and the join point model. In the second part, they were taught AspectJ specific constructs, such as pointcut, advice, and inter-type declaration.

After the training session, each group of participants received the source code of a small Java application and a textual description of the crosscutting concern. Using the Eclipse IDE (version 3.7 Indigo) with the AJDT plug-in (version 2.1.3) properly installed, subjects were asked to refactor the crosscutting concern from the given application using the AspectJ language. Subjects were responsible for both correctly identifying the concern code in the application and choosing the proper AspectJ construct to refactor the given crosscutting concern. No instruction was given in this sense because we consider these tasks part of the AOP programmer duties. Applications refactored by subjects do not include the tests. We took this decision to prevent subjects of checking whether the faults are in the refactored code and, thereby, mitigating some sorts of mistakes.

After subjects concluded their tasks, we performed our analysis in three steps. First, we investigated which were the categories of AOP-specific mistakes the programmers frequently made (Section 3). We then classified the subjects and analyzed the results according to three criteria (Section 4): OOP knowledge, work experience, and pair programming. This analysis relied on 98 subjects composed of undergraduate and post-graduated students and junior professionals with different background levels. Finally, we identified situations in OO code, named *code pitfalls* (Section 5), that may lead programmers to make mistakes.

3. Classification of recurring mistakes

Mistakes made by subjects during the experimental tasks were collected and categorized according to a classification of mistakes described in this section. The classification presented herein was built upon existing catalogs of AOP-specific faults

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

```
public class ATM {
  private static final int LOG = 0;
  void performTransactions() {
    ...
    while (!userExited) {
      ...
      switch (mainMenuSelection) {
        ...
        case LOG:
          Logger.printLog();
          break;
      }
    }
  }
  ...
}
```

(a) Before refactoring

```
public aspect Logging {

  public pointcut performTransactions() :
    call (void ATM.performTransactions());

  after() returning () :
      performTransactions() {
    Logger.printLog();
  }
  ...
}
```

(b) Faulty code after refactoring

Fig. 7. Example of the Incorrect Implementation Logic mistake.

```
public class BankDatabase {

  public void credit(int userAccount, double amount) {
    getAccount(userAccount).credit(amount);
    Logger.log(userAccount + " made a deposit of " + amount);
  }
  ...
}
```

```
public aspect Logging {

  public pointcut credit(int userAccount,double amount):
    call (public void BankDatabase.credit(int, double))&&
    args(userAccount, amount);

  before(int userAccount, double amount) :
      credit(userAccount, amount) {
    log(userAccount + " made a deposit of " +amount);
  }
  ...
}
```

(b) Faulty code after refactoring

Fig. 8. Example of the Incorrect Advice Type mistake.

and bug patterns [3,6,12,46]. It also includes additional mistake categories derived from our observations. We aim to classify all mistakes found in all 98 AOP implementations analyzed in this study. Therefore, some categories are general enough to embrace several similar mistakes. Each category of mistake is presented together with an example extracted from one of the applications described in Section 2.3. This classification focuses on recurring mistakes made by programmers that are not familiar with AOP and these mistakes do not always lead to a fault in software.

3.1. Pointcut and advice

This section presents two types of mistakes, namely Incorrect Implementation Logic and Incorrect Advice Type, which are associated with pointcut and advice.

Incorrect Implementation Logic. This mistake occurs when a piece of code is aspectized through pointcut and advice mechanisms. However, the refactored code behaves differently from the original one. For instance, by selecting an option in the main menu of ATM (Section 2.3), a logging report of all performed operations is displayed on screen. Fig. 7.a presents the Java code fragment to exemplify this scenario. Code shadowed in grey implements the Logging concern. Programmers often made Incorrect Implementation Logic when refactoring this code because it involves an intricate concern code nested in switch and while constructs. Fig. 7.b shows an instance of faulty code produced by a subject of this study after refactoring the code of Fig. 7.a. This AspectJ solution is incorrect because the log report is always displayed on screen after the `performTransactions()` method is called.

Incorrect Advice Type. In this mistake category, the programmer uses an incorrect type of advice to refactor a piece of concern code. For instance, Fig. 8.a shows that the message that records information about the deposit operation is logged at the end of the `credit()` method of the `BankDatabase` class in the OO version of the ATM application. This operation should be refactored as an after advice. However, some subjects of this study wrongly used a before advice to refactor the Logging concern, as presented in Fig. 8.b. The refactored code is incorrect because the aforementioned log message is stored

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

```
public class Task implements Subject {
  ...
  public void createTask(String nName, String nNotes, String nTimeRequired) {
    name = nName;
    notes = nNotes;
    timeRequired = getTimeRequired(nTimeRequired);
    notifyObservers();
  }
  ...
}
```

(a) Before refactoring

```
public aspect TaskObserver {

  declare parents: Task implements Subject;

  pointcut subjectChange(Subject subject):
    call (void Task.createTask() && target(subject);
  ...
}
```

(b) After refactoring

Fig. 9. Example of the Compilation Warning mistake.

at the beginning of the credit transaction. Therefore, a credit may be logged in this AspectJ solution even if an error occurs during the transaction.

3.2. Compilation-related mistakes

Compilation Error or Warning. This mistake refers to either a syntactic fault or a potential fault signaled by the AspectJ compiler.[1] For instance, in the Task Manager application, the subject class has to notify observers after a task has been created. That is, the method `createTask()` has a call to `notifyObservers()` as shown in Fig. 9.a. Although this method has three parameters, some subjects have not specified the method parameters neither used a wildcard when writing the pointcut to select this join point (Fig. 9.b). As a result, the AspectJ plugin for Eclipse (AJDT) notifies the programmer by presenting a warning message that no join point matched this specific pointcut. The piece of code in Fig. 9.b is incorrect because the `subjectChange` pointcut is never applied. It is interesting to observe that, in our study, sometimes novice programmers are not aware of warning messages issued by AJDT. Therefore, AJDT maintainers may consider improving the way warnings are presented to AOP programmers.

3.3. Misplaced code

Duplicated Crosscutting Code. In this mistake, part of the code that implements a crosscutting concern appears replicated in both aspects and the base Java code. For instance, the `getErrorMsg()` method of the `ChessPiece` class implements part of the ErrorMessages concern and should be refactored to aspects in the Chess application (see Fig. 4 in Section 2.3). However, this piece of code is sometimes not only implemented into aspects but also left in the original class (i.e., `ChessPiece`). Therefore, this mistake results in duplicated code.

Excessive Refactoring. This kind of mistake is made when part of the base code that does not belong to a concern is refactored to aspects. As an example, when refactoring the Timing concern in Telecom, developers should move the `timer.start()` call from the `complete()` method in the `Connection` class to an advice. Fig. 10.a shows this method call highlighted in grey. However, some subjects of our experiment also refactored the previous statement: `System.out.println("connection completed")`. Although this statement also appears in the `complete()` method, it should not be refactored. The AspectJ code shown in Fig. 10.b. is considered incorrect because the `System.out.println("connection completed")` call does not belong to the Timing concern and, therefore, it should not be refactored.

Incomplete Refactoring. This mistake occurs when developers fail to refactor part of the crosscutting concern. Consequently, part of concern still remains in the base code in the AOP solution. This mistake was made, for instance, in the aspectization of the `totalConnectionTime` attribute of the `Costumer` class that implements part of Timing in the Telecom application (see Fig. 17.b in Section 5.1). When refactoring this attribute, some subjects did not transfer this attribute to an aspect.

