A Change Impact Analysis Model for Aspect Oriented Programs

Fabrice Déhoulé, Linda Badri and Mourad Badri
Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Canada

Keywords: Software Evolution, Aspect-Oriented Programming, Change, Ripple Effect, Impact Analysis, Predictive Analysis, Model, Impact Rules, Empirical Analysis.

Abstract: Software change impact analysis (IA) plays a crucial role in software evolution. IA aims at identifying the possible effects of a source code modification. It is often used to evaluate the effects of a change after its implementation. However, more proactive approaches use IA to predict the potential effects of a change before it is implemented. In this way, IA provides useful information that can be used, among others, to guide the implementation of the change and to support regression tests selection. This paper aims at proposing a change impact analysis model for AspectJ programs. Aspect-Oriented Programming (AOP) is a natural extension of Object-Oriented Programming (OOP). It particularly promotes improved separation of crosscutting concerns into single units called aspects. The IA techniques proposed for object-oriented programs are not directly applicable for aspect-oriented programs due to the new dependencies introduced by aspects. The proposed model was designed to particularly support predictive IA. The model includes several impact rules based on the AspectJ language constructs. We performed an empirical evaluation of the model using several AspectJ programs. In order to assess the model prediction quality, we used two traditional measures: precision and recall. The reported results show that the model is able to achieve high accuracy.

1 INTRODUCTION

As software systems are used for a long period of time, software evolution is inevitable. Indeed, software systems need to continually evolve for various reasons, including: adding new features to satisfy user requirements, changing business needs, introducing novel technologies, correcting faults, improving quality, and so forth. So, as software evolves, the changes made to the software must be carefully managed. It is particularly important to ensure that modified software still verifies its specification and whether new errors were introduced inadvertently (Kung et *al*., 1995; Rothermel and Harrold, 1997; Harrold et *al*., 2001; Hunt et *al.*, 2008). It is, therefore, crucial to find where changes occur and to identify parts of the software that are possibly affected by the changes, parts that must be correctly retested. Indeed, for obvious reasons, retesting all the software after instantiating a change is inefficient, costly and unacceptable in practice (Rothermel and Harrold, 1996). In the software life cycle, maintenance plays a fairly important role (Lehman et *al.*, 1997). It is during this step that we can change a program to improve it, adapt it to new specifications, or prevent any errors (Law and Rothermel, 2003; Lehnert, 2011). Its importance is even more increased by the fact that the systems produced nowadays are becoming more complex and voluminous (Lehman et *al.*, 1997).

Software evolution faces many challenges (Lehman, 1980; Lehman et *al.*, 1997; Ebert and De Man, 2005; Mens et *al.*, 2008). Software maintenance is, in this context, a vital activity (Bennett and Rajlich, 2000). It is, however, costly (Grubb and Takang, 2003; Abran et *al.*, 2004). Several experts agree that two of the most important activities of software maintenance are: understanding the software and evaluating the potential effects of a change (Barros et *al*., 1995; Aggarwal et *al.*, 2002; Riaz et *al.*, 2009; Baggen et *al.*, 2011; Cho et *al.*, 2011). The second activity is closely related to the first one. Indeed, to understand the effects of a given change, it is necessary to understand the system beforehand (Lee et *al.*, 2000). The software design, particularly the dependencies between its components, can make this task difficult.

Déhoulé, F., Badri, L. and Badri, M.
A Change Impact Analysis Model for Aspect Oriented Programs.
DOI: 10.5220/0006350701440157
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 144-157
ISBN: 978-989-758-250-9

A change to a system, even minor, can lead to several unintended effects (ripple-effect). One effective way to deal with this important issue is to develop models (and techniques) that can be used to support the evaluation of the potential effects of a change. This can be used to guide the decision-making of software development managers seeking to produce high quality software.

Software change impact analysis (IA) plays a crucial role in software evolution. Bohner and Arnold (1996) defined change impact analysis as "the process of identifying the potential consequences of a change, or estimate what need to be modified to accomplish a change". IA allows, indeed, developers assessing the possible effects of a given source code modification (Yau and Collofello, 1980; Li and Henry, 1995; Li and Offutt, 1996; Bohner and Arnold, 1996; Briand et *al*., 1999; Lee et *al.*, 2001; Chaumun et *al.*, 2002; Law and Rothermel, 2003; Ren et *al.*, 2004; Ackermann and Lindvall, 2006; Li et *al.*, 2012). IA can be used to support various maintenance tasks such as: planning changes, assessing the cost of changes, implementing changes, tracking the effects of changes and regression tests selection (Law and Rothermel, 2003; Orso et *al.*, 2003; Orso et *al.*, 2004; Ackermann and Lindvall, 2006).

