Computer Science and Engineering 2014, 4(1): 7-16
DOI: 10.5923/j.computer.20140401.02

Refactoring and AOP under Scrutiny

Sandra Casas[*], **Cecilia Fuentes Zamorano**

GISP, Instituto de Tecnología Aplicada, Universidad Nacional de la Patagonia Austral, Río Gallegos, 9400, Argentina

Abstract A conflict between refactoring and AOP techniques can arise whenever an application with aspects is subjected to a refactoring application. When the units of code being restructured are also part of a pointcut definition, changes in the external behaviour of the application can occur. This study presents an approach to anticipate the impact of refactoring changes in AO applications. We first decompose refactorings into atomic change operations. Then we individually analyse and evaluate each of these operations. The overall results anticipate the consequences of the refactoring. Our approach is partially automated. We also provide some examples and introduce a discussion (left open in this work) about the relationship between different variables that characterize the refactorings.

Keywords Aspect-Oriented Software Development, Refactoring, Software Evolution, Change Impact Analysis, AspectJ

1. Introduction

The process of changing a software system in a way that improves its internal structure yet does not alter the external behaviour of the code yet is called “restructuring”[1]. Refactoring[2] is the object-oriented variant of restructuring. The key idea is to redistribute classes, variables, and methods across the class hierarchy in order to facilitate future adaptations and extensions[3]. In the context of software evolution, restructuring and refactoring are used to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability, and efficiency).

Refactoring is not only a theoretical technique; it has also won acceptance in real practice, with several commercial and noncommercial tools supporting refactoring, such as Eclipse, IntelliJ, NetBeans, and JBuilder. These tools automatically perform a set of refactorings for any programming language, such as Java.

Aspect-Oriented Programming (AOP)[4] is a technique that provides for separation of concerns[5]. AOP proposes a new kind of modularization called *aspects*. An aspect is a module that can localize the implementation of crosscutting concerns (CCC)[6]. The main dynamic abstractions of an aspect are pointcuts and advice. Pointcuts are predicates that describe a set of join points where an advice code should be executed. A join point is a well-defined point in a program’s control flow. The key to the AOP modularization technique lies in its composition mechanism called *weaving* [7]. In traditional approaches such as OO, subroutines explicitly invoke the behaviours implemented by other subroutines. In contrast, aspects have an implicit invocation mechanism, so that the behaviour of an aspect is implicitly invoked in the implementation of other modules. Consequently, the implementation of these other modules can be largely unaware of the CCC. However, this structure (pointcuts and advice) makes it difficult for developers to evaluate the behaviour of a system. In particular, the implicit invocation mechanism introduces an additional layer of complexity in the construction of a system. This can make it difficult to understand how and when the base system and the aspects interact, and consequently, how the overall system will behave.

As suggested in[8], a conflict between refactoring and AOP techniques can arise whenever an application with aspects is subjected to a refactoring application. When the units of code under restructuring are also pieces of pointcut definitions, changes in the external behaviour of the application can occur. If the refactoring unintentionally causes links or connections (generated by the aspect weaving process) between the base code and the aspects to disappear or appear, the impact is not evident to the developer. Such situations can arise because of the tight coupling and dependency among aspects and classes and the fact that refactoring is an invasive technique.

The present study answers some questions raised in[8]. We propose an approach to anticipate the impact of refactoring changes in AO applications. Our proposal first decomposes the refactorings into atomic change operations. Then we analyse and evaluate each of these individually, and use the overall results to anticipate the consequences of the refactoring. Our approach is partially automated. The central software artifact in this study is source code, and for this purpose we have focused on the AspectJ language[9]. However, our approach may be applied to any other AO

* Corresponding author:
scasas@unpa.edu.ar (Sandra Casas)
Published online at http://journal.sapub.org/computer

language.

This study comprises two parts. The first part presents our approach and some examples, and the second part uses descriptive statistics to discuss the relationships between different variables involved in the application of refactorings to AO systems. The remainder of the paper is organised as follows. Section 2 presents a simple motivating example for the problem. Section 3 introduces the model that will be used to anticipate the consequences of refactorings in AO applications. Section 4 explains how the predictions are made and introduces the tool BaLaLu, followed by some examples. Section 5 uses descriptive statistics to discuss the relationships between certain variables that are involved in the application of refactoring in AO systems. Section 6 describes some related work, and Section 7 presents our conclusions.

2. Motivating Example

As we have explained, pointcuts and advice provide the AOP mechanism that encapsulates CCC by changing the dynamic execution of the base code. Advice is a fragment of code, such as a method, that will be executed with the base code (before, after, or around). A pointcut is an expression that establishes the events and conditions specifying when and where advice code will be executed, typically as a method call. Pointcuts are the more critical elements in AOP evolution, because a simple change in the base code may alter the set of join points of any pointcut and thus have consequences for advice execution. Pointcuts can refer to events either explicitly or by using defined pattern names with wildcards.