[1] Syntactic problems in AspectJ code are not always promptly signaled by the compiler and programmers can only notice such problems when they perform a full weaving. Considering that programmers usually work with tied schedule, they do not always double-check their code to remove these types of faults.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

```
public abstract class Connection {
  void complete() {
    state = COMPLETE;
    System.out.println("connection completed");
    timer.start();
  }
  ...
}
```

(a) Before refactoring

```
public aspect Timing {

  pointcut complete(Connection c):
      call(void Connection.complete())&& target(c);

  after(Connection c): complete(c) {
    System.out.println("connection completed");
    c.timer.start();
  }
  ...
}
```

(b) After refactoring

Fig. 10. Example of the Excessive Refactoring mistake.

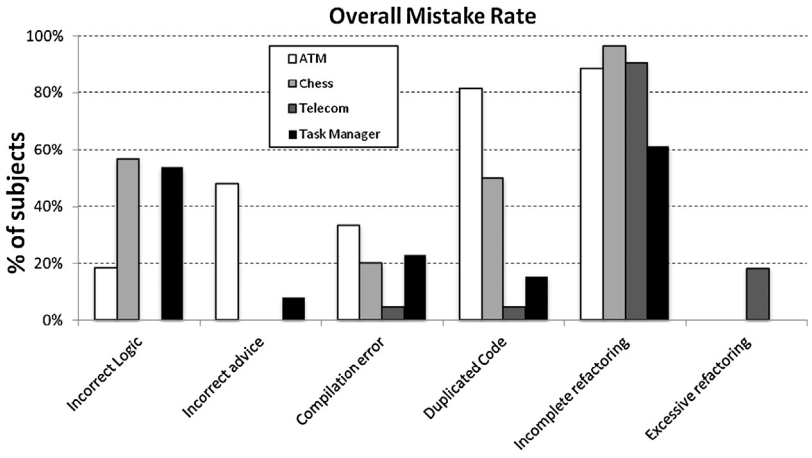

Fig. 11. Percentage of subjects that made each category of mistake.

4. Experimental results and analysis

This section presents and analyzes the results of our experiment. Section 4.1 presents the overall data of mistakes organized by target application. Sections 4.2 to 4.4 discuss the results in terms of background information of subjects: OOP knowledge, work experience, and pair programming, respectively.

4.1. Overall mistakes per application

Our analysis is based on 98 AOP implementations of the four target applications (Section 2.3). AOP-related mistakes in these implementations were analyzed and classified according to the categories discussed in Section 3. Fig. 11 presents the overall percentage of subjects that made mistakes within each category taking each application into consideration. Subjects were related to a given category if they made at least one mistake within that category regardless of the absolute number of mistakes. We made this decision because it would be hard to quantify how many times a subject makes some kinds of mistakes, such as Incomplete Refactoring, due to the higher granularity of them. Therefore, counting individual instances of a mistake could give us the wrong impression that a mistake (e.g., Incomplete Refactoring) is less frequent than others (e.g., Incorrect Advice Type).

Fig. 11 shows that Incomplete Refactoring is the most common mistake in all applications. Additionally, this mistake occurred with fairly the same rate (above 80%) in three (out of four) applications. Unlike Incomplete Refactoring, some mistakes are more common in some applications than in the others. For instance, Duplicated Crosscutting Code is more common in the ATM and Chess applications than in Telecom or Task Manager. The Incorrect Advice Type mistake occurred only in ATM and Task Manager, whilst no occurrences were observed in Chess and Telecom. Incorrect Implementation Logic, on the other hand, is more frequent in Chess and Task Manager applications. These results are mainly due to (i) the

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

```
public class BasicSimulation extends AbstractSimulation {
...
  protected void report(Customer c) {
    System.out.println(c + " spent " + c.getTotalConnectTime());
  }
}
```

Fig. 12. The Timing concern implementation in the BasicSimulation class.

crosscutting nature of the aspectized concern and (ii) properties of the target application. For instance, no subject has made the Incorrect Advice Type mistake in Chess and Telecom. After inspecting the concern code, we observed that all advices used to aspectize ErrorMessages (Chess) and Timing (Telecom) should be activated after the join point. Therefore, due to the homogeneity of the concern code, it is unlike that someone uses a different type of advice and makes this kind of mistake in these two concerns.

Incomplete Refactoring is often made in all sorts of systems due to at least two major reasons: inability to identify the concern code and inability to separate it. In the former case, programmers fail to assign code elements to the concern that should actually be later refactored. Nunes et al. [34] have highlighted that the inability to identify the concern code is one of the most problematic steps for concern refactoring. We empirically confirm their results. With respect to the inability to separate concern code, we found out that programmers avoid refactoring very tangled pieces of code. That is, in cases which are not easy to find an appropriate AOP solution to untangle the concern code, programmers end up leaving it mixed up with the base application code.

The Excessive Refactoring mistake only happened in the Telecom application (see Fig. 12). We observed that this kind of mistake barely occurs in software aspectization. In fact, the particular way Timing is implemented in Telecom might have caused programmers to make Excessive Refactoring. For instance, Fig. 12 shows the partial OO implementation of the `BasicSimulation` class in the Telecom application. The grey shadow indicates a line of code realizing the Timing concern. Programmers are expected to use pointcut/advice to aspectize this part of the Timing code. However, some programmers moved the `report()` method from the `BasicSimulation` class to an aspect and introduced it back by means of inter-type declaration [25]; that is, they aspectized the whole method instead of just a single line of code. This mistake occurred mainly because the `report()` method has a solo line of code. Therefore, OO programmers feel uncomfortable with an empty method in the application and prefer moving it to an aspect.

4.2. Tied OOP practices lead to AOP-related mistakes

This section further discusses how the level of experience in OOP may have impacted on the mistakes subjects made. Based on answers acquired from the background questionnaire (Section 2.2), subjects were divided into three categories according to their claimed OOP knowledge: (i) *novice* includes subjects with no or little knowledge in OOP; (ii) *moderate* includes subjects with moderate knowledge; and (iii) *expert* represents subjects who claimed to have extensive OOP knowledge. We grouped subjects with no or little knowledge in the same category because only three subjects claimed to have no knowledge in OOP. Additionally, when subjects worked in pairs, we considered the level of the most knowledgeable participant in the pair-wise group. For instance, if one group member claimed to have moderate knowledge in OOP and the other group member have extensive knowledge, then we consider the subject (i.e., the group) as an expert in OOP. Subjects were divided as follows: 42 novices, 45 moderates, and 11 experts.

Fig. 13 presents the percentage of subjects in each group that made mistakes, organized per mistake category (Section 3). Surprisingly, data on this chart indicates that the more experienced in OOP programmers are, the more AOP-related mistakes they make. The only exception is Compilation Error which is more frequently made by less experienced OO programmers. This result is interesting because the common sense tells us that better OO programmers are expected to be better in AOP as well. However, if we reason carefully about the results depicted in Fig. 13, we come up with the conclusion that experienced programmers in OOP stick using the OO constructions which they are familiar with, avoiding or making mistakes with the AspectJ new constructs.

For instance, experienced OO programmers more often make the Incomplete Refactoring mistake. In fact, all subjects that claimed to have extensive knowledge in OOP made this kind of mistake in the Telecom application. As illustrated by Fig. 12 and discussed in Section 4.1, subjects tend to refactor the `report()` method of `BasicSimulation` using inter-type declaration instead of using pointcut-advice mechanisms. This inappropriate choice of AspectJ mechanisms is due to the fact that the use of pointcut-advice results in an empty hook method in the `BasicSimulation` class. Programmers with knowledge in OOP see empty methods as an awful OO code [29]. That is, they are tied to the best OOP practices and do not open up their minds to a different programming paradigm. Therefore, we conclude that fresh programmers are better prepared to accept the AOP philosophy in their first experience with it.