Aspect-Oriented Software Development (AOSD) is a promising new software engineering paradigm (Sabbah, 2004; Dong, 2011). AspectJ, as an aspect-oriented programming language, represents an interesting extension of Java (Przybylek, 2011). In fact, existing object-oriented programming languages suffer from a serious limitation in modularizing adequately crosscutting concerns (Przybylek, 2011). Many concerns crosscut several classes in an object-oriented program. Crosscutting is a structure that goes beyond hierarchy as stated in (Bernardi and Di Lucca, 2007; Bernardi et *al.*, 2009; Przybylek, 2011). The code related to a crosscutting concern is generally duplicated within several classes in an object-oriented program. Consequently, these classes would be difficult to understand, maintain and reuse. Aspect-Oriented Programming (AOP) deals with scattered and tangled code related to crosscutting concerns. It particularly promotes improved separation of crosscutting concerns into single units called aspects (Zhao, 2004; Baggen et *al.*, 2011; Przybylek, 2011).

Although AOP was introduced to separate concerns and improve software modularity, modifying aspect-oriented programs will lead to more complex impacts than in object-oriented programs (Zhang et *al.*, 2008; Burrows et *al.*, 2010). Storzer (2007) indicates that the aspects and the base code are decoupled at syntax and that we need to know the relationships between classes and aspects of the program. He also indicates that aspects can interfere with each other, interference that may be difficult to resolve. It is important to mention that the evolution of the base object code may change or break the aspects' semantic, since they are based on the object code.

With an aspect-oriented program, there will be four impact possibilities: (i) impacts of changes introduced into the object-code part on itself (Object - Object), (ii) impacts of changes made in the object code part on the aspect part (Object - Aspect), (iii) impacts of changes in the aspect code (Aspect - Aspect), and (iv) impacts of changes made into the aspect part on the object part (Aspect - Object). Our work focuses on the last three types of impacts. The first type of impacts concerns only object-oriented programs and has been addressed in our previous work (Badri et *al.*, 2015).

We present, in this paper, a new static change impact analysis model for AspectJ programs. The model, including several impact rules based on the AspectJ language constructs, was designed to support predictive impact analysis. We performed an empirical evaluation of the model using several AspectJ programs. We considered in the study different types of changes. In order to assess the model prediction quality, we used two traditional measures: precision and recall. In addition, we evaluated the proposed approach using the properties of the Framework proposed by Li et *al.* (2012) characterizing impact analysis techniques. The proposed model complements, in fact, the Change Impact Model for Java programs (CIMJ) that we developed in our previous work (Badri et *al.*, 2015).

The rest of the paper is organized as follows: Section 2 presents an overview of the main related work. Section 3 provides a brief overview of aspect-oriented programming (AOP). Section 4 introduces the impact analysis model we propose. Section 5 presents the empirical study we conducted in order to assess the performance on the proposed model. Section 6 gives a conclusion and some future work directions.

2 RELATED WORK

Change impact analysis aims to predicting which parts of the code will be affected following a modification (Ackermann and Lindvall, 2006). IA

allows identifying the consequences of a change, or estimating the parts to modify in a program to ensure that the change is made correctly (Ali et *al.*, 2012). IA is very important because it allows, among others, to help change management, and keeping the system stable (Arnold and Bohner, 1993; Ali et *al.*, 2012).

Many criteria have been proposed in the literature for classifying existing IA techniques (Kilpinen, 2008; Lehnert, 2011; Li et *al.*, 2012; Sun et *al.*, 2014). These techniques addressed, in fact, various specific tasks of software maintenance. Existing IA techniques can be static and/or dynamic (Lee et *al.*, 2000; Law and Rothermel, 2003; St-Yves, 2007; Petrenko and Rajlich, 2009; Zhou et *al.*, 2011; Acharya and Robinson, 2012; Li et *al.*, 2012), based on the source code of the program and/or on models (St-Yves, 2007; Petrenko and Rajlich, 2009). Static IA techniques include structural static analysis, textual analysis, and historical data analysis (Zimmermann et *al.*, 2005; Petrenko and Rajlich, 2009; Gethers and Poshyvanyk, 2010; Sun et *al.*, 2014). Static analysis techniques are based on the syntax and semantic dependencies of the program. These techniques use most of the time system representations such as call graphs, control flow graphs, etc. Dynamic analysis techniques are based on information gathered during the execution of the program (Sun et *al.*, 2010).

Impact analysis techniques can be divided in two major classes: impact analysis techniques that support predictive analysis - pre-change (Chaumun et *al.*, 2000; Badri et *al.*, 2005; St-Yves, 2007; Abdi et *al.*, 2007; Badri et *al.*, 2015) and impact analysis techniques that support retrospective analysis - post-change (Kabaili et *al.*, 2001; Ren et *al.*, 2004). Predictive impact analysis techniques are used before the change is implemented, and aim mainly at predicting the potential effects of a change, which allows assessing the effort required for its implementation. Retrospective impact analysis techniques are used after a change has been implemented. These techniques aim mainly at supporting the correction of potential errors that are introduced by changes, and regression testing (Kabaili et *al.*, 2001; Orso et *al.*, 2003; Orso et *al.*, 2004; Ren et *al.*, 2004; Li et *al.*, 2012).