A well-known refactoring is the Pull Down Method, as documented in[2]. This refactoring involves moving one or more methods of a superclass to a subclass; it is recommended when behaviour of a superclass is relevant for only some of its subclasses. A problem can arise when an aspect intercepts the method that refactoring pulled down. In Figure 1, the *LogQuota* aspect intercepts all calls of the *getQuota* method of the *Employee* class, and it records some information in a log. Then, when the *getQuota* method is moved to the *Salesman* subclass, the *changeQuota* pointcut is no longer valid. After application of Pull Up Method refactoring, the behaviour of the *LogQuota* aspect is not linked because the join point does not exist—that is to say, the join point is broken.

```
public aspect LogQuota {
  pointcut changeQuota(..):
       call(* Employee.setQuota(..int))&& target(p);
            after(Employee p): changeQuota(p) {
                 Logger.writeLog("Change Quota:”
                 +p.toString());    }
}
```

Some change operations, such as adding a class, removing a method or field, or renaming a method, are important because they can generate diverse nonlocal consequences in AO applications. An elementary change in the base code can produce potential false positives/negatives. After a change operation has been applied, a pointcut may either capture too many join points (false positives) or fail to capture certain join points that were intended to be captured (false negatives). Refactorings are very complex change operations because they comprise a set of different change operations. Thus, a refactoring can generate a set of false positives/ negatives.

When this happens, developers must identify the problem and resolve it. However, the identification of false positives/negatives and their causes is not a trivial task in medium-scale applications. This analysis is even more difficult when it is performed after the source code has been modified and particularly when it was automatically modified by a tool. Developers must perform several tasks such as exhaustive code analysis and inspection and intensive execution of test cases. All these tasks impact maintenance time and effort, increasing the maintenance costs. Therefore, new methods and tools are necessary to reduce the maintenance time, effort, and costs.

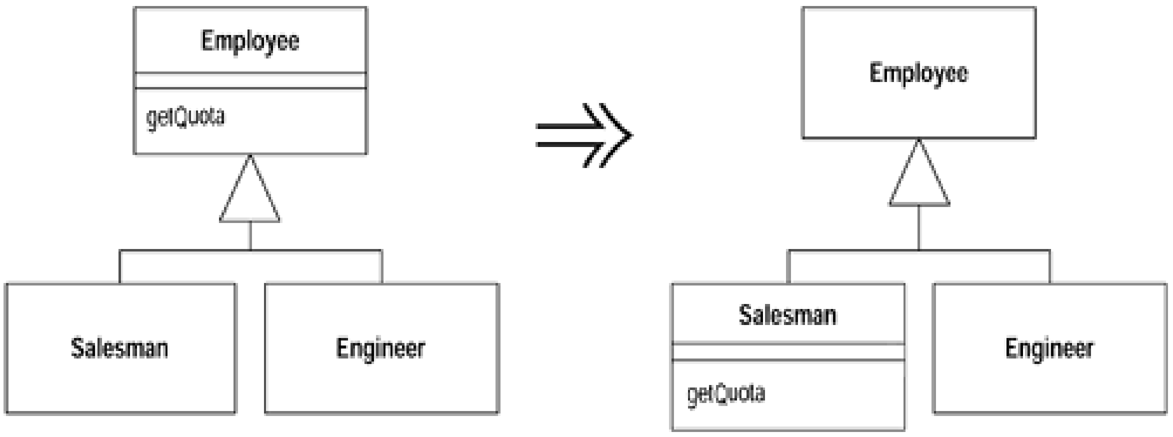


Figure 1. LogQuota aspect and Pull Down Method refactoring

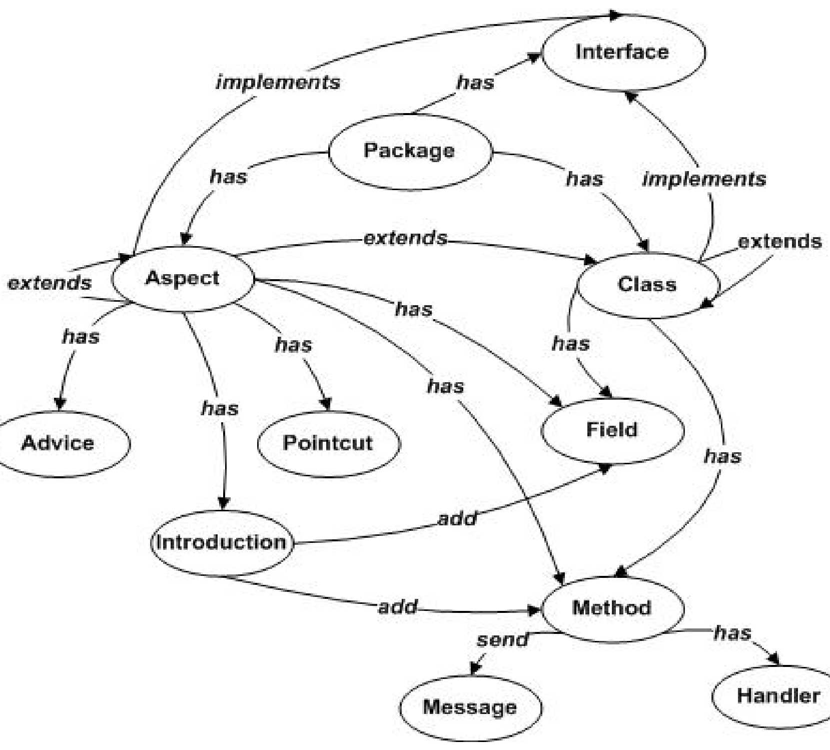


Figure 2. Entities and relationships in the repository

3. Overview of Anticipation Model

Change operations are the core of software evolution. A change operation can be simple (atomic), such as adding a field in a class, or very complex, as in refactoring. Our model's underlying idea for anticipating the consequences of refactoring in AOP is to anticipate the consequences of atomic change operations, as follows. i) Identify the consequences of change operations in AO applications. This implies the possibility of detecting the effects of change operations on the source code. ii) Quantify the consequences of change operations in metrics that facilitate the analysis for developers. This implies the possibility of quantitatively measuring the false positives/negatives that a change operation may produce. iii) Identify where the consequences occur and relate them to the quantified information. This implies the possibility of delimiting the segments of source code that may be affected by a change operation.