Fig. 13 is not enough to determine whether the errors made by programmers in AOP are significantly connected with the level of experience in OOP. Therefore, a significance test should be applied. We start by defining the hypotheses below, which can be accepted or rejected by the test.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

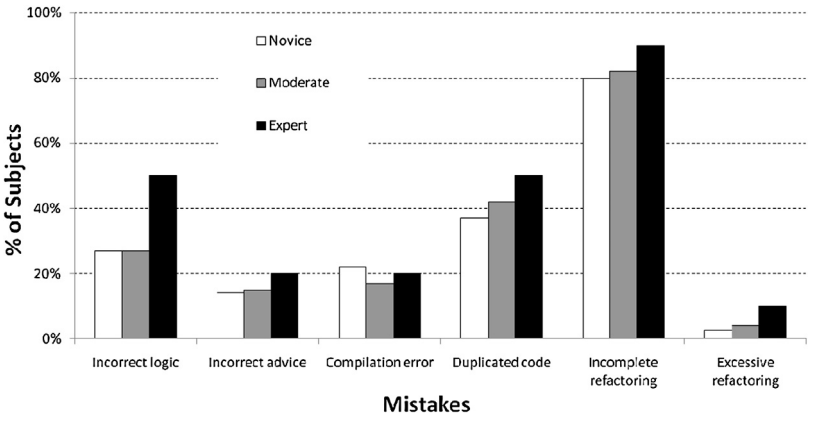

Fig. 13. Mistakes per level of OOP experience.

Table 3
p values for factor OOP experience.

OOP experience	
Type of mistake	p value
Incorrect Implementation Logic	0.001
Incorrect Advice Type	0.712
Compilation Error	0.485
Duplicated Crosscutting Code	0.792
Incomplete Refactoring	0.727
Excessive Refactoring	0.503

H1.0: Level of experience in OOP does not impact the number of mistakes in AOP.
H1.1: Level of experience in OOP impacts the number of mistakes in AOP.

The significance test we used is the full factorial type with application of one-way Analysis of Variance (ANOVA). ANOVA computes a p-value which indicates whether at least two treatments are significantly different from each other. The smaller the value of p the more significant the difference is and, consequently, the stronger the rejection of the null hypothesis. Given the level of significance, α, we can reject the null hypothesis if $p < \alpha$. In this study, we decided to use $\alpha = 0.1$. Note that the same level of significance has been used in other similar studies within the Software Engineering context [9,27,47].

The p values reported by ANOVA for the factor level of OOP experience for each AOP mistake are presented in Table 3. Table 3 shows that only the Incorrect Implementation Logic mistake has significant difference between treatments. Therefore, the factor experience in OOP only contributes to rejects the null hypothesis for the mistake Incorrect Implementation Logic with a p value $= 0.001$. These results show that OOP experience impacts on this mistake made by programmers. In other words, this statistical analysis confirms that more experienced OO programmers make more mistakes of Incorrect Implementation Logic. We could not reject the null hypothesis for the other types of mistakes at 90% level of confidence.

4.3. Selecting the appropriate Advice Type and avoiding Duplicated Code are hard, even for experienced professionals

This section reports and analyzes the impact of work experience on the mistakes made by subjects. To support this analysis, we grouped subjects according to their professional experience in software development, as follows: (i) subjects with no work experience; (ii) subjects with less than 1-year work experience, (iii) subjects with 1-year to 3-year work experience, and (iv) subjects with more than 3-year work experience. In case of subjects working in pairs, we considered the participant with more experience to represent the subject. For instance, if a group subject had a member claiming no experience in software development and another with 2 years of work experience, then we consider this subject in the third category, i.e., 1-year to 3-year work experience. We identified 30 subjects with no experience, 29 subjects with less than 1-year experience, 24 subjects with 1-year to 3-year experience, and 15 subjects with more than 3 years of experience in software development.

Fig. 14 presents the overall percentage of mistakes per group of subjects, taking into account their professional experience. In this analysis, experienced subjects made more mistakes in two categories of mistakes, namely Incorrect Advice Type and Duplicated Code. Such result suggests that work experience in software development is not always a key factor to learn AOP. That is, even young programmers with little or no experience in software development are able to learn the fundamentals of AOP and implement code with fewer mistakes in some cases. Fig. 14 also indicates that identifying and refactoring all crosscutting code are not easy tasks, even for professionals with years of experience. This observation is clear by the large number of subjects that make mistakes, like Incomplete Refactoring and Duplicated Crosscutting Code. These

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

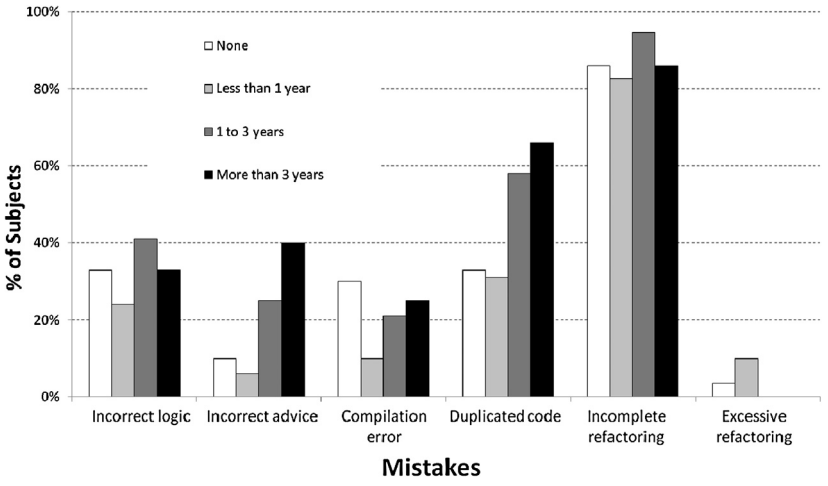

Fig. 14. Mistakes per level of work experience.

Table 4
p values for work experience.

Work experience	
Type of mistake	*p* value
Incorrect Implementation Logic	0.200
Incorrect Advice Type	0.013
Compilation Error	0.791
Duplicated Crosscutting Code	0.042
Incomplete Refactoring	0.404
Excessive Refactoring	0.262

two categories of mistakes are closely related to the ability of subjects to identify the concern code and separate it, as discussed in Section 4.1.

On the other hand, more experienced subjects made less mistakes in two categories, namely Compilation Error and Excessive Refactoring. This finding means, for instance, that the more work experience programmers have, the more attention they pay to the compiler messages. In addition, experienced programmers have more conservative behavior towards code refactoring. Therefore, they only move to aspects a piece of code that they are sure it is crosscutting. As a result, the Excessive Refactoring mistake has not occurred in code written by programmers with more than one year of work experience (Fig. 14).

Descriptive statistics presented in Fig. 14 is not enough to determine whether the difference in work experience is significant to impact on the number of mistakes. Therefore, we applied a significant test to verify the hypotheses below.

H2.0: Work experience does not impact on the number of mistakes in AOP.
H2.1: Work experience impacts on the number of mistakes in AOP.

Table 4 presents the p value reported by ANOVA for the factor level of work experience for each AOP mistake. This table shows that the Incorrect Advice Type and Duplicated Crosscutting Code mistakes have significant difference between treatments. The statistic analysis contributes to reject the null hypotheses for Incorrect Advice Type with a p value $= 0.013$ and for Duplicated Crosscutting Code with a p value $= 0.042$. The results of the statistical analysis are consistent with the results shown in Fig. 14 as we observe that programmers with more work experience made a larger number of mistakes in these two categories.