Different approaches have been proposed to predict the impact of changes made on aspect-oriented programs. Bernardi and Di Lucca (2007) developed an Inter-procedural Aspect Control Flow Graph. This graph shows the relationships between class's methods and advices in aspect code, and also indicates where the advice code will be inserted in a method during weaving. One of the benefits of the graph, according to the authors, is to save time during maintenance steps, about 20% less time. The graph do not take into account neither exception management, nor inter-type declarations and static initializers, and the result may be a large number of nodes, making the analysis a little more complex. The authors have addressed the static analysis of the code, even if the graph is also able to manage the dynamic analysis.

Shinomi and Tamai (2005) developed an algorithm that aims to list the impacts. This algorithm successively generates a syntactic abstract tree, a control flow graph, a call graph of methods or advices and finally a dependency graph for each method. Once all these graphs are obtained, the algorithm establishes a starting point of impact list for each aspect. Then, from each starting point of impact corresponding to an aspect (weaving point), the parts affected by aspects are marked. This impact analysis is done after the weaving and concerns only impacts on object code after changes made in aspect code.

The impact analysis technique proposed by Zhang et *al.* (2008) identifies the parts of the source program and the tests affected by a change. This technique relies on atomic changes that capture the semantic differences between two versions of a program. These atomic changes are at the level of the source program, but also in the aspect code. Atomic change transactions are meant for an addition / deletion / modification of method, aspect, pointcut, class or attribute. Thus, to determine which parts of the program may be affected by a change (atomic change), the authors list the changes. Then, a program call graph is generated. The code fragments affected by a change will be the nodes of the graph that are linked to the modified node. This is the transitive closure of each modified node. This impact analysis technique has the disadvantage of not being predictive since it is necessary that the modifications were made before determining which pieces of the code are affected.
Zhao (2002) operates on a slicing aspect oriented system dependency graph (Aspect-Oriented System Dependence Graph) made of several modules dependencies graphs (Module Dependence Graph) interconnected.

3 ASPECTJ: BASIC CONCEPTS

Designed by G Kiczales (Kiczales et *al.*, 2001), AspectJ is an aspect-oriented extension to the Java

language (Gradecki and Lesiecki, 2003). With AspectJ, the aspect code will be incorporated into the basic program through weaving when compiling the code. AspectJ introduces several new language constructs (aspect members) such as: aspect, join points, pointcuts, advice as well as inter-type declarations (Kiczales et *al.*, 2001; Dong, 2011). These various elements allow an aspect expressing a concern that crosscut several basic classes. Aspects are built like object classes, in which all elements related to an aspect are defined. A join point represents well-defined points in the program flow where the aspects will be weaved; such as method calls, exceptions, interfaces or other instructions of the basic object classes. Pointcuts describe join points and context to expose. An advice is a method-like abstraction that defines code to be executed when a join point is reached. Pointcuts are used in the definition of an advice. Inter-type declarations define how AspectJ modifies a program's static structure, namely, the members and the relationships between components. Inter-type declarations alter the structure of the object program by adding methods or attributes to an object class, changing the inheritance of a class, or by specifying that a class implements one or more interfaces. Pointcuts and advices dynamically affect program flow, and inter-type declarations statically affect a program's class hierarchy. For further information on AspectJ mechanisms, one can see (Kiczales et *al.*, 2001; Przybylek, 2011).

4 IMPACT MODEL

4.1 Objectives

We present, in this section, the Impact Analysis Model that we defined for Changes in AspectJ programs. Our model aims to predicting the different parts of an aspect-oriented application that will be affected due to a change in the program. An aspect-oriented program contains object code and aspect code, so there will be changes that may affect both parts. The model specifies for each type of change a set of impacts which provide useful information allowing taking into account cascading impact. The model includes various atomic changes. An atomic change is the smallest unit of change that cannot be decomposed into other changes. Atomic changes are divided in two distinct groups: structural and non-structural changes. Our Aspect Oriented Change Impact Model (AOCIM) complements the CIMJ (Badri et *al.*, 2015), developed in our previous work for Java programs, and will be used in a predictive analysis context.

Change Impact Model for Java (CIMJ) was developed by N. Joly et *al.* (Joly, 2010; Badri, et *al.*, 2015). The impact model considers the impact of object code to object code. As mentioned, the CIMJ allows a predictive impact analysis and a post-analysis. However, the emphasis was on predictive analysis. It is an impact model that allows a cascade impact analysis, responding to the problem of the ripple effect of change impacts. The MICJ is limited only to the object-oriented programs. Indeed, it only considers the impacts that will occur in the object-oriented code following the changes made on the same object-oriented code. To extend the CIMJ to the aspect part of the program, we have introduced three impacts categories:

Impacts - object to aspect: Here, we will see the impact of any changes in the classes, methods, attributes on the aspect code. These changes will also involve exceptions and inheritance relationships.