Our model comprises three main components: program repository, identification of change operations, and identification of their consequences.

3.1. Program Repository

In contrast to CVS or Subversion repositories that manage text files, our approach represents programs (classes and aspects) as entities and relationships. Since we focus on AO applications, we consider constructs such as packages, classes, methods, fields, aspects, pointcuts, advice, and exception handlers. We also represent different relationships among these entities that are relevant for AO, such as inheritance, method calls, and aspect weaving/compositions. Each entity has several properties and states such as identifier, type, and access modifier. These properties and states identify and represent entities in the repository and the relationships between them. Pointcuts are represented in two ways, as expressions and as sets of join points intercepted in specific instances. The program repository has semantic and syntactic information about the program, but it does not store the source code or text. Figure 2 represents the main entities and relationships in the repository.

3.2. Change Operations and Consequences

A change operation is a function whose inputs produce specific outputs over a specific instance of the repository. Usually, the "add" change operations can generate potential false positives, the "remove" change operations can generate potential false negatives, and the "move" and "rename" change operations can generate potential false positives and negatives. For example, the change operation "remove class X" impacts all designators of pointcuts that refer to class X. That is, the join point expressions of a primitive pointcut designator include "call", "execution", "target", "within", and so on, and if X is referenced in any of these expressions, then a potential false negative is present.

In general, we say

if (ChOp(x) && P(x)) then[C],

where ChOp is any change operation, P is any pointcut of the application, x is a source code entity (package, class, method, field, etc.), and C is the set of consequences of ChOp (false positives/negatives). In general, when we refer to the consequences, we describe them as "potential" false positives/negatives. This is because not all impacts are bad or problematic. The significance of the result should be evaluated by the developer, according to the software requirements and objectives.

A change operation can be atomic or composite. Atomic change operations are indivisible operations that cannot be separated into more than one task or step; thus, they are very simple. An atomic change operation contains all the necessary information to represent a function that can be analysed with the repository information. An atomic change operation can produce false positives/negatives during system evolution. A composite change operation is a set of atomic change operations, such as move or rename an entity.

In Table 1, we have analysed the potential consequences (impacts) of a repertory of change operations to which code units or entities are "sensitive" in the AOP context, specifically in pointcut expressions of the AspectJ language. That is, we only examine change operations that can alter pointcut definitions. AspectJ provides several pointcut descriptors that identify groups of join points that meet different criteria. These descriptors are classified into different groups as follows.

Table 1. Change Operation Analysis and Consequences (Impacts)

Change Operation	**Type**	**Group**						**PFP**	**PFN**
		G1	**G2**	**G3**	**G4**	**G5**	**G6**		
Add package	A	X	X	X	X	X	X	+	
Add class	A	X	X	X	X	X	X	+	
Add method	A	X	X	X	-	-	-	+	
Add field	A	X	X	X	-	-	-	+	
Add handler	A	X	X	X	-	-	-	+	
Add message	A	X	X	X	-	-	-	+	
Remove package	A	X	X	X	X	X	X		+
Remove class	A	X	X	X	X	X	X		+
Remove method	A	X	X	X	-	-	-		+
Remove field	A	X	X	X	-	-	-		+
Remove message	A	X	X	X	-	-	-		+
Remove handler	A	X	X	X	-	-	-		+
Move package	C	X	X	X	X	X	X	+	+
Move class	C	X	X	X	X	X	X	+	+
Move method	C	X	X	X	-	-	-	+	+
Move field	C	X	X	X	-	-	-	+	+
Move handler	C	X	X	X	-	-	-	+	+
Move message	C	X	X	X	-	-	-	+	+
Rename Package	C	X	X	X	X	X	X	+	+
Rename class	C	X	X	X	X	X	X	+	+
Rename method	C	X	X	X	-	-	-	+	+
Rename field	C	X	X	X	-	-	-	+	+

Based on the categories of join points (G1): these capture join points according to the category to which they belong, i.e., call, execution, get, set, handler, staticinitialization, initialization, preinitialization, adviceexecution.

Based on the control flow (G2): these capture join points of any category as long as they occur in the context of another pointcut, i.e., cflow and cflowbelow.

Based on the location of code (G3): these capture join points of any category that are located in certain fragments of code, for example, within a class or within the body of a method, e.g., within and withincode.

Based on run-time objects (G4): these capture join points whose current objects (this) or objects (target) are of a certain type.

Based on the arguments of the join points (G5): these capture join points whose arguments are of a certain type, using the "args" descriptor.

