Avoiding Code Pitfalls in Aspect-Oriented Programming

Péricles Alves[1], Eduardo Figueiredo[1], and Fabiano Ferrari[2]

[1] Software Engineering Laboratory, Federal University of Minas Gerais (UFMG), Brazil
[2] Computing Department, Federal University of São Carlos (UFSCar), Brazil
{periclesrafael,figueiredo}@dcc.ufmg.br, fabiano@dc.ufscar.br

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. Based on a series of experiments, this paper reports a catalogue of code pitfalls that are likely to lead programmers to make mistakes in AOP. Each experiment required the aspectization (i.e. refactoring) of a crosscutting concern in one object-oriented application. Six rounds of the experiment provided us with the data of 80 aspect-oriented (AO) implementations where three crosscutting concerns were aspectized in three applications. We developed a prototype tool to warn programmers of the code pitfalls during refactoring activities.

Keywords: AOP, mistakes, code pitfalls, empirical study.

1 Introduction

Aspect-Oriented Programming (AOP) [19] 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 [28]. It is well known that novice programmers need special guidance while learning how to program with a new language [30, 31]. 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.

A mistake is “a human action that produces an incorrect result” [17]. In our study, a mistake occurs when a developer performs an inconsistent code refactoring, which may result in a fault into the application. A fault, on the other hand, consists in an incorrect step, process, or data definition in a computer program [17]. In other words, a fault occurs when there is 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 not necessarily does so.

Previous research [3, 5, 9, 11] investigated mistakes that are likely to be made by AOP programmers either to build systems from scratch or to refactor existing ones. Other studies identified AO code smells [21, 24, 26]. However, these studies often consider complex systems developed by experienced programmers. In such 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. In turn, it may represent one of the reasons for the cautious adoption of AOP in real software development projects [32].

Towards addressing this problem, we identified a preliminary set of recurring mistakes made by novice AOP programmers in our previous work [4]. We derived these mistakes by running three rounds of an experiment in which 38 novice AOP programmers were asked to refactor out to aspects two crosscutting concerns from two small Java applications. The mistakes observed in our previous study are often related to fault types documented in previous research [3, 5, 9, 11]. We also documented new kinds of programming mistakes, such as Incomplete Refactoring, which does not necessary lead to faults.

This paper extends our previous study [4] and builds up on top of its results. First, this paper reports on other three rounds of the same experiment (resulting in six rounds in total) with a different application (called Telecom) and 42 additional participants. Therefore, this paper relies on data of 80 refactored code samples of three small-sized applications (Section 2). In this follow up study, our goal is not only to further investigate and confirm our previous findings, but also to find out additional categories of common mistakes made by AOP programmers. Second, this paper also presents a novel catalogue of situations that appear to lead programmers to make the identified categories of recurring mistakes (Section 3). Such situations, named Aspectization 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 Aspectization Code Pitfalls, we propose a prototype tool called ConcernReCS (Section 4). Section 5 summarizes related work and Section 6 concludes this paper with directions for future work.

2 Empirical Study

This section presents the configurations of the experimental study we conducted. Section 2.1 presents its research questions and Section 2.2 briefly describes the study participants. Section 2.3 introduces the target applications and characteristics of the refactored crosscutting concerns. Section 2.4 explains the experimental tasks.

2.1 Research Questions

This study aims at investigating the types of mistakes made by students and junior professionals when using AOP. Our goal in this study is to uncover and document code pitfalls that lead programmers to make these mistakes. Based on this goal, we formulate the research questions below. To answer RQ1, we first identify and classify the recurring categories of mistakes made by programmers learning AOP. Then, we document error-prone situations as a catalogue of code pitfalls to address RQ2.

RQ1. What kinds of mistakes do novice AOP programmers often make while refactoring crosscutting concerns?

RQ2. What are the code pitfalls in the source code that lead to these mistakes?

2.2 Background of Subjects

Participants of all six rounds of this study were organized in groups of one or two members, called subjects of the experiment. In three rounds, participants worked in pairs in order to investigate possible impact of pair programming on the mistakes [4].