Impacts - aspect to aspect: This category of impacts will be located only in the aspect code. We deal here with the consequences that will occur on the remaining aspect code following the changes performed at portions such as a pointcut or intertypes declarations.

Impacts - aspect to object: When changing the aspect part of a program, this could affect the object code. Therefore, it is about to determine which parts of the object code will be impacted.

4.2 Relationships in an Aspect-oriented Program

Three types of relationships in an aspect-oriented program were considered in the impact model.

Association: a module referring to another (Kumar et *al.*, 2007). An association relationship is created between an aspect and one or more classes through joinpoints or by inter-type statements. The association is also created at the level of the advices when an object of a class is instantiated in an advice. Finally, there will be an association between an aspect and a class when a method at the aspect level will take as parameter object type of the class.

Inheritance: an inheritance link created between two modules leads child to benefits from the properties of the parent module (Kumar et *al.*, 2007). An aspect S can inherit from a class A but also from another aspect P.

Local pseudo-relationships: some impacts may occur inside the aspect in which the change takes place (Kumar et *al.*, 2007). This happens for example when the parameters of a pointcut are changed. Therefore, it will also change the parameters of advices related to this pointcut.

4.3 Structural and Non-structural Changes

A structural change is a change that alters the structure of a class or an aspect. These changes can be for example a removal of a pointcut, an addition or deletion of a class, a method, an attribute. We have counted a total of 61 structural changes, 22 at the object code level and 39 at the aspect code level (see annex 1). As shown in Figure 1, removing the pointcut setter (in bold) would be a structural change since it changes the structure of the appearance of the aspect.

```
pointcut setter(Point p): call(void Point.set*(*)) &&
target(p);

void around(Point p): setter(p) {
  String
propertyName=thisJoinPointStaticPart.getSignature().ge
tName().substring("set".length());
  int oldX = p.getX();
  int oldY = p.getY();
  proceed(p);
  if (propertyName.equals("X"))
  {
      firePropertyChange(p, propertyName, oldX,
p.getX());
        } else {
      firePropertyChange(p, propertyName, oldY,
p.getY());
  }
}
```

Figure 1: Structural change.

```
after () returning (Player player): call
(Player+.new(..)) {
    Enumeration elements = DISPLAYS.elements();
    while (elements.hasMoreElements() ) {
       Display display =
(Display)elements.nextElement();
          display.addKeyListener(player);
       }
    }
}
```

Figure 2: Non-structural change.

Non-structural changes will not alter the program structure. They are within a method or a cup or an advice. One example of no structural change is removing a method call. In the example given in Figure 2, there will be no structural impact at the after-advice in which the call of *addkeyListener* method is deleted (in bold in the example).

4.4 Concept of Certainty

The AOCIM model uses the notion of certainty, a concept which also exists in the CIMJ model. The notion of certainty allows basically mitigating the information provided by the impact analysis using the AOCIM model. To illustrate this concept, let us consider two simple examples. As a first example, let us consider the atomic change “deleting a pointcut". This removal will impact all uses of this pointcut. To compile the code after removing the pointcut, we must also remove all its uses (advices attached to it). In this case, we are talking about an impact that is certain (certainty of the impact). Let us take as a second example the atomic change "add of joinpoint to a pointcut". Normally, if we add a joinpoint to a pointcut, it is that we intend to use it. Otherwise, it would be an unnecessary change, but which nevertheless remains possible. So, we can expect an impact related to the addition of the use of this joinpoint. In this case, we are talking about an impact that is uncertain (uncertainty of the impact). The AOCIM model is based on several impact rules, which make the distinction between the impacts that are certain and the impacts that are uncertain. So, the model makes a difference between what will be impacted and what could possibly be impacted.

4.5 Impact Rules

A total of 41 impact rules have been defined. We classified these impact rules based on the three impact categories we mentioned above, which are the impacts of the object part on the aspect part (17 rules), impacts occurring in the aspect code following a change in the aspect code (23 rules), and the impact of the aspect code on the object code (1 rule) (see annex 2). For the first two impact categories, the targeted elements are located only at the aspects level. The last impact category, aspect on object impact, will target only elements in classes. The rules presented in our model take into account all possible impacts that are related to AOP. To a better understanding of how the model we developed works, we will take few examples.

4.6 Cases of Objet Code Impacts on Aspects

When changes are made on the object code of an aspect-oriented program, the impacts could occur in the aspect code. Our model presents 17 rules to identify these impacts.

4.6.1 Removing a Method

As a first example, we will take the removal of a method in a class (void *calculate*()). The relationship between aspects and classes will be done through joinpoints. Following the deletion of a method in a class, the joinpoint associated with this method is automatically affected.

```
public class Circle extends Figure
{
   private double radius;

   public void calculate()
   {
          circumference =2*3.14*radius;
    }
}
```

```
public aspect AspectFigure
{
    public pointcut pcCalculateCircle():
      execution (void Circle.calculate());

    before():pcCalculateCircle()
    {
      System.out.println("Circumference calculation");
    }
}
```

Figure 3: Class to aspect relationship.