Based on conditions (G6): these capture join points based on some condition using the "if" descriptor.

The change operations cited in the G1 group have the greatest impacts (consequences), because their semantics always refer to program elements such as class identifier, method, or attribute. Also, every pointcut must be defined around a designator of this group. By contrast, operations in the G6 group turn out to be much less used, and although they might be mentioned in the definition of a program element, such use is not mandatory or frequent.

4. Anticipation of Refactoring Consequences in AO Applications

Refactoring is a code restructuring discipline. Every refactoring includes an application's sequential steps. These steps can be analysed as atomic change operations. For example, a move method can be analysed as two atomic change operations: "remove method" and "add method". The main idea is to treat a refactoring as a composite change operation that can be decomposed into a set of atomic change operations**.**

For example, the steps of the "Extract Class" refactoring are:

1. Decide how to split the responsibilities of the class.
2. Create a new class to express the split-off responsibilities.
3. Make a link from the old class to the new class.
4. Apply "Move Field" to each field you wish to move.
5. Compile and test after each move.
6. Use "Move Method" to move methods over from the old class to the new class.
7. Compile and test after each move.
8. Review and reduce the interfaces of each class.
9. Decide whether to expose the new class.

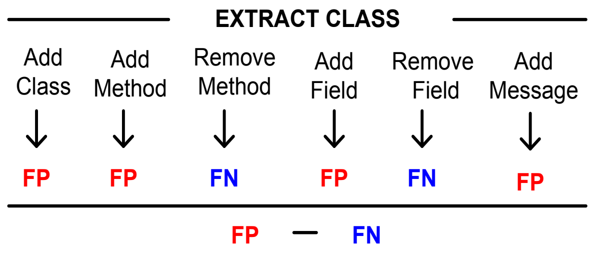
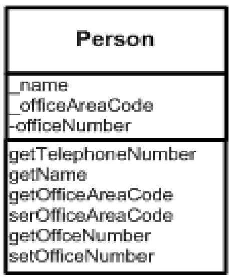


Figure 3. "Extract Class" refactoring decomposition



```
AC(TelephoneNumber)    (1)

RF(_officeAreaCode Person)                  }
AF(_officeAreaCode TelephoneNumber)         }  (2-3)
RM(getOfficeAreaCode Person)                }
AM(getOfficeAreaCode TelephoneNumber)       }  (4-5)
RM(setOfficeAreaCode Person)                }
AM(setOfficeAreaCode TelephoneNumber)       }  (6-7)

RF(_officeNumber Person)                    }
AF(_officeNumber TelephoneNumber)           }  (8-9)
RM(getOfficeNumber Person)                  }
AM(getOfficeNumber TelephoneNumber)         }  (10-11)
RM(setOfficeNumber Person)                  }
AM(setOfficeNumber TelephoneNumber)         }  (12-13)

AF(_telephone Person)             (14)
AS(_getOfficeAreaCode TelephoneNumber          }
    getTelephoneNumber Person )                }
AS(_getOfficeNumber TelephoneNumber            }  (15-16)
    getTelephoneNumber Person)                 }
```

```
public class Person {
  private String _name;      (2-8)
  private TelephoneNumber telephone;        (14)

  public String getName() {  (4-6-10-12)
   return _name;   }
  public String getTelephoneNumber() {
   return ("("+telephone.get_officeAreaCode()+")"   (15-16)
               +telephone.getOfficeNumber());   }
}
public class TelephoneNumber
  private String _officeAreaCode; (1) }    (3-9)
  private String _officeNumber;       }

public String getOfficeAreaCode() {
  return _officeAreaCode;   }                      (5 -7)
public void setOfficeAreaCode(final String arg) {
   _officeAreaCode = arg;   }

public String getOfficeNumber() {
  return _officeNumber;   }                    (11-13)
public void setOfficeNumber(String arg) {
  _officeNumber = arg;       }
}
```

Figure 4. "Extract Class" refactoring decomposition into atomic change operations

Step 2 is the "add class" atomic change operation. Step 4 is a set of pairs of "add field" and "remove field" atomic change operations. Step 6 is similar. Graphically, the "Extract Class" refactoring's decomposition into atomic change operations is shown in Figure 3; here the potential consequence of each single atomic change operation can be anticipated.

Following this scheme and using the practical notation in Table 2, the refactoring decomposition can be expressed in atomic change operations. Only atomic change operations that can be impacted in a pointcut definition will be identified. This set of atomic change operations, which we call "relevant", is enumerated in Table 1. Other atomic change operations exist, such as "remove a local variable", but this atomic change operation is not relevant in the AOP context because a local variable cannot be reached by a pointcut definition.

Table 2. Notation Used to Decompose Refactorings

Expression	Description
AC(c)	Add class c
AM(m c)	Add method m to class c
AF(f c)	Add field f to class c
AH(h m c)	Add exception handler h to method m of class c
AS(s m c)	Add message s to method m of class c
RC(c)	Remove class c
RM(m c)	Remove method m from class c
RF(f c)	Remove field f from class c
RH(h m c)	Remove exception handler h from method m of class c
RS(s m c)	Remove message s from method m of class c

Figure 4 presents a concrete typical example of decomposing an "Extract Class" refactoring into atomic change operations. Expressions (14-16) are the atomic change operations necessary to establish the relationship between two classes. The entire set of atomic change operations (1-16) can be evaluated to discover the consequences of refactoring. The evaluation is automatically performed with the BaLaLu tool.