Table 1 summarizes the number of subjects (i.e., groups of participants) and how they were organized in each study round. Each subject took part of only one round of the experiment. Note that in the odd rounds (1st, 3rd, and 5th), the numbers of participants are twice the number of subjects since each subject includes two participants. That is, the total number of participants is 108 divided into 80 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, of the following two rounds refactored a Chess game, and of the last two rounds refactored a Telecom system. We describe the software systems and the refactored concerns in Section 2.3.

Table 1. Number of subjects in each round

System	Round	# of Subjects	Grouping
ATM	First	11	Pairs
	Second	17	Individually
Chess	Third	15	Pairs
	Fourth	15	Individually
Telecom	Fifth	3	Pairs
	Sixth	19	Individually
Total number of subjects		**80**	

The participants filled in a questionnaire with their background information. The answers regard participants' level of knowledge in Object-Oriented Programming (OOP) and their work experience in software development. This questionnaire aims to characterize the background of the study participants. In general, more than 80% of participants claimed to have some experience in OOP. Since most participants are undergraduate or graduate students, about half of them have never worked in a software development company. Due to space constraints, we provide detailed information of the participants' backgrounds in the project website [1].

2.3 Target Applications and Crosscutting Concerns

Three small Java applications were chosen to be used in this study: ATM, Chess, and Telecom. Each participant were asked to refactor one crosscutting concern of each application using the AspectJ programming language [18]. 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) [12, 14] and Lines of Concern Code (LOCC) [12, 14] measure the concern size using the same units. All concerns and applications share similar complexity and were carefully chosen to allow subjects to finish the experimental task within 90 minutes. 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 [18].

Table 2. Size metrics of the target applications and concerns

Application/ concern	**Application size**		**Concern size**	
	NC	**LOC**	**CDC**	**LOC**
ATM / Logging	12	606	4	19
Chess / ErrorMessages	13	1011	8	36
Telecom / Timing	8	213	4	32

The ATM application simulates three basic functionalities of an ordinary cash machine: show balance, deposit, and withdraw. 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. The Chess application implements a chess game with a graphical user interface. 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. This concern is implemented by 36 LOC spread over 8 classes. Finally, Telecom is a connection management system for phone calls implemented in Java and AspectJ [2]. It simulates a call between two or more customers. We only used the Java implementation and asked each subject of the last two 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.

2.4 Experimental Tasks

Before performing the experimental tasks, all participants attended a 2-hour training session about AOP and AspectJ. 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 constructs. Note that the subjects were responsible for both correctly identifying the concern code in the application as well as choosing the proper AspectJ constructs to refactor the given crosscutting concern. No instruction was given in this sense because we consider these tasks part of the AOP programmer regular duties. After subjects concluded their tasks, we performed our analysis in two steps. First, we investigated which were the categories of AOP-specific mistakes the programmers frequently made (Section 3). Then, we identified situations in the OO original code, named code pitfalls (Section 4), that may lead programmers to make mistakes.

3 Recurring Mistakes in AOP

This section summarizes the results of our study and classifies the mistakes recurrently made by subjects during the experimental tasks. This classification is based on our previous work [4]. Mistakes were identified by manual code inspection.

3.1 Classification of Recurring Mistakes

Incorrect Implementation Logic. It occurs when a piece of concern code is aspectized through pointcut and advice mechanisms. However, the refactored code behaves differently from the original code. For instance, by selecting an option in the main menu of ATM, a logging report of that operation can be displayed on screen. However, novice programmers often make this kind of mistake because it involves an intricate concern code nested in switch and while constructs.

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

Compilation Error or Warning. It refers to either a syntactic fault or a potential fault signaled by the AspectJ compiler through an error or warning message[1]. For instance, in the OO version of the ATM application, the logging message that records the user authentication is stored when the `authenticateUser()` method of the `BankDatabase` class is executed. Although this method has two integer parameters, some subjects have not specified the method parameters neither used a wildcard when writing the pointcut to select this join point. Consequently, a warning message notifies the programmer that no join point matched this specific pointcut.