4.4. Does pair programming favor AOP refactoring?

This section discusses the impact of pair programming on the overall number of AOP-related mistakes. Each group of subjects who worked in pairs, used one of the four applications used in the experiment, namely ATM, Chess, Telecom and Task Manager. As presented in Table 1 (Section 2.2), subjects worked in pairs in three rounds of our study: Rounds 1, 3, 5, and 7.

Fig. 15 presents the percentage of subjects in each group that make the analyzed mistakes. Interestingly, our results show that pair programming is not always better than solo programming with respect to minimizing programming mistakes in AOP. For instance, Fig. 15 shows that in 3 out of 6 categories of mistakes, namely Incorrect Implementation Logic, Incorrect

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

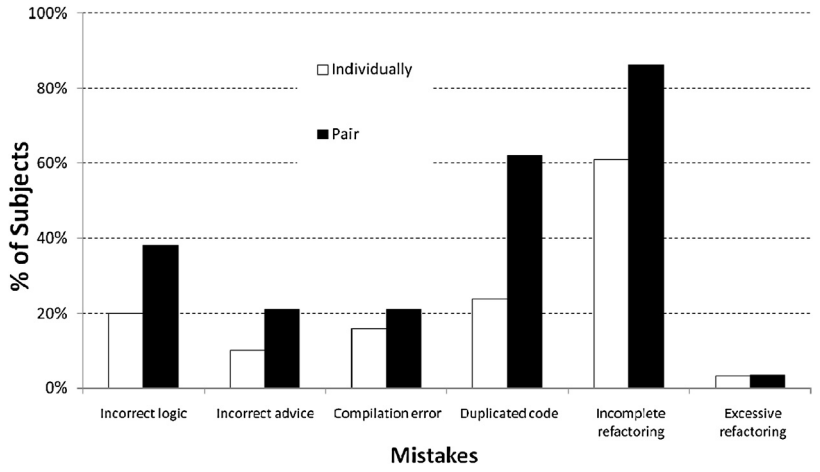

Fig. 15. Analysis of mistakes with respect to pair programming.

Table 5
p values for pair programming.

Pair programming	
Type of mistake	*p* value
Incorrect Implementation Logic	0.054
Incorrect Advice Type	0.384
Compilation Error	0.350
Duplicated Crosscutting Code	0.018
Incomplete Refactoring	0.131
Excessive Refactoring	0.696

Advice Type, and Duplicated Crosscutting Code, the number of mistakes increases with pair programming. This result might be due to the modular (localized) reasoning imposed by AOP. That is, the programmer has to reason about a single concern and sharing the knowledge about this concern might not be straightforward. For instance, to implement the logic behind a pointcut or to select the appropriated advice type, a programmer must reason about the concern in that specific localized code fragment.

On the other hand, Fig. 15 indicates that pair programming may reduce mistakes in other three categories: Compilation Error, Incomplete Refactoring, and Excessive Refactoring. It is interesting to observe that these three mistakes do not strongly depend on a localized modular reasoning. That is, they are related to the global way a concern is implemented. For instance, programmers make the Incomplete Refactoring mistake when they miss code scattered in the application. Similarly, Excessive Refactoring has to do with refactoring code of other global concerns which are not supposed to be separated.

After the descriptive statistics analysis, we also performed a significance test to verify the hypotheses presented below.

H3.0: Pair programming does not impact on the number of mistakes in AOP.
H3.1: Pair programming impacts on the number of mistakes in AOP.

Table 5 shows the *p* value reported by ANOVA for the factor level of pair programming for each AOP mistake. This table shows that only the Incorrect Implementation Logic and Duplicated Crosscutting Code mistakes have significant difference between treatments. Therefore, pair programming only contributes to reject the null hypotheses for these mistakes with a *p* value of 0.054 and 0.018, respectively. In other words, this analysis shows that pair programming increases the number of these mistakes made by programmers in accordance with Fig. 15.

4.5. Overall root cause analysis

This section summarizes the results of our study by discussing some of the root causes of mistakes in AOP. Table 6 summarizes some factors that may impact on the mistakes programmers made. Lines of this table list the six categories of mistakes described in Section 3, whilst the columns relate them to four possible impacting factors, namely Aspectized Concerns, OOP Knowledge, Work Experience, and Pair Programming. In the second column (Aspectized Concern), we indicate whether the concern impact (Yes) or not (No) on the number of mistakes we observed. For the last three columns, if the analyzed factor impacts on the mistake frequency we also indicate whether it causes an increase (Negatively) or decrease (Positively) in the number of mistakes.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

Table 6
Root cause of AOP-related mistakes.

Mistakes	Impacting factors			
	Aspectized concern	OOP knowledge	Work experience	Pair programming
Incorrect Impl. Logic	Yes	Negatively*	Negatively*	Negatively*
Incorrect Advice Type	Yes	Negatively	Negatively	Negatively
Compilation Error	Yes	Positively	Positively	Positively
Duplicated Cross. Code	Yes	Negatively	Negatively*	Negatively*
Incomplete Refactoring	No	Neutral	Neutral	Positively
Excessive Refactoring	Yes	Negatively	Positively	Positively

* A star means that the result is statistically significant.

```
public class ATM {

   private static final int LOG= 0;
   ...
}
```

Fig. 16. Example of the Primitive Constant code pitfall.

Table 6 indicates, for instance, that programmers with both OOP knowledge and work experience should be particularly careful to avoid the Incorrect Advice Type and Duplicated Crosscutting Code mistakes. Instances of Incomplete Refactoring, on the other hand, depends neither on experience in software development nor on the target concern. However, pair programming practices might help avoiding this kind of mistake.

5. Aspectization code pitfalls

After collecting the mistakes made by the subjects of this experiment, we analyze situations that could have led them to make these mistakes. We name these situations *code pitfalls*. The analysis of code pitfalls was divided into two phases. First, we performed a manual inspection in the OO source code of the applications used in the experiment in order to identify code pitfalls for AO refactoring. Second, we interviewed some of the experiment participants with the purpose to verify whether such code pitfalls may have impacted on the mistakes they made. The result of this investigation is a preliminary catalog of code pitfalls presented in this section. We organize these code pitfalls into two major categories: Dedicated Implementation Elements (Section 5.1) and Non-Dedicated Implementation Elements (Section 5.2).

5.1. Dedicated implementation elements

This section discusses two code pitfalls related to dedicated implementation elements, namely Primitive Constant and Attribute of a Non-Dedicated Type. We define a *dedicated implementation element* as a class, method or attribute completely dedicated to implement a concern.

Brinkley et al. [7] advocated that refactoring dedicated implementation elements should be straightforward since it only requires moving them from a class to an aspect by using inter-type declarations, for instance. However, a high number of refactoring mistakes were observed in this experiment when developers have to refactor dedicated elements. In particular, Duplicated Crosscutting Code and Incomplete Refactoring (Section 3) are commonly related to dedicated implementation elements. The synergy between these two mistakes and dedicated implementation elements could be due to the concern mapping task, i.e., the developer fails to assign dedicated elements to a concern. In fact, Nunes et al. [34] have pointed out that not assigning a dedicated implementation element to a concern is a common mistake when developers perform concern mapping. Therefore, it is not a surprise that developers make the same mistake when refactoring a crosscutting concern to aspects.