Figure 3 shows how a method is referenced by a joinpoint in an aspect. This joinpoint belongs to a pointcut called " *pcCalculateCircle* " (bold text). And finally, an advice is linked to this pointcut in order to execute a code. Thus, when the method in question will be removed, first, there will be an impact on the joinpoint, then on the pointcut containing that joinpoint, and finally on the advice related to that joinpoint. So, we have the following impact rule:

Mr -> JPr + PCm + ADVm

"M" refers to a method, "JP" for a joinpoint, "PC" for pointcut and finally "ADV" for advice. The letter "r" means that the item is removed and the letter "m" means that the item is modified. The rule is read as follows: The suppression of a method (Mr) causes (->) the suppression of the referring joinpoint (JPr), followed by the modification of the pointcut containing the joinpoint (PCm) and finally the modification of the advice associated to the pointcut (ADVm).

4.6.2 Distinctive Feature of the Removal of a Class

In an aspect, multiple joinpoint and several intertype statements can be defined. These joinpoints and intertype declarations can be linked to different classes as shown in Figure 4.

In this example, the aspect "Logging Aspect" is related to the "Client" and "Article" classes. By removing the Client class, only the advice and the method associated with the class Client (bold an italic text in the figure) will be affected, not forgetting the import of Client class (bold and underlined text in Figure 4).

To increase the accuracy of the impact rule concerning the suppression of a class, we added a variable indicating the class concerned by the deletion. This will result to focus only on the elements of the aspect having a link with the deleted class. Thus, we have the following impact rule:

```
import ca.uqtr.gl.entities.Article;
import ca.uqtr.gl.entities.Client;

public aspect LoggingAspect {
      private Logger logger =
Logger.getLogger("trace");

      pointcut addClientMethod():
      call(void addClient(..));

      pointcut addArticleMethod():
      call(void addArticle(..));

      after() returning() : addClientMethod() {
            Object[] paramValues =
thisJoinPoint.getArgs();
            String lastName = (String) paramValues[0];
            String firstName = (String) paramValues[1];

         Client client = new Client(lastName,
    firstName);

            logger.logp(Level.INFO, null, null, "\n "
            + "New CLIENT\n" + getLogClient(client)
      );
      }

      private String getLogClient(Client c) {

         String log = "Full name: " + c.getFirstName()
         + " " + c.getLastName();
         return log;
      }

      after() returning() : addArticleMethod() {
            Object[] paramValues =
thisJoinPoint.getArgs();

            Article a = new Article();

            a.setCode((String) paramValues[0]);
            a.setDescription((String) paramValues[1]);

         logger.logp(Level.INFO, null, null, "\n "+
         "New ARTICLE\n"+ getLogArticle(a));
      }

      private String getLogArticle(Article a) {
```

Figure 4: Aspect related to several classes.

Impact rule: Cr(Class) -> JPr + [PCm || PCr] + [ADVr|| ADVm] + I-TYr + DCLm + Mm + IMPr

Removing of class in brackets (Cr (Class)) implies the removal of the joinpoints (JPr) potentially followed by (represented by the brackets) the suppression or the modification of the pointcut

([PCm || PCr]). Then, the possible removal or modification of an advice ([ADVr ||ADVm]), the removal of inter-type declarations (I-TYr), and the modification of the declarations (DCLm). Finally, we have the modification of the methods using the class (Mm) and the suppression of import of the class in question in the aspect (IMPr).

Applying this impact rule to our example, the advice and the method using the *Client* class are well indicated by the impact rule and the import will be also deleted.

4.7 Cases of Aspect Impacts to Aspect Code

Aspects include various elements such as advices and pointcuts, but also attributes and methods. In addition, several aspects are interlinked through inheritance. Our model allows knowing which parts of the aspect will be impacted after an element of the aspect is changed. Our model presents 23 rules for this category of impacts.

4.7.1 Removing a Pointcut

Deleting a pointcut will give the following impact rule:

PCr -> PJr + ADVr + PCm(L) + PCr(H) + PJr(H) + ADVr(H)

Removing a pointcut (PCr) automatically leads to the suppression of joinpoints (PJR) and the removal of the advices attached to them (ADr). Also, pointcuts referencing the deleted pointcut will be modified (PCm (L)). All pointcuts redefining the deleted pointcut in a child aspect will be also deleted (PCr (H)) as well as joinpoints and advices of this child aspect (PJr(H) + ADVr(H)).

In this impact rule, the "L" indicates that the change is made at the aspect, that is to say locally. Meanwhile, the "H" indicates that impacts occur in aspect inheriting from the aspect where the modification has been made.