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 ErrorMessages and should be refactored to aspects in the Chess application. However,

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

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.

Incomplete Refactoring. This mistake occurs when developers fail to refactor part of the crosscutting concern. As a result, 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 in the `Costumer` class that implements part of Timing in the Telecom application. When refactoring this concern, some subjects have not transferred this attribute to an aspect.

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 `Connection` to an advice. However, some subjects also refactored the previous statement: `System.out.println("connection completed")`. Although this statement also appears in the `complete()` method, it should not be refactored since it does not belong to the Timing concern.

3.2 Experimental Results

Our analysis is based on 80 AOP implementations of the three target applications (Section 2.3). AOP-related mistakes in these implementations were analyzed and classified according to the categories discussed in Section 3.1. Figure 1 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).

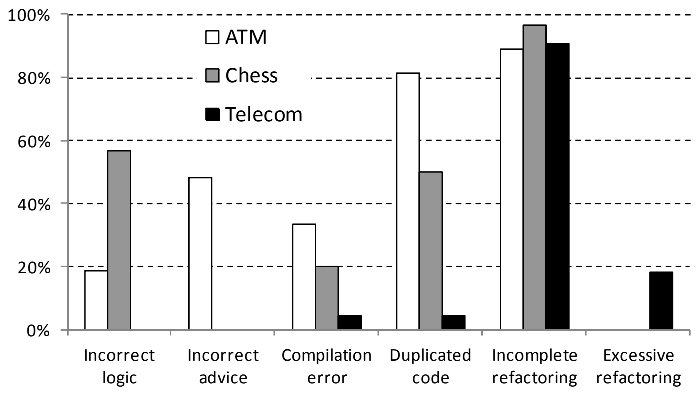


Fig. 1. Subjects that made mistakes in each category

Figure 1 shows that Incomplete Refactoring is the most common mistake in all applications. This mistake occurred with fairly the same rate (about 95%) in all applications. Unlike Incomplete Refactoring, some mistakes are more common in one application than in the others. For instance, Duplicated Crosscutting Code and Incorrect Advice Type are common in the ATM application. Incorrect Implementation Logic, on the other hand, is more frequent in the Chess application. This result is mainly due to (i) the 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 the Chess and Telecom applications. After inspecting the concern code, we observed that all advices used to aspectize ErrorMessages (Chess) and Timing (Telecom) should be after advices. Therefore, due to the homogeneity of the concern code, it is unlike that someone would use a different type of advice and make this kind of mistake in these two concern instances.

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.* [25] highlighted that the inability to identify the concern code is one of the most problematic steps for concern refactoring. We empirically confirm their results. Regarding the inability to separate concern code, we found out that programmers avoid refactoring pieces of code if such code is very tangled; in such cases, programmers end up leaving it mixed up with the base code.

The Excessive Refactoring mistake only happened in the Telecom application (see Figure 1). We observed that this kind of mistake seems to be less often in software aspectization. In fact, the particular way Timing is implemented in Telecom might have led programmers to make Excessive Refactoring. For instance, Figure 2 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 concern. However, some programmers moved this method from the `BasicSimulation` class to an aspect and introduced it back by means of inter-type declaration; 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.

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

Fig. 2. Timing implementation in the BasicSimulation class

4 Aspectization Code Pitfalls

After collecting the mistakes made by the subjects of the experiment described in Section 3, we manually inspect situations that could have led them to recurrently make these mistakes. We name these situations *code pitfalls* and analyzed them in two phases. Firstly, we performed a manual inspection in the OO source code of the applications in order to identify code pitfalls for AO refactoring. Secondly, 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 catalogue of code pitfalls described in this section. We organized these code pitfalls into two major categories: Dedicated Implementation Elements (Section 5.1) and Non-Dedicated Implementation Elements (Section 5.2).