Primitive Constant. The first code pitfall, named Primitive Constant, occurs when a constant is dedicated to implement the concern. For instance, Fig. 16 shows a constant called `LOG` that implements the Logging concern in the ATM application. In our study, 85% of the subjects of the first round and 86% of the second round made mistakes when refactoring this part of the Logging concern. For instance, subjects often replaced this constant by its value in places where it is used; this seems a common strategy to simplify the refactoring. However, they did not remove this constant declaration from the base code which results in an unreachable code [12]. This kind of mistake was classified as Duplicated Crosscutting Code since both the base code and the aspect are dealing with the constant value. In addition to Duplicated Crosscutting Code, we observed that Primitive Constant is also related to the Incomplete Refactoring mistake. One example of this kind of mistake occurs when developers just ignored moving the constant to an aspect, leaving it at the base code.

Attribute of a Non-Dedicated Type. This code pitfall occurs when an attribute has either a primitive type (e.g., boolean, int, long) or a non-dedicated element type. For instance, the Timing concern in Telecom is implemented by two attributes. One of these attributes, shown in Fig. 17.a, is of the type Timer which is completely dedicated to the concern. Subjects of the

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

```
public abstract class Connection {
  Timer timer = new Timer();
  ...
}
```

(a) Attribute of a dedicated type

```
public class Costumer {
  ...
  long totalConnectionTime = 0;
  ...
}
```

(b) Attribute of a non-dedicated type

Fig. 17. Example of the Attribute of the Non-Dedicated Type code pitfall.

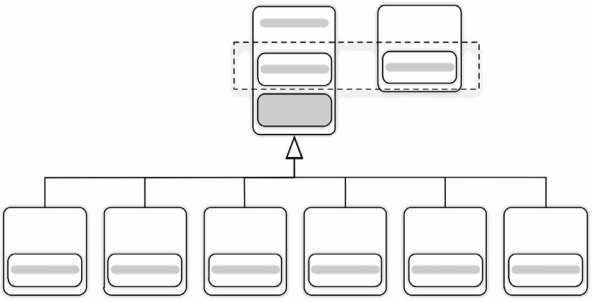

Fig. 18. Representation of the Disjoint Inheritance Trees code pitfall.

experiment had no problem to locate and refactor this attribute to aspect. On the other hand, Fig. 17.b shows an attribute of the type long which is not dedicated to Timing; it is actually a primitive Java type. When this concern is refactored, 33% of subjects in the fifth replication and 32% in the sixth replication did not aspectize the `totalConnectTime` attribute, but none of them failed to refactor the `timer` attribute. In our analysis, we observed that developers appear to have difficulty to assign attributes of non-dedicated types (e.g., `totalConnectTime` in Fig. 17.b) to a concern during concern mapping tasks. This fact is probably due to the lack of searching support from IDE. Consequently, the presence of such attributed in the concern code leads programmers to make the Incomplete Refactoring mistake.

5.2. Non-dedicated implementation elements

Several mistakes in this experiment occurred when subjects had to refactor elements which are not completely dedicated to implement a concern; we call them *non-dedicated implementation elements*. To refactor these elements, developers usually rely on pointcut/advice constructs of the AOP language. Three code pitfalls are related to non-dedicated implementation elements: Disjoint Inheritance Trees, Divergent Advice, and Concern Attribute Accesses. These code pitfalls are discussed below.

Disjoint Inheritance Trees. This code pitfall is characterized by two or more inheritance trees involved in the concern realization. Elements of one trees usually does not refer to the elements of the other tree. The lack of explicit relationships between two inheritance trees might be one reason that subjects correctly refactored elements in one tree, but not in the other. In addition, elements in the same inheritance tree tends to follow the same pattern in the concern implementation. Fig. 18 illustrates the Disjoint Inheritance Trees code pitfall. Elements inside the dotted square are more prone to the Incomplete Refactoring mistake because they do not follow the same pattern of concern code implemented in tree leaves.

For instance, the ErrorMessages concern in the Chess application is implemented by two inheritance trees. Moreover, pieces of the concern code scattered in one tree follow the same pattern. In this case, all subclasses of `ChessPiece` set a new value in the `strErrorMsg` attribute (see Fig. 4 in Section 2.3) as part of the concern implementation. All subjects of the third and fourth round made the Incomplete Refactoring mistake when refactoring parts of the concern code either in `ChessPiece` or in `ChessBoardWindow`. However, almost everybody correctly aspectized code in the `ChessPiece` subclasses (e.g., `Pawn`, `Rock`, `King`, etc.). This code pitfall might also be related to the concern mapping task, i.e., developers assign to the concern just code inside one inheritance tree. Consequently, we conclude that when a system has one or more inheritance trees related to the concern, like the aforementioned case, developers are more prone to make Incomplete Refactoring.

Divergent Jointpoint Location. This code pitfall occurs when (i) pieces of the concern code is scattered over several methods and (ii) these pieces of code appear in different locations with respect to the method body. The problem mainly occurs when just a couple of pieces of code have a different location compared to the others. Fig. 19 illustrates this code pitfall. In this figure, the concern code is located at the end of four methods, but at the beginning of one method (marked with a dotted square). Therefore, developers should use after advice to refactor the former pieces of code, but a before advice in the latter case. This divergent jointpoint location usually tricks developers and they end up making the Incorrect Advice Type mistake.

As an example of this code pitfall, Logging in the ATM application is realized by the Logger class and several scattered calls to the `log()` method of this class. Only one call to `log()` appears in the beginning of a method and should be

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

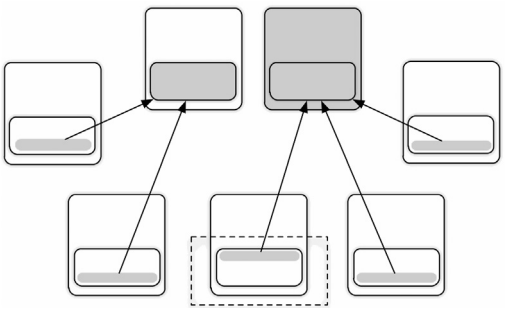

Fig. 19. Representation of the Divergent Jointpoint Location code pitfall.

```
public class ATM {
  private void performTransactions() {
    ...
    while(!userExited) {
      int mainMenuSelection = displayMainMenu();
      switch(mainMenuSelection) {
        ...
        case Log:
          Logger.printLog();
          break;
        ...
      }
    }
  }
  ...
}
```

Fig. 20. Part of the Logging concern related to Incorrect Implementation Logic.

refactored using a before advice. All other `log()` calls appear in the end of a method and should be refactored using after advice. Therefore, developers have to be careful to use the appropriate advice type – before or after in this case - to refactor the Logging concern. In fact, we have noticed that many subjects of the first and second rounds refactored all of the above concern parts using only after advice. They thus made the Incorrect Advice Type mistake in the case which they were expected to use a before advice.

Accesses to Local Variables. This code pitfall is characterized by the presence of concern code inside a method that accesses a local variable. Previous work [10] has advocated that the dependence on local variables make the concern code harder to refactor, since the join point models of AspectJ-like languages cannot capture information stored in such variables. On the other hand, capturing the value of class members and method parameters is straightforward. For the former, for instance, it can be done by using the *args* AspectJ pointcut designator.

To illustrate this code pitfall, we rely on a piece of the Logging concern (ATM) shown in Fig. 20. About 40% of subjects made the Incorrect Implementation Logic mistake when refactoring to aspects this piece of code. The logical expression that controls the execution flow of the switch statement in Fig. 20 is composed by the local variable `mainMenuSelection`. This variable is declared inside the method body in which this code appears. In order to aspectize the grey code in Fig. 19, developers need to access to this variable and check whether its value is equal to the `LOG` constant. However, this refactoring is not easy and, so, developers recurrently make mistakes in such trick aspectization scenario.