The *_officeAreaCode* and *_officeNumber* fields and their corresponding set/get methods will be moved to the *TelephoneNumber* class. Expression (1) represents the "Add Class" atomic change operation, the first step of this refactoring. Next, the refactoring indicates moving features between the two classes. Expressions (2-3) are the decomposition of the "Move Field" action, as are expressions (8-9). Expressions (4-5) represent the decomposition of "Move Method", as do the pairs of expressions (6-7), (10-11), and (12-13).

4.1. BaLaLu Tool

We have developed the BaLaLu (*Bajo La Lupa*, "Under Scrutiny" in Spanish) tool to support our approach to predicting the impact of refactoring changes. BaLaLu supports both Java and AspectJ source code. The main aspects of the design and implementation of BaLaLu that we will discuss here are how change operations (atomic operations and refactorings) are represented and how they interact with the repository.

The atomic change operations comprise a hierarchy in which AddClass, AddPackage, AddMethod, AddField, AddMessage, RemoveClass, RemoveMethod, RemoveField, RemoveMessage, and so on, are subclasses of the AtomicChangeOperation class. The refactoring class is a container of AtomicChangeOperation objects. Each atomic change operation has its own consequences (potential false positives/negatives). Consequence objects represent information about false positives/negatives (such as join points, aspects, or pointcuts) that will be given to the user. Figure 5 shows a simple schema of the design.

The repository manages the entity-relationships model that represents the program source code. The repository is implemented as a relational database. Each change operation class has a specific SQL query to execute. The parameters of the query are fields of the particular atomic change operation class. Figure 6 presents a very simple scenario in which we need to evaluate the consequences of removing the *setBalance* method of the *Account* class.

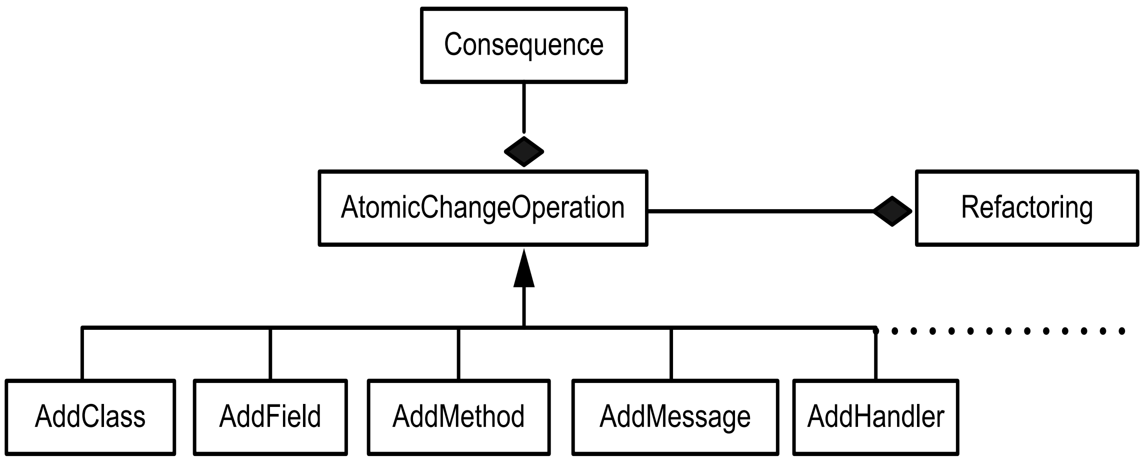


Figure 5. Diagram of classes of change operations

The *Logging* aspect is matching calls of all methods of the *Account* class. The Pointcut table contains all join points matched by each pointcut. An instance of the Remove Method class is created with *setBalance* and *Account* fields. Then a query is set up with these values. The executeQuery method executes the query and maps the results to Consequence objects.

4.2. Examples

Next we present several examples of anticipating the consequences of refactoring in a Spaceware AO application. Spaceware implements a clone of the famous "Asteroids" computer game. The source code is distributed by Eclipse. When an application is edited with Eclipse, the "Refactor" wizard automatically reports or recommends the refactorings that the developer can apply. In this case, Refactor suggests 15 refactorings. First, we have (manually) decomposed each refactoring into atomic change operations, and then we have analysed them with BaLaLu. An instance is showed in Table 3. Refactor recommended Pull up Method refactoring in which the *handleCollision* method of the *EnergyPacket* and *Bullet* classes should be moved to the *SpaceObject* superclass. This movement of code requires 3 atomic change operations, which are explicit in the Table. This set of atomic change operations (this refactoring) will produce 3 consequences: 1 potential false positive and 2 potential false negatives.

```
chop = new RemoveMethod("setBalance", "Account")
```