4.1 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.* [6] 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 had to refactor dedicated elements. In particular, the mistakes 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.* [25] 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.

```
public class ATM {
   private static final int LOG= 0;
   ...
}
```

Fig. 3. Primitive Constant code pitfall

Primitive Constant. The first code pitfall, named Primitive Constant, occurs when a constant is dedicated to implement the concern. For instance, Figure 3 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 [10]. 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 has to do with the type of a dedicated implementation attribute. In this case, the attribute has either a primitive type (e.g., boolean, int, long) of a non-dedicated element type. For instance, the Timing concern in Telecom is implemented by two attributes. One of them, shown in Figure 4.a, is of the type Timer which is completely dedicated to the concern. Subjects of the experiment had no problem to locate and refactor this attribute to an aspect. On the other hand, Figure 4.b shows the `totalConnectionTime` attribute, declared as a long data type, which is not dedicated to Timing; it is actually of 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 such attribute, but none of them failed to refactor the `timer` attribute (Figure 4.a). We observed that developers appear to have difficulty to assign attributes of non-dedicated types 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 attributes in the concern code leads programmers to make the Incomplete Refactoring mistake.

```
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. 4. Attribute of the Non-Dedicated Type code pitfall

4.2 Non-dedicated Implementation Elements

Many mistakes 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 AOP languages. Three code pitfalls are related to non-dedicated implementation elements: *Disjoint Inheritance Trees*, *Divergent Join Point Location*, 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 tree usually does not refer to the elements of the other. 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 tend to follow the same pattern in the concern implementation. Figure 5.a 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. This mistake was very recurring while the experiment subjects aspectized the ErrorMessages in the Chess application.

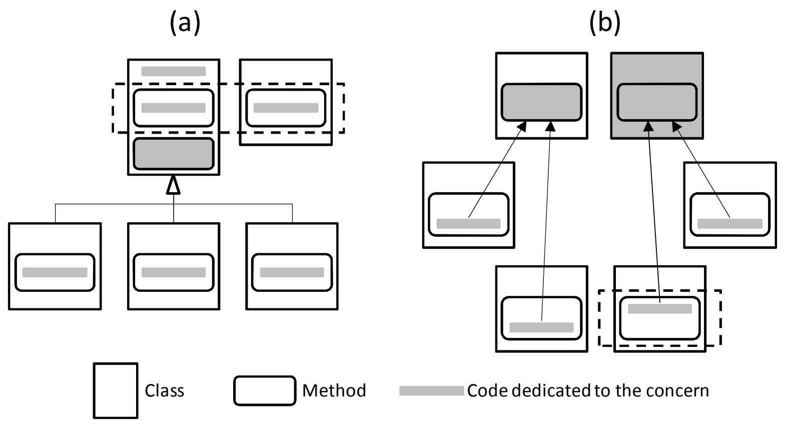


Fig. 5. Disjoint Inheritance Trees and Divergent Join Point Location code pitfalls

Divergent Joint Point Location. This code pitfall occurs when (i) pieces of the concern code are 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. Figure 5.b illustrates this code pitfall. In this figure, the concern code is located at the end of three 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 joint point location usually tricks developers and they end up making the Incorrect Advice Type mistake.

As an example of this code pitfall, the Logging concern 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 refactored using a before advice. All other calls to `log()` appear in the end of a method and should be refactored using after advice. Thus, developers should use the appropriate advice type - before or after in this case - to refactor the Logging concern. We noticed that many subjects of the first and second rounds refactored all of the Logging 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. Subjects of the other four rounds have not made this type of mistake because ErrorMessages (Chess) and Timing (Telecom) have a uniform implementation. That is, all parts of the concern should be refactored using the same advice type.

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 [8] has advocated that the dependence on local variables makes the concern code harder to refactor, since the join point model 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.

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

Fig. 6. Incorrect Implementation Logic related Logging