6. Tool support

This section presents ConcernReCS[2] [5], an Eclipse plug-in to find in Java source code the aspectization code pitfalls discussed in Section 5. ConcernReCS extends ConcernMapper [39] which is a tool to allow the mapping of methods and attributes to concerns. Fig. 21 describes a simplified flow chart of ConcernReCS. This figure illustrates that ConcernReCS receives information from both ConcernMapper and the Eclipse platform. The Data Analyzer module of ConcernReCS threats this information and generates warnings of code pitfalls.

The first step in order to use ConcernReCS is to map the dedicated implementation elements of one or more concerns using ConcernMapper. That is, methods, classes, and attributes completely dedicated to a concern have to be manually mapped. Non-dedicated implementation elements are automatically inferred by (i) calls to dedicated implementation methods, (ii) read or write accesses to dedicated implementation attributes, and (iii) syntactic references to dedicated classes or interfaces. Other sorts of concern code, such as conditional statements, should be extracted by the Extract Method refactoring [20] and, then, the resulting method should be mapped to the concern.

[2] Available at http://sourceforge.net/p/concernrecs/.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

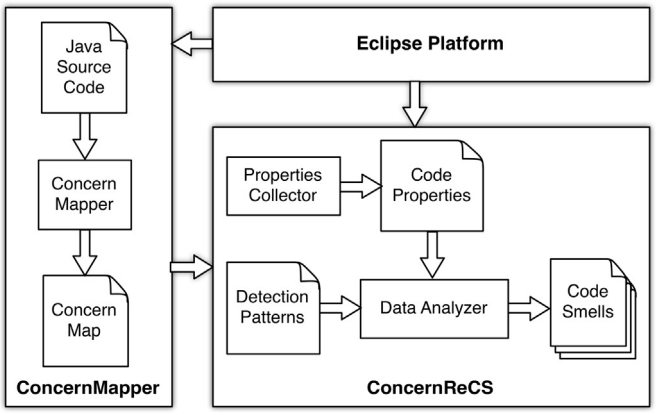

Fig. 21. ConcernReCS simplified flow chart.

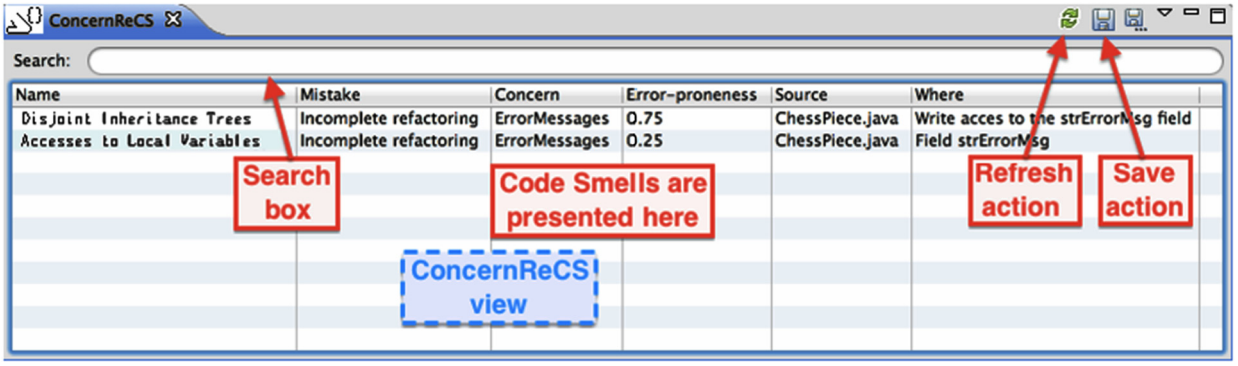

Fig. 22. The ConcernReCS main view.

Fig. 22 presents the main view of ConcernReCS in the Eclipse IDE. Code pitfalls are presented one per line and, for each of them, it is shown the code pitfall name and the mistake that it can lead to. ConcernReCS also indicates the concern in which the code pitfall appears, the source file and where in the system code it appears. For each code pitfall, the tool also gives a number representing its error-proneness. The error-proneness value can be 0.25, 0.5, 0.75, or 1. These values mean that at leas this percentage of subjects made mistakes related to the code pitfall. For instance, the 0.5 value for Accesses to Local Variables means that between 50% and 74% of subjects made mistakes in cases where this code pitfall occurred in code. The value of 1 represents a situation in which all subjects in our experiment made mistakes related to a specific code pitfall. The tool preference page allows developers selecting code pitfalls and concerns that should be taken into account by ConcernReCS.

7. Limitations

The use on one AOP language (AspectJ) can be a limitation in our study and its conclusions. However, it is widely known that AspectJ-based systems have been used as targets in the large majority of AOP assessment studies. Even though there are other programming techniques which are able to realize the concepts of AOP (examples mentioned by Filman and Friedman [19] are Intentional Programming, Meta Programming and Generative Programming), so far AspectJ has been the most adopted AOP technology [37].

Another limitation regards the size and representativeness of the target applications and the crosscutting concerns selected to be refactored in each application; namely, Logging in ATM, ErrorMessages in Chess, Observer in TaskManager, and Timing in Telecom. Regarding the size of the applications, we recognize that they do not reflect the complexity observed in the industrial context with respect to lines of code and number of modules. However, the need to perform the experimental tasks in a short pre-specified period of time restricts the size and complexity of applications and concerns that can be used in this kind of experiment. This is why applications of similar size have been used in recent programming-related experiments with different goals within varied contexts [8,17,26]. We have also selected crosscutting concerns of different nature; i.e., two non-functional (Logging and ErrorMessages), functional (Timing), and one design pattern (Observer). These concerns hold similar complexity in order to provide subjects with balanced implementation scenarios in terms of difficulty.

Section 2.2 showed that around 50% of the participants declared to have moderate to high level of knowledge on OOP, while the remaining participants classified themselves as having low-expertise in OOP. Despite this heterogeneity, it does

not seem to have had great impact on the types of mistakes subjects made while performing the experiment tasks, as discussed in Section 4.2. Note that their level of expertise in AOP was indeed not expected to be high, since one of the experiment goals is the analysis of the kinds of mistakes which are most likely to be made by novices programmers in AOP.

8. Related work

This section summarizes two categories of related work. The first category aims at investigating AOP-related pitfalls, fault types and faulty scenarios. The second category investigates concern identification and code smells in AO artifacts.

Investigations of AOP-related pitfalls, fault types and faulty scenarios. Rashid et al. [37] discussed two main pitfalls that may hinder the adoption of AOP techniques and technologies. The first regards the fragility of pointcuts that are based on naming conventions of coding elements (which misleads join point selection as long as the software evolves), while the second regards unexpected exception flows caused by the mixed use of OO and AO exception handling constructs. These pitfalls were empirically observed by Ferrari et al. [13] and Coelho et al. [11] in two exploratory studies, respectively. Note that the code pitfalls described by Rashid et al. and observed by others [11,13] were characterized based on a series of releases of medium-sized, robust[3] AspectJ systems implemented by experienced developers. Our study, on the other hand, addressed small-sized, simple implementations that could be handled by novice programmers in a restricted experiment time. Moreover, the aspectization code pitfalls we described in our work regard characteristics of OO code which one aims to refactor out to aspects, whilst the ones described by Rashid et al. [37] are related to code that already includes AspectJ constructs.