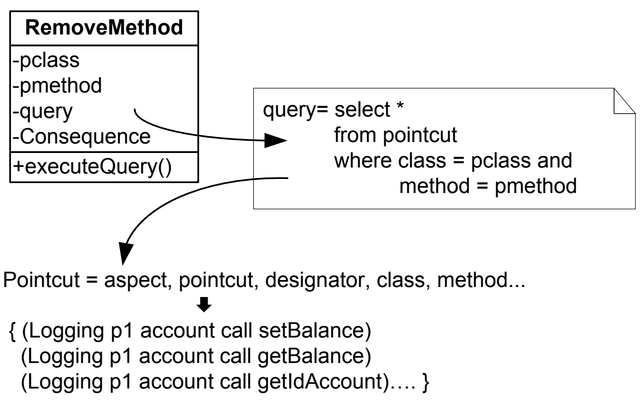

RemoveMethod
-pclass
-pmethod
-query
-Consequence
+executeQuery()
query= select *
from pointcut
where class = pclass and
method = pmethod
Pointcut = aspect, pointcut, designator, class, method...
{ (Logging p1 account call setBalance)
(Logging p1 account call getBalance)
(Logging p1 account call getIdAccount)…. }

Figure 6. Instance of atomic change operation

Table 3. Decomposing a Refactoring Recommended by Refactor

Refactoring	Set of Atomic Change Operation	PFP	PFN
Pull up Method[pull handleCollision method of EnergyPacket and Bullet subclasses up to SpaceObject superclass]	RM(handleCollision,EnergyPacket)	0	1
	RM(handleCollision,Bullet)	0	1
	AM(handleCollision,SpaceObject)	1	0
		1	**2**

BaLaLu reports the number of potential false positives/ negatives for each atomic change operation. It also reports the location of each of these consequences, i.e., the aspects, pointcuts, and join points that are affected.

The analysis of 15 refactorings proposed by Refactor indicates that all the refactorings will produce consequences. The graph in Figure 7 presents the numbers of potential false negatives (PFN) and false positives (PFP) reported by BaLaLu for each refactoring. Refactorings 7, 12, 13, and 14 are the most critical operations from this perspective.

Another issue that we should note is the number of atomic change operations in each refactoring. Figure 8 illustrates these. Here refactorings 7, 12, 13 and 14 are composed of sets of more than 20 atomic change operations, and in 2 cases (12 and 13), more than 60. There is an apparent relationship between the number of atomic change operations in a refactoring and the number of consequences that the refactoring can generate. We address this relationship in the next section.

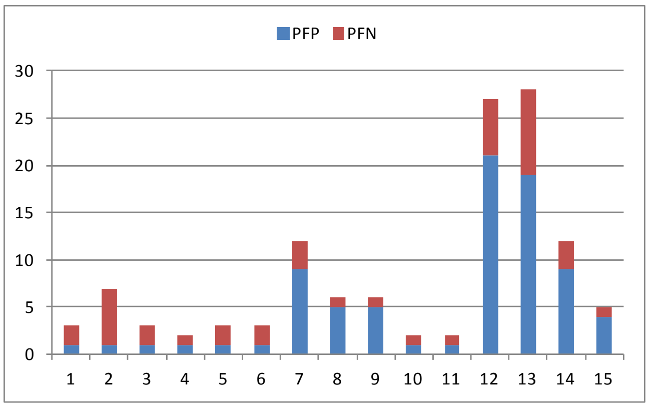


Figure 7. Consequences calculated by BaLaLu

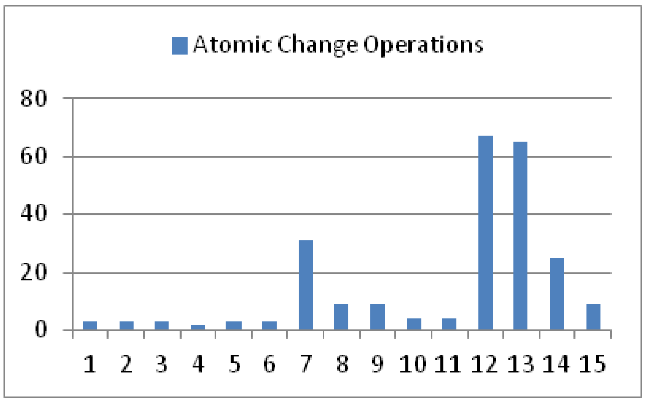


Figure 8. Atomic change operations in refactorings

5. Exploration of Descriptive Statistics

Based on our examples, we considered the possible relationships between the different variables involved when anticipating the impact of refactoring on AO applications. The questions we asked are: Does the number of atomic change operations composing a refactoring have any impact on the number of consequences? And does the number of join points intercepted by aspects affect the size of the impact a refactoring may cause? A priori, we believe that the answer to both questions is *yes*. To direct our discussion, we use descriptive statistics to test the hypotheses.

The AO applications used are Spaceware, Telecom (an Eclipse distribution code source), and Tetris (http://www.guzzzt.com/coding/aspecttetris.shtml). The main features of these systems are shown in Table 4.

Table 4. Features of Telecom, Tetris and Spaceware Systems

Feature	Telecom	Tetris	Spaceware
LOC	277	1043	1415
Classes	10	8	21
Methods	38	28	150
Fields	15	47	102
Aspects	3	8	9
Pointcuts	6	20	22
Advices	4	18	9

The refactorings recommended by Eclipse were used to formulate the list of refactorings which BaLaLu evaluated. Each refactoring was decomposed into atomic change operations, as discussed previously, and the set of atomic change operations was processed by BaLaLu. Refactor recommended 15 refactorings for Spaceware, 7 refactorings for Tetris, and 4 refactorings for Telecom. The number of atomic change operations composing the refactorings ranges from 2 to 67. The value used to measure the consequences of a refactoring is the number of potential false positives/ negatives.