To illustrate this code pitfall, we rely on a piece of the Logging concern (ATM) shown in Figure 6. 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 Figure 6 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 Figure 6, 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.

5 Tool Support

This section presents ConcernReCS , an Eclipse plug-in to find the aspectization code pitfalls discussed in Section 4. The tool is based on static analysis of Java code and builds on knowledge of our experiment results. ConcernReCS extends ConcernMapper [29] which is a tool to allow the mapping of methods and attributes to concerns. Figure 7 describes a simplified flow chart of ConcernReCS. ConcernReCS receives information from both ConcernMapper and Eclipse. The Data Analyzer module relies on this information to generate warnings of code pitfalls.

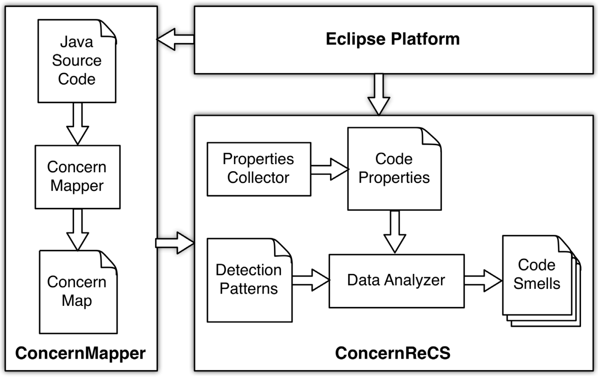
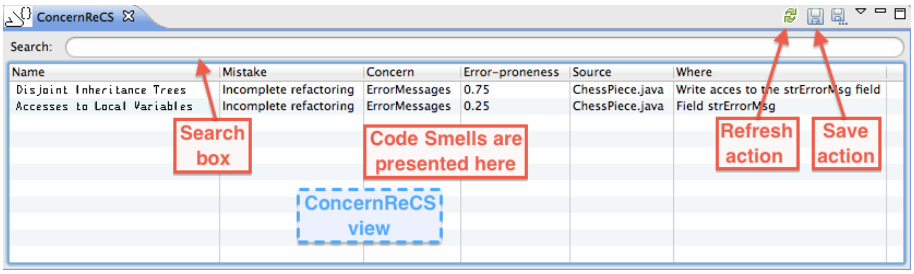


Fig. 7. ConcernReCS simplified flow chart

The first step to use ConcernReCS is to manually map the dedicated implementation elements (i.e. methods, classes and attributes) of one or more concerns using ConcernMapper. 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. Concern code of any other kind (e.g. conditional statements), should be extracted by the Extract Method refactoring [16] and, then, the resulting method should be mapped to the concern.

Figure 8 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, each of which representing approximately the percentage of subjects who made mistakes related to the code pitfall (the value of 1 means that all subjects in our experiment made mistakes related to a specific code pitfall).



Name	Mistake	Concern	Error-proneness	Source	Where
Disjoint Inheritance Trees	Incomplete refactoring	ErrorMessages	0.75	ChessPiece.java	Write acces to the strErrorMsg field
Accesses to Local Variables	Incomplete refactoring	ErrorMessages	0.25	ChessPiece.java	Field strErrorMsg

Fig. 8. The ConcernReCS main view

6 Study Limitations

The AspectJ language can be pointed out as a limitation in our study. However, it is widely known that AspectJ-based systems have been used in the large majority of AOP assessment studies. Even though there are other programming techniques which are able to realize the AOP concepts (examples are Intentional Programming, Meta Programming and Generative Programming [15]), so far AspectJ has been the mainstream AOP technology within both industrial and academic contexts [27].

Another limitation regards the size and representativeness of the target applications and the selected crosscutting concerns to be refactored. Regarding the size of the applications, they indeed do not reflect the complexity observed in the industrial context. However, this type of experiment requires tasks being performed within a short pre-specified period of time. Furthermore, applications of similar size have been used in recent programming-related experiments with different goals within varied contexts [7, 20]. The crosscutting concerns are of different nature but with similar complexity in order to pose balanced implementation scenarios in terms of difficulty.

The number and representativeness of the subjects can also be considered a limitation of this study. Around 50% of the experiment participants declared to have moderate to high level of knowledge in OOP. Despite of the observed heterogeneity, all subjects made similar types of mistakes while performing the experiment tasks. Note that their level of expertise in AOP was indeed not expected to be high, since the experiment was designed to be performed by novice programmers in AOP.

7 Related Work