As shown in Figure 5, pointcut "**xSet**" (in bold in Figure 5) is related to an "**after**" advice and is also referenced by another pointcut, the pointcut "**Set**" (underlined and bold in Fugure 5). When we remove pointcut xSet, the advice related to it is no longer useful, so we can remove it. Also, in the pointcut Set, an impact will occur because we have to remove the reference to that pointcut xSet.

Before removal of pointcut

```
public aspect Intro
{
    pointcut xSet():
        set(int Point.x);

    after(): xSet()
    {
        System.out.println("Attribute X modified");
    }

    pointcut ySet():
      set(int Point.y);

    pointcut Set():
      xSet()||
      ySet();

    after(): Set()
    {
        System.out.println("X and Y modified");
    }
}
```

After removal of pointcut

```
public aspect Intro
{
    pointcut xSet():
        set(int Point.x);
    after(): xSet()
    {
        System.out.println("Attribute X modified");
    }

    pointcut ySet():
      set(int Point.y);

    pointcut Set():
      xSet()||
      ySet();

    after(): Set()
    {
          System.out.println("X and Y modified ");
    }
}
```

Figure 5: Example of impact rule: removal of pointcut.

4.7.2 Deleting a Method in an Aspect

Modification or deletion of a method in the aspect code will impact the elements using this method in this aspect, but also the child aspects. So we have the impact rule:

Mr -> ADVm + Mm(L) + Mm(H)

The removal of a method (Mr) will cause the modification of advices using this method (ADVm). There will also be an impact on the methods using the method deleted not only locally (Mm (L)) but also at the level of child aspects of that modified aspect (Mm (H)).

Figure 6 shows an example of removing a method in an aspect. After removing the *loggedAllCatalogues* method (bold text in the figure), the advices using the method (italic and underlined text in the figure) will be impacted.

```
pointcut loggedDelete(Object elem) :
call(*Catalogue*.delete*(..)) &&
args(elem)&&!withincode(* *Test.*(..));

    after(Object elem) : loggedDelete(elem)
    {
      String msg = "Element deleted";
      loggedAllCatalogue(msg, elem,thisJoinPoint);
    }

    private void loggedAllCatalogue(String
msgLog,Object elem,JoinPoint thisjoinpoint)
     {
      if(elem instanceof Client)
      {
            elem = (Client)elem;
      }
      else if(elem instanceof Article)
      {
            elem = (Article)elem;
      }

        msgLog = elem.getClass().getName().toString())
        + " " + elem.toString();

      logger.info("\n\t"+msgLog);
     }
```

Figure 6: Example of an aspect impact on object code.

4.8 Cases of Aspect Impacts on Object Code

In the relationship between object classes and aspects, aspects depend on object classes. In addition, adding an aspect does not affect the object code in a static point of view (before weaving). Thus, any modification, addition or deletion of joinpoint, pointcut or advice in an aspect will have no impact on the object code which that aspect is attached to.

However, there will be an impact of the aspect code on the object code at the intertype declaration level. Indeed, any declaration inserted into the aspect code can be used in the object code. Thus, any modification or deletion of these intertype statements will affect the object code. Therefore, one only rule was developed for this type of impact:

I-TYr | I-TYm-> Mm

Any deletion or modification of an intertype declaration will result to impacts on the object code methods (Mm) where intertype declaration is invoked.

Figure 7 shows that in the aspect "Aspect_Limousine", an attribute is introduced in the "Drivers" class with an intertype declaration (bold text). This attribute appears in the two methods of the "Drivers" class. If we delete this attribute in the aspect, the reference to it in both methods will no longer exist, causing an impact in those methods. We will then have to modify the code of the methods to keep the code errorless.

```
public aspect Aspect_Limousine
{
      int Drivers.iNbMaxRun = 100;

      public pointcut pcNewRun():
                initialization(Run.new(String, int,
                Limousine));

      after(): pcNewRun()   {
                System.out.println("New run created");
      }
-
```

```
public class Drivers {
      private Run aRun[];
      private int iNbRun, iYearOfHiring;
      private String sLastName;
      private String sFirstName;
      private String sAddress;
      private boolean bAvailable;

    public Drivers(int yearOfHiring, String lastName,
    String firstName, String address)
    {
            iNbRun = -1;
            iYearOfHiring = yearOfHiring;
            sLastName = lastName;
            sFirstName = firstName;
            sAddress = address;
            bAvailable = true;
            aRun = new Run[iNbMaxRun];
      }
      public void addRun(Run r)
     {
      if((bAvailable == true) && (iNbRun < iNbMaxRun -
      1))
      {
            bAvailable = false;
            iNbRun ++;
            aRun [iNbRun] = r;
        System.out.println("Run created");
      }
      else
      {
        System.out.println("Impossible to add this
      run");
```

Figure 7: Example of impact from aspect code to object code.