5.1. Hypothesis 1

In this test, the variables that we analyse are the number of consequences of refactorings and the number of atomic change operations in the refactorings. The set of refactorings was divided into three groups. G1 is the group of refactorings that consist of at most 6 atomic change operations, G2 is the group of refactorings that consist of 7 to 19 atomic change operations, and G3 is the group of refactorings that consist of more than 19 atomic change operations. Table 5 shows the results.

Table 5. Descriptive Statistics for Consequences and Sets of Atomic Change Operations

	G1 2-6	G2 7-19	G3 20 or more
No. of Refactorings Evaluated	10	7	9
Consequences			
Min	0.00	0.00	0.00
Max	7.00	6.00	28.00
Mean	2.00	2.42	9.22
Median	2.00	0.00	4.00
Std. Dev.	2.01	3.05	11.47

The minimum number of consequences is 0 for any group, and the maximum is 28 for G3. The mean values increase from group G1 to group G3. The overall mean is 4.77. We have also evaluated the correlation factor between the number of atomic change operations in refactorings and the number of their consequences, which is 0.80. We can therefore say for the tested cases that there is a strong relationship between the number of atomic change operations in a refactoring and the number of potential consequences, which increases proportionally.

5.2. Hypothesis 2

To analyse whether there is a relation between the number of join points intercepted by pointcuts and the number of consequences generated by refactorings, we calculate the quotient between the total number of intercepted join-points for all aspects of the system and the number of pointcuts. The total number of join-points intercepted in Telecom is 6, in Tetris is 21, and in Spaceware is 129. The results are presented in Table 6.

Table 6. Join points/pointcuts for Spaceware, Tetris and Telecom

	Telecom	Tetris	Spaceware
Join-points/pointcuts	1	1.05	5.86

We also calculated the same values for the potential consequences of each system (Table 7). Using the numbers of join points/pointcuts (Table 6) and the number of consequences (Table 7), we evaluated the correlation factor, which is 0.99. As with the previous test, for these specific cases we can say that there is a strong relationship between the number of intercepted join points and the potential consequences.

Table 7. Descriptive Consequence Statistics for Telecom, Tetris and Spaceware

Consequences	Telecom	Tetris	Spaceware
Min	0	0	1
Max	0	4	28
Mean	**0**	**0.57**	**8**
Median	0	0	5
Std. Dev.	0	1.51	8.58

6. Related Work

A classification of AO refactoring methods based on their measurable effect on (i) internal software quality metrics, (ii) external software quality attributes, (iii) AOP constructs, and (iv) AO refactoring between and within aspects was presented in[10]. The classification was carried out by mapping the changes in the internal quality metrics, caused by applying refactoring methods, to the external quality attributes based on research studies that show correlations between the internal quality metrics and the external quality attributes. Six software systems were used to propose the classification; three other systems were also used for validation.

The idea of anticipating the impacts and consequences of change operations using a repository was inspired by the Change Bases Software Evolution (CBSE) approach[11][12][13]. CBSE, which arose to overcome typical system configuration and version problems, treats changes as first-class entities. One important difference between the CBSE model and our proposal lies in the intention of CBSE.

The CBSE model attempts to define the history of a program as the sequence of submitted program changes. Based on the history of changes, a developer can rebuild each successive state of a program's source code. The success of the CBSE model requires it to be implemented in these IDEs or development tools, while our model can be incorporated in the IDEs or in other specific tools such as BaLaLu. Finally, CBSE applies only to OO applications (Java and Squeak) and does not consider AO applications, although we assume that it is possible to extend the CBSE model to AOP.

Several tools such as PCDiff[14], Celadon[15], and Souyoul[16] and[17] analyse change impacts in AO programs. In general, these tools analyse and compare two or more versions of a source code program. The observed differences are used to derive a set of atomic change operations. These tools employ abstract representations of programs such as syntax trees, call graphs, and dependency graphs, and they also include test cases. Important differences between these tools and our proposal are as follows. a) These tools subscribe to methods based only on the comparison of program versions, and the analysis of change impacts is performed *after* the changes occur, while our goal is to propose a method that can identify the consequences of changes *before* they occur. b) Insofar as these tools and approaches work with program versions they can only identify the impacts of "atomic" change operations, while our approach also aims to analyse composite (move, rename) and complex (refactoring) operations. A key difference here is that the cited tools analyse atomic change operations in isolation, as independent and dissociated operations, and not as part of a more complex structure. These analyses and results inevitably lose the original semantics and integrity that existed before such changes.

ViDock[18] is a tool for analysing the impact of aspect weaving on test cases. The approach uses static analysis to identify the subset of test cases that are affected by the tissue counts. This tool works after making source code changes.

A method to analyse the change impacts of woven aspects is proposed in[19], but the method is not supported by a tool. This work analyses how aspects can change the control flow, input/output parameters, values of data members, and inheritance dependencies of the base code. It also describes the influences and possible effects of pointcut declarations on inheritance and overriding dependencies and how the ripple effects can be computed.