Investigations of fault types and faulty scenarios in AO software: Previous studies in AOP-specific fault types and bug patterns [3, 5, 9, 11] address from the most basic concepts of AOP to advanced constructions of AspectJ-like languages. In previous research, Ferrari *et al.* [10] analyzed the existing taxonomies and catalogues and identified four main categories of faults that can be found in AO software. According to them, faults can be associated with pointcut expressions (i.e., quantification mechanisms), inter-type declarations and other declare-like expressions (i.e., introduction mechanisms), advice signatures and bodies (i.e., the crosscutting behavior), and in the intercepted (i.e., base) code. When we compare the contributions of this paper to the results presented by Ferrari *et al.* [10], we can spot one major difference. We characterize mistakes that novice programmers are likely to make while refactoring crosscutting concerns. Ferrari *et al.*, on the other hand, characterize faults that can be revealed during the testing phase, be them refactoring-related or not, which shall be prioritized in testing strategies.

Burrows *et al.* [7] analyzed how faults are introduced into existing AO systems during adaptive and perfective maintenance tasks. The AO version of the Telecom system was used as a basis for a set of maintenance tasks performed by experienced, paired programmers. The authors then applied a set of coupling and churn metrics in order to figure out the correlation between these metrics and the introduced faults.

Our study differs from the Burrows *et al.* one 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.

Previous studies about AOP-specific mistakes [3, 9, 10, 11] range from the most basic concepts of AOP to advanced constructions of AspectJ-like languages, such as intertype declarations. Since our study was conducted only with novice programmers in AOP, our classification partially overlaps previous 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 the two most basic features of AOP languages: pointcut and advice.

Investigations of concern identification and code smells: Recent studies [12, 25] investigated mistakes in the projection of concerns on code – a key task in concern refactoring. For instance, Figueiredo *et al.* [12] 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.* [25] characterized a set of recurring concern mapping mistakes made by software developers. Amongst the causes that led to mistakes in the mapping tasks, Nunes *et al.* highlight the crosscutting nature – i.e., spreading or tangling – of several concerns they analyzed. This may help us to explain the high number of refactoring mistakes observed in our study: the participants had to first identify which parts of the code referred to the target crosscutting concern. That is, they had to map the target concern in the code. Then, they should perform the refactoring step.

Macia *et al.* [21] report on the results of an exploratory study of code smells in evolving AO systems. They analyzed 18 releases of three medium-sized AO systems to assess previously documented AOP code smells 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 4. 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.

8 Conclusions and Future Work

This paper presented the results of a series of experiments whose main goal was to characterize mistakes made by novice programmers in typical crosscutting concern refactorings. In this study, we revisit recurring mistakes made by programmers with specific backgrounds (Section 3). 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 proposed a catalogue of aspectization code pitfalls (Section 4). 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 5).

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 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. This shall allow us to perform more comprehensive analysis with statistical significance. Moreover, we aim to evaluate whether the proposed tool (ConcernReCS) is effective to advise programmers about code pitfalls and help them avoiding typical refactoring mistakes.

Acknowledgments. This work was partially supported by CNPq (grants Universal 485907/2013-5 and 485235/2013-7) and FAPEMIG (grants APQ-02532-12 and PPM-00382-14). Fabiano also thanks the financial support received from Federal University of São Carlos (RTN grant).

References

1. Results of Avoiding Code Pitfalls in Aspect-Oriented Programming (2014), `http://www.dcc.ufmg.br/~figueiredo/aop/mistakes`
2. AJDT: AspectJ Development Tools, `http://www.eclipse.org/ajdt/`