4.9 Cascading Analysis

Our model is able to perform a cascading analysis. Thus, when a change is made, we can see the consequence of this modification on the rest of the program. Cascading analysis means change impact analysis of impacts (Badri et *al.*, 2015). This is the ripple effect of a change made on the rest of the program. Using the cascading analysis, programmers will have a good idea on the ripple effect following a change. We will take an example to illustrate this type of analysis.

In Figure 8, the "BoundPoint" aspect contains an intertype declaration that introduces in "Point" class an attribute named "support" of type "PropertyChangeSupport" (bold text in Figure 8). Plus, there is a pointcut named "setter" (italic underlined text in Figure 8) related to that "Point" class. If the "Point" class is deleted, there will be direct impacts on the "BoundPoint" aspect as the pointcut "setter" and the intertype declaration

introducing the "support" attribute should be removed from the code.

```
aspect BoundPoint
{

    private PropertyChangeSupport Point.support = new
    PropertyChangeSupport(this);

    public void
    Point.addPropertyChangeListener(PropertyChangeList
    ener listener)
    {
      support.addPropertyChangeListener(listener);
    }


    pointcut setter(Point p): call(void Point.set*(*))
    && target(p);

    void around(Point p): setter(p) {
        String propertyName =
        thisJoinPointStaticPart.getSignature().getName
        ().substring("set".length());
        int oldX = p.getX();
        int oldY = p.getY();
        proceed(p);
        if (propertyName.equals("X"))
       {
          firePropertyChange (p, propertyName, oldX,
          p.getX());
       }
       else
       {
           firePropertyChange (p, propertyName, oldY,
           p.getY());
       }
  }

    void firePropertyChange(Point p,String
    property,double oldval, double newval)
    {
        p.support.firePropertyChange(property, new
        Double(oldval), new Double(newval)) ;
    }
}
```

Figure 8: Example of cascading analysis.

However, the pointcut "setter" is linked with an advice around (italic text in Figure 8) and the attribute "support" is used in the "firePropertyChange" method (bold, italic and underlined text in Figure 8). Therefore, given that the attribute "support" and the pointcut "setter" are going to be removed due to the impact caused by the deletion of class "Point", the "firePropertyChange" method and the around advice are going to be also impacted.

5 EMPIRICAL EVALUATION

In order to evaluate the ability of the AOCIM model to accurately predict the impact of changes, we conducted an empirical study. We have developed different programs using AspectJ and used projects developed by students in the Department of mathematics and computer science (University of Quebec at Trois-Rivières) (see Table 1).

Table 1: List of tested programs.

Projects	# classes	# aspects
Bean	3	1
Introduction	1	3
Observer	6	2
Spacewar	17	10
Telecom	10	3
TJP	1	1
Tracing (version 1)	5	3
Bank	2	1
Living beings	6	2
Figure	4	2
Matrix handling	3	1
Limousine	5	1
QuickSort	2	1
Stack	2	1
Soft. Eng. project 1	49	3
Store management	58	2
Soft. Eng. project 2	45	3
TOTAL	219	40

For some programs, we added a second aspect that inherits from another. Also, we used the programs contained in the file of the supplied examples provided with the AspectJ compiler to achieve our tests. In order to measure objectively the performance of the model, we used in this study two traditional measures: precision and recall. We considered the following types of changes: removal of class or aspect - rename of classes, aspects, methods and attributes - addition, modification and removal of methods, attributes - modification of pointcuts and removal of inheritance link. For each change, we introduced in a program, we observed the consequences.

5.1 Metrics

We used two well-known measures to evaluate the quality of the prediction of the impact analysis model: precision and recall. Impact analysis may have some false positives (elements in the impact set which aren't really impacted) and false negatives (elements really impacted which aren't identified in the impact analysis) (Li et *al.*, 2012). During the evaluation, we obtained three types of results: the number of actual impacts due to a given change, the number of impacts predicted by the model and the number of impacts correctly predicted by the model. The recall, which is an inverse measure of false negatives, is the ability of the model to predict all real impacts (% of actual impacts). This allows assessing whether the rules of the impact model

predict all real impacts. The precision, which is an inverse measure of false positives, is the ability of the model to predict correctly the impacts (% of predicted impacts corresponding to reality). This indicates whether the model is accurate enough to predict only the real impacts and also validate the recall. If it is too high than the precision, this means that the model predicts too many impacts. A perfect model would be a model having a recall of 100% and a precision of 100 %. The model succeeds in this case only to predict the actual impacts and to predict them all.

5.2 Evaluation Procedure

To conduct our experiments, we have made changes to the used aspect-oriented programs. These changes were made both in the object code and in the aspect code. For example, we deleted classes, renamed methods, changed the scope of an attribute or modified pointcuts. These changes are made without considering the impact given by the rules of the proposed model. Once these changes were made, we identified the different impacts that occurred. To do so, we went through the program and searched every part of the code related to the change we made. Some impacts were highlighted by the IDE we used (Eclipse), and others were about relationships (i.e. inheritance, association). This step allowed us to determine the number of real impacts on each modified program. Then, we toke each of the changes and identify the impacts that our model has predicted. Finally, a comparison of the results was carried out after the previous phases. We were then able to determine: (1) the ratio of impacts predicted by the model which actually occurred compared to observed impacts (precision), and (2) the percentage of real impacts predicted by the model compared to the set of predicted impacts by the model in relation with the change made (recall).