Several methods have been proposed to recover or discover "subsets of refactorings", but only in OO systems [20][21][22][23][24]. They employ different techniques such as UML diagram analysis, spatial vectors, heuristic metrics, data mining, analysis of CVS repositories, etc. These proposals analyse changes "after" they are produced and do not predict potential consequences.

7. Conclusions

This work has presented an approach to predicting the consequences of refactoring in AO software. The decomposition of refactoring into sets of atomic change operations is an efficient strategy that enables anticipating the impacts that they will produce. A disadvantage of our approach is the need to manually identify the atomic change operations of refactoring, but this is not a complex task. Additionally, we only work with the more usual refactorings, specifically, those that are recommended by Eclipse Refactor. We have not analysed the group known as Big Refactorings. For these cases, we plan to use a similar strategy, by first dividing the "big refactorings" into smaller refactorings. We will also investigate passing BaLaLu to IDE-based tools for Eclipse and connecting BaLaLu with Refactor.

In Section 5 we presented several tests and descriptive statistic studies that allowed us to remark that when a) a refactoring consists of many atomic change operations and/or b) the aspects have a large number of intercepted join-points, more consequences can follow. However, we do not consider these results to be conclusive, because more systems need to be tested. We are working on completing this study; meanwhile, the discussion of these questions remains open.

ACKNOWLEDGEMENTS

This work was partially supported by the Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.

REFERENCES

[1] Arnold R. S., 1986, An introduction to software restructuring, in Tutorial on Software Restructuring, Robert S. Arnold, Ed., Proceeding of IEEE, vol. 77, issue 4, 607–617.

[2] Fowler M., 1999, Refactoring: Improving the Design of Existing Programs, Addison-Wesley.

[3] Mens T. and Tourwé T., 2004, A survey of software refactoring, IEEE Transactions on Software Engineering, 30(2), 126–139.

[4] Kiczales G., Lamping G., Mendhekar J., Maeda A., Lopes C., Loingtier C., and Irwin J., 1997, Aspect-oriented programming, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finlands, LNCS 1241, Springer-Verlag, 220–241.

[5] Dijkstra E. W., 1976, A Discipline of Programming, USA, Prentice-Hall.

[6] Hürsch W. and Lopes C., 1995, Separation of concerns, Northeastern University Technical Report NU-CCS-95-03, Boston.

[7] Piveta E. and Zancanela L., 2003, Aspect weaving strategies, Journal of Universal Computer Science, 9(8), 970–983.

[8] Wloka J., 2003, Refactoring in the presence of aspects, 13th Workshop for Phd Students in Object Oriented Programming (ECOOP), Germany.

[9] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W., 2001, An overview of AspectJ, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Hungary, 327–353.

[10] Alshayeb, M., Al-Jamimi, H. and Elish, M. O.,2013, Empirical taxonomy of refactoring methods for aspect-oriented programming. J. Softw. Evol. and Proc., 25: 1–25. doi: 10.1002/smr.544.

[11] Robbes R. and Lanza M., 2007, An approach to software evolution based on semantic change, Proceedings of Fundamental Approaches to Software Engineering (FASE), LNCS Volume 4422, Portugal, 27–41.

[12] Robbes R. and Lanza M., 2006, Change-based software evolution, 2nd International ERCIM Workshop on Software Evolution, France, 159–164.

[13] Robbes R. and Lanza M., 2007, A change-based approach to software evolution, ENTCS, 166(1), 93–109.

[14] Koppen C. and Stoerzer M., 2004, PCDiff: Attacking the fragile pointcut problem, European Interactive Workshop on Aspects in Software, Germany.

[15] Zhang S. and Zhao J., 2007, Change impact analysis for aspect-oriented programs, Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, Shanghai Jiao Tong University.

[16] Bouteraa I. and Bounour N., 2011, Towards the use of program slicing in the change impact analysis of aspect oriented programs, Proceedings International Arab Conference on Information Technology, Saudi Arabia.

[17] Cavallero L. and Monga M., 2009, Unweaving the impact of aspect changes in AspectJ, Proceedings of the workshop on Foundations of aspect-oriented languages (FOAL), USA, 13–18.

[18] Delamare R., Muñoz F., Baudry B., and Le Traon Y., 2010, Vidock: A tool for impact analysis of aspect weaving on test cases, Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, Berlin: Springer-Verlag, 250–265.

[19] Liu C., Chen S., and Jhu W., 2011, Change impact analysis for object-oriented programs evolved to aspect-oriented programs, ACM Symposium on Applied Computing, Taiwan, 59–65.

[20] Di Penta G. and Merlo E., 2004, An automatic approach to identify class evolution discontinuities, 7th International Workshop on Principles of Software Evolution, Japan, 31–40.

[21] Serge Demeyer O. and Ducasse S., 2000, Finding refactorings via change metrics, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, USA, 166–177.

[22] Xing Z. and Stroulia E., 2003, Recognizing refactoring from change tree, 1st International Workshop on Refactoring: Achievements, Challenges, Effects, Canada, 41–44.

[23] Xing Z. and Stroulia E., 2004, Understanding class evolution in object-oriented software, 12th International Workshop on Program Comprehension, Italy, 34–43.

[24] Carsten P., 2005, Detecting and visualizing refactorings from software archives, Proceedings of the 13th International Workshop on Program Comprehension, USA, 205–214.