Other previous studies in AOP-specific fault types and bug patterns [3,6,14,46] address from the most basic concepts of AOP to advanced constructions of AspectJ-like languages. In previous research, Ferrari et al. [12] analyzed the existing taxonomies and catalogues and identified four main categories of faults that can be found in AO software programs. They also quantified the occurrences of each fault type in several releases of three medium-sized AspectJ systems. According to the authors, their study was the first that analyzed quantitative data regarding fault occurrence in AO systems.

When we compare the contributions of this paper to the results presented by Ferrari et al. [12], we can spot two major differences. First, we characterized mistakes that novice programmers are likely to make while refactoring out to aspects crosscutting concerns in existing OO systems. Note that such mistakes may or may not lead to a fault in the refactored system. Ferrari et al., on the other hand, characterized faults revealed during the testing phase. Second, we are not concerned in this paper with the frequency at which each mistake is committed, but with the types of mistakes the programmers make. Differently, Ferrari et al. quantified the number of times each fault type was localized in the evaluated systems, be them refactoring-related or not, which shall be prioritized in testing strategies.

Burrows et al. [8] performed an exploratory experiment in order to analyze how faults are introduced into existing AO systems during adaptive and perfective maintenance tasks. The AO version of the Telecom system,[4] as distributed with the AspectJ language, was used as a basis for a set of maintenance tasks performed by experienced, paired programmers. Our study differs from Burrows et al.'s in the sense that while they evolved an existing AO system, our study participants were assigned to tasks of refactoring crosscutting concerns of OO systems in order to produce equivalent AO counterparts.

Shen et al. [40] presented XFindBugs, an extension of FindBugs for AspectJ. This tool aims to help programmers find potential bugs in AspectJ applications through static analysis. XFindBugs supports 17 bug patterns to cover common error-prone features in an aspect-oriented system. It also integrates the corresponding bug detectors into the FindBugs framework. Unlike XFindBugs, ConcernReCS do not integrate with FindBugs and it aims to warn novice developers of code pitfalls documented in this study.

Investigations of concern identification and code smells. Recent studies [16,34] have investigated mistakes in the projection of concerns on code – a key task in concern refactoring. For instance, Figueiredo et al. [16] 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. Nunes et al. [34] characterized a set of recurring concern mapping mistakes made by software developers. They stress the fact that concern mapping is a primordial task to guide software maintenance tasks. This may help us to explain the high occurrence of refactoring mistakes observed in our study: the participants had to first identify which parts of the code referred to the target crosscutting concern.

In another piece of recent research, Macia et al. [28] reported on the results of an exploratory study of code smells in evolving AO systems. They analyzed 18 releases of three medium-sized AO systems with the aim of assessing previously documented AOP code smells [35,43] and new ones characterized by the authors. Although our goal in this paper was neither characterizing nor assessing code smells for AO (AspectJ) programs, we shall perform similar analysis of the code pitfalls described in Section 6. For instance, we can investigate the relationships between different pitfalls spotted in common implementations and the similarities of these pitfalls with code smells documented by previous studies [28,35,43].

Previous studies in AOP-specific mistakes [3,12,14,46] range from the most basic AOP concepts to advanced constructions of AspectJ-like languages, such as intertype declarations. Since our study was conducted only with novice programmers in

[3] By robust systems, we mean systems that include comprehensive exception handling considering its major modules (i.e. classes and packages).

[4] Note that the Telecom version we used in our experiments is purely object-oriented, written in Java. This version was used in the refactoring tasks performed by the study participants.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

AOP, our classification partially overlaps previously defined fault taxonomies and adds new categories, such as compilation errors or warning and incomplete refactoring. Moreover, it focuses on mistakes made by programmers when using basic features of AOP languages such as pointcut, advice and intertype declarations.

9. Conclusions and future work

This paper reports a series of experiments to characterize mistakes made by programmers in typical crosscutting concern refactorings. We first document the recurring mistakes made by programmers with specific backgrounds (Section 3) and observed, for instance, that pair programming may help programmers avoid some mistakes, such as incomplete refactoring. However, it does not help to avoid advice-related mistakes (Section 4.4). The results of this study may be used for several purposes, like helping in the development and improvement of new AOP languages and tools. Based on an analysis of recurring mistakes, this paper presented a catalogue of aspectization code pitfalls (Section 5). These code pitfalls appear to lead programmers to make the documented mistakes. To support the automatic detection of code pitfalls, we developed a prototype tool called ConcernReCS (Section 6). The results (Section 4.2) indicate that the more experienced in OO programmers are, the more AOP-related mistakes they make in some specific categories, such as incorrect advice type and duplicated crosscutting code. On the other hand, compilation errors are more frequently made by less experienced OO programmers. This result is interesting because the common sense tells us that better OO programmers are expected to be better in AOP as well. Experienced programmers in OOP stick using the OO constructions which they are familiar with, avoiding the new AspectJ constructs or making mistakes using them.

Further research can rely on our study settings to perform new replications of the experiment in order to (i) enlarge our dataset, (ii) uncover additional categories of mistakes, and (iii) expand the proposed catalogue of code pitfalls. In fact, we have plans to replicate ourselves this study using different applications and subjects. For instance, we aim to perform further replications with experienced AO programmers. These replications shall allow us to perform more comprehensive analyses with statistical significance. In such further replications, we may specialize some general categories of mistakes, such as Compilation-related Mistakes, into more specific ones, such as Incorrect Pointcut Matching. Moreover, we aim to evaluate whether the proposed tool (ConcernReCS) is effective to warn programmers about code pitfalls and help them avoiding typical refactoring mistakes.

Acknowledgements

This work was partially supported by CAPES, CNPq (grants 485907/2013-5 and 485235/2013-7), and FAPEMIG (grant PPM-00382-14).

References

[1] Results of avoiding code pitfalls in aspect-oriented programming, http://www.dcc.ufmg.br/~figueiredo/aop/mistakes, 2012.
[2] AJDT: AspectJ Development Tools, http://www.eclipse.org/ajdt/.
[3] R.T. Alexander, J.M. Bieman, A.A. Andrews, Towards the systematic testing of aspect-oriented programs, Dep. of Computer Science, Colorado State University, 2004, Report CS-04-105.
[4] P. Alves, E. Figueiredo, F. Ferrari, Avoiding code pitfalls in aspect-oriented programming, in: Proc. of the Brazilian Symposium on Programming Languages, SBLP, Brazil, 2014, pp. 31–46.
[5] P. Alves, D. Santana, E. Figueiredo, ConcernReCS: finding code smells in software aspectization, in: Proceedings of the 34th International Conference on Software Engineering, Informal Research Demonstration, ICSE, Zurich, Switzerland, 2012.
[6] J. Baekken, R. Alexander, A candidate fault model for AspectJ pointcuts, in: Proc. of the 17th Int'l Symposium on Software Reliability Engineering, ISSRE, Raleigh, USA, 2006, pp. 169–178.
[7] D. Brinkley, M. Ceccato, M. Harman, F. Ricca, P. Tonella, Tool-supported refactoring of existing object-oriented code into aspects, IEEE Trans. Softw. Eng. 32 (17) (2006) 698–717.
[8] R. Burrows, F. Taiani, A. Garcia, F.C. Ferrari, Reasoning about faults in aspect-oriented programs: a metrics-based evaluation, in: Proc. of the 19th Int'l Conf. on Program Comprehension, ICPC, 2011, pp. 131–140.
[9] R. Burrows, F.C. Ferrari, O.A.L. Lemos, A. Garcia, F. Taïani, The impact f coupling on the fault-proneness of aspect-oriented programs, in: Proc. of the 21st Int'l Symposium on Software Reliability Engineering, ISSRE, 2010, pp. 329–338.
[10] F. Castor Filho, N. Cacho, E. Figueiredo, A. Garcia, C.M.F. Rubira, J.S. Amorin, H.O. Silva, On the modularization and reuse of exception handling with aspects, Softw. Pract. Exp. 39 (17) (2009) 1377–1417.
[11] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. Staa, C. Lucena, Assessing the impact of aspects on exception flows: an exploratory study, in: Proc. of the 22nd European Conf. on Object-Oriented Programming, ECOOP, 2008, pp. 207–234.
[12] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, J. Maldonado, Characterising faults in aspect-oriented programs: towards filling the gap between theory and practice, in: Proc. of the Brazilian Symposium on Software Engineering, 2010, pp. 50–59.
[13] F.C. Ferrari, R. Burrows, O.A.L. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P.C. Masiero, T. Batista, J.C. Maldonado, An exploratory study of fault-proneness in evolving aspect-oriented programs, in: Proc of the 32nd Int'l Conference on Software Engineering, ICSE, 2010, pp. 65–74.
[14] F. Ferrari, J. Maldonado, A. Rashid, Mutation testing for aspect-oriented programs, in: Proc. of the Int'l Conf. on Software Testing, Verification, and Validation, ICST, 2008, pp. 52–61.
[15] W. Fenske, S. Schulze, Code smells revisited: a variability perspective, in: Proceedings of the 9th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS, 2015, pp. 3–10.
[16] E. Figueiredo, A. Garcia, M. Maia, G. Ferreira, C. Nunes, 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, 2011, pp. 81–92.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003