5.3 Results and Discussion

Table 2 presents a summary of the results obtained during the experiments. As it can be seen, we have an accuracy of 98% and a recall of 86.6%. Our impact model is able to detect almost all the impacts that will occur following a change in the aspect-oriented program. For Object impacts to Aspect, we have an accuracy of 100%. However, for Aspect impacts to Aspect, we have an accuracy of 95%. Impacts from aspects to object code present a recall and an accuracy of 100%. This is explained by the fact that there is only one impact rule for this impact category, and only methods will be impacted after changing an intertype declaration. Recalls of 93% and 77.8% respectively for the impacts from object to aspect and aspect to aspect are due to the fact that, in certain situations, the model can identify more impacts than those observed after a change because of the uncertain impacts. Indeed, it may happen that the impacts are not reported (following an effective change) because they do not affect the program run. It will be left up to the programmer's discretion to consider these impacts in order to keep consistency in the maintenance of the current program.

Table 2: Summary of the results.

	Impacts predicted by the model	Observed Impacts	Real impacts predicted by the model	Precision	Recall
Object impacts to aspect	40	37	37	100%	93%
Aspect impacts to aspect	27	22	21	95%	77,8%
Aspect impacts to object	3	3	3	100%	100,0%
TOTAL	**67**	**59**	**58**	**98%**	**86,6%**

5.4 Limits of the Model

The model we propose is a static impact analysis model and focuses only on the syntactic aspects of the code. Some semantic aspects of the code are unfortunately not taken into account in the current version of the model. An example of semantic aspects, which have not been considered in our model, is the use of wildcards and pointcuts designators (Kiczales et *al.*, 2001) in the aspect code. In fact, those wildcards can refer to any class or method in the Java code and pointcuts designators determine the moment the joinpoints are reached at runtime (call, execution, etc.). However, if a class or a method is modified, the wildcard is not impacted from a syntactic point of view as the program is still consistent and able to run. But from a semantic point of view and during execution of the program, there may be an impact as the join point related to the change is never reached.

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed a predictive change impact analysis model for aspect-oriented programs. Three change impact categories were identified:

object code impacts on aspect code, aspect code impacts on aspect code and aspect code impacts on object code. The model includes 41 impact rules.

Experiments were conducted on several aspect-oriented programs in order to show the effectiveness of the model. Two prediction quality measures were used: precision and recall. The results show that our model is quite effective in predicting impacts. Although some rules indicate uncertain impacts, they still allow the programmer to have a good idea on the changes to be done to keep the program consistent. Combined with the CIMJ model, which we developed in our previous work for Java programs, our impact analysis model could be very useful during program maintenance stages. Moreover, the proposed approach allows a better support for cascading impact analysis. Furthermore, the approach satisfies five of the seven properties of the Framework proposed by Li et *al.* (2012) characterizing impact analysis techniques. These properties are: *object*- the change set and the source analysis, *impact set*- the impacted elements of the system, *type of analysis*- static analysis or dynamic analysis, *intermediate representation*, *language support*- support various programming paradigms, *tool support*, and *empirical evaluation*. Our work could also help solving the pointcut fragility problem mentioned in some studies in the literature.

The work presented in this paper should be viewed as exploratory rather than conclusive. The model we proposed has some limitations that will be addressed in our future work. In addition, the study should be replicated on many other aspect-oriented programs in order to draw more general conclusions. As future work, we plan: (1) to address the limitations of the model, particularly by taking into account semantic relationships in aspect-oriented programs, (2) to develop a tool supporting the proposed technique, which should allow us to experiment the model on large aspect-oriented programs, (3) to improve the accuracy of our model, and (4) to perform other tests on many other aspect-oriented applications in order to have more general conclusions.

REFERENCES

Abdi M.K, Lounis H. and Sahraoui H., 2007: Analyse et prédiction de l'impact de changements dans un système à objets : Approche probabiliste. In proceedings of LMO 2009.

Abran A., April A., Dumke R., 2004: SMCMM Model to Evaluate and Improve the Quality of the Software Maintenance Process, 8th European Conference on Software Maintenance and Reengineering.

Acharya M., Robinson B., 2012: Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems, SIGSOFT'12/FSE-20, ACM, November 11–16, Cary, NC, USA.

Ackermann C., Lindvall M., 2006: Understanding Change Requests to Predict Software Impact, 30th Annual IEEE/NASA Software Engineering Workshop, pages 66 - 75, Columbia, MD, USA.

Aggarwal K.K., Singh Y