3. Alexander, R.T., Bieman, J.M., Andrews, A.A.: Towards the Systematic Testing of Aspect-Oriented Programs. Colorado State University, TR CS-04-105 (2004)
4. Alves, P., Santos, A., Figueiredo, E., Ferrari, F.: How do Programmers Learn AOP: An Exploratory Study of Recurring Mistakes. In: LA-WASP 2011 (2011)
5. Baekken, J., Alexander, R.: A Candidate Fault Model for AspectJ Pointcuts. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 169–178 (2006)
6. Brinkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P.: Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. IEEE Transactions on Software Engineering (TSE) 32(17), 698–717 (2006)
7. Burrows, R., Taiani, F., Garcia, A., Ferrari, F.C.: Reasoning about Faults in Aspect-Oriented Programs: A Metrics-based Evaluation. In: ICPC, pp. 131–140 (2011)
8. Castor Filho, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C., Amorin, J., Silva, H.: On the Modularization and Reuse of Exception Handling with Aspects. Software: Practice and Experience 39(17), 1377–1417 (2009)
9. Coelho, R., Rashid, A., Garcia, A., Ferrari, F., Cacho, N., Kulesza, U., von Staa, A., Lucena, C.: Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 207–234. Springer, Heidelberg (2008)
10. Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Maldonado, J.: Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice. In: Brazilian Symposium on Software Engineering (SBES), pp. 50–59 (2010)
11. Ferrari, F., Maldonado, J., Rashid, A.: Mutation Testing for Aspect-Oriented Programs. In: International Conference on Software Testing (ICST), pp. 52–61 (2008)
12. Figueiredo, E., Garcia, A., Maia, M., Ferreira, G., Nunes, C., Whittle, J.: On the Impact of Crosscutting Concern Projection on Code Measurement. In: AOSD, pp. 81–92 (2011)
13. Figueiredo, E., et al.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: Int'l Conference on Software Engineering (ICSE), pp. 261–270 (2008)
14. Figueiredo, E., et al.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In: CSMR, pp. 183–192 (2008)

15. Filman, R.E., Friedman, D.: Aspect-Oriented Programming is Quantification and Obliviousness. Aspect-Oriented Software Development, pp. 21–35. Addison-Wesley (2004)
16. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
17. IEEE Standard Glossary for Software Engineering Terminology. Standard 610.12, Institute of Electrical and Electronic Engineers, New York, USA (1990)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Lemos, O., Ferrari, F., Silveira, F., Garcia, A.: Development of Auxiliary Functions: Should You Be Agile? An Empirical Assessment of Pair Programming and Test-First Programming. In: ICSE, pp. 529–539 (2012)
21. Macia, I., Garcia, A., von Staa, A.: An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In: AOSD, pp. 203–214 (2011)
22. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall (2000)
23. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: AOSD (2003)
24. Monteiro, M.P., Fernandes, J.M.: Towards a Catalogue of Refactorings and Code Smells for AspectJ. In: Rashid, A., Akşit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 214–258. Springer, Heidelberg (2006)
25. Nunes, C., Garcia, A., Figueiredo, E., Lucena, C.: Revealing Mistakes in Concern Mapping Tasks: An Experimental Evaluation. In: CSMR, pp. 101–110 (2011)
26. Piveta, E., Hecht, M., Pimenta, M., Price, R.: Detecting Bad Smells in AspectJ. Journal of Universal Computer Science 12(7), 811–827 (2006)
27. Rashid, A., et al.: Aspect-Oriented Programming in Practice: Tales from AOSE-Europe. IEEE Computer 43(2), 19–26 (2010)
28. Robillard, M., Murphy, G.: Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology (TOSEM) 16, 1 (2007)
29. Robillard, M.P., Weigand-Warr, F.: ConcernMapper: Simple View-based Separation of Scattered Concerns. In: OOPSLA Workshops, pp. 65–69 (2005)
30. Soloway, E., Ehrlich, K.: Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering (TSE), 595–609 (1984)
31. Spohrer, J., Soloway, E.: Novice Mistakes: Are the folk Wisdoms Correct? Communications of the ACM 29(7) (1986)
32. Munhoz, F., et al.: Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study. In: International Conference on Software Maintenance (ICSM), pp. 137–146 (2009)