[17] E. Figueiredo, et al., Evolving software product lines with aspects: an empirical study on design stability, in: Proc. of the Int'l Conf. on Software Engineering, ICSE, 2008, pp. 261–270.
[18] E. Figueiredo, C. Sant'Anna, A. Garcia, T. Bartolomei, W. Cazzola, A. Marchetto, On the maintainability of aspect-oriented software: a concern-oriented measurement framework, in: Proc. of the 12th European Conf. on Software Maintenance and Reengineering, CSMR, 2008, pp. 183–192.
[19] R.E. Filman, D. Friedman, Aspect-oriented programming is quantification and obliviousness, in: Aspect-Oriented Software Development, Addison–Wesley, 2004, pp. 21–35, Ch. 2.
[20] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison–Wesley Publishing Company, Reading, MA, 1999.
[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.
[22] IEEE Standard Glossary for Software Engineering Terminology, Standard 610.12, Institute of Electrical and Electronic Engineers, New York, USA, 1990.
[23] R. Jain, The Art of Computer System Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling, John Wiley & Sons, 1991, pp. 1–702.
[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: Proc. of the 15th European Conf. on Object-Oriented Programming, ECOOP, 2001, pp. 327–353.
[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proc. of the European Conf. on OOP, ECOOP, in: LNCS, vol. 1241, 1997, pp. 220–242.
[26] O.A.L. Lemos, F.C. Ferrari, F.F. Silveira, A. Garcia, Development of auxiliary functions: should you be agile? An empirical assessment of pair programming and test-first programming, in: Proc. of the 34th Int'l Conf. on Software Engineering, ICSE, 2012, pp. 529–539.
[27] O.A.L. Lemos, A.C. de Paula, F.C. Zanichelli, C.V. Lopes, Thesaurus-based automatic query expansion for interface-driven code search, in: Proc. of the 11th Working Conference on Mining Software Repositories, MSR, 2014, pp. 212–221.
[28] I. Macia, A. Garcia, A. von Staa, An exploratory study of code smells in evolving aspect-oriented systems, in: Proc. of the 10th Int'l Conf. on Aspect-Oriented Software Development, AOSD, 2011, pp. 203–214.
[29] B. Meyer, Object-Oriented Software Construction, 2nd edition, Prentice Hall, 2000.
[30] M. Mezini, K. Ostermann, Conquering aspects with Caesar, in: Proc. of the 2nd Int'l Conf. on Aspect-Oriented Software Development, AOSD, 2003.
[31] M.P. Monteiro, J.M. Fernandes, Towards a catalogue of refactorings and code smells for AspectJ, in: Transactions on Aspect-Oriented Software Development, in: LNCS, vol. 3880, 2006, pp. 214–258.
[32] M.P. Monteiro, J.M. Fernandes, Refactoring a Java code base to AspectJ: an illustrative example, in: Proc. of International Conference on Software Maintenance, ICSM, 2005, pp. 17–26.
[33] F. Munhoz, B. Baudry, R. Delamare, Y. Le Traon, Inquiring the usage of aspect-oriented programming: an empirical study, in: International Conference on Software Maintenance, ICSM, 2009, pp. 137–146.
[34] C. Nunes, A. Garcia, E. Figueiredo, C. Lucena, Revealing mistakes in concern mapping tasks: an experimental evaluation, in: Proc. of the 15th European Conf. on Software Maintenance and Reengineering, CSMR, 2011, pp. 101–110.
[35] E.K. Piveta, M. Hecht, M.S. Pimenta, R.T. Price, Detecting bad smells in AspectJ, Journal of Universal Computer Science 12 (7) (2006) 811–827.
[36] E.K. Piveta, A. Moreira, M.S. Pimenta, J. Araújo, P. Guerreiro, R.T. Price, An empirical study of aspect-oriented metrics, Sci. Comput. Program. 78 (1) (2012) 117–144.
[37] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Sudholt, W. Joosen, Aspect-oriented programming in practice: tales from AOSE-Europe, Computer 43 (2) (2010) 19–26.
[38] M. Robillard, G. Murphy, Representing concerns in source code, ACM Trans. Softw. Eng. Methodol. 16 (1) (2007), Article No. 3.
[39] M.P. Robillard, F. Weigand-Warr, ConcernMapper: simple view-based separation of scattered concerns, in: OOPSLA Workshop on Eclipse Technology Exchange, 2005, pp. 65–69.
[40] H. Shen, S. Zhang, J. Zhao, J. Fang, S. Yao, XFindBugs: eXtended FindBugs for AspectJ, in: Proceedings of the 8th Workshop on Program Analysis for Software Tools and Engineering, PASTE, 2008, pp. 70–76.
[41] E. Soloway, K. Ehrlich, Empirical studies of programming knowledge, IEEE Trans. Softw. Eng. (1984) 595–609.
[42] J. Spohrer, E. Soloway, Novice mistakes: are the folk wisdoms correct?, Commun. ACM 29 (7) (1986).
[43] K. Srivisut, P. Muenchaisri, Bad-smell metrics for aspect-oriented software, in: Proc. of the 6th Int'l Conf. on Computer and Information Science, 2007, pp. 1060–1065.
[44] S.A. Vidal, C.A. Marcos, Toward automated refactoring of crosscutting concerns into aspects, J. Syst. Softw. 86 (6) (2013) 1482–1497.
[45] C. Wohlin, et al., Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2012.
[46] S. Zhang, J. Zhao, On identifying bug patterns in aspect-oriented programs, in: Proc. of the 31st Int'l Computer Software and Applications Conf., COMPSAC, 2007, pp. 431–438.
[47] M. Zhao, C. Wohlin, N. Ohlsson, M. Xie, A comparison between software design metrics and code metrics for the prediction of software fault content, Inf. Softw. Technol. 40 (14) (1998) 801–809.

Please cite this article in press as: A. Santos et al., Avoiding code pitfalls in Aspect-Oriented Programming, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2015.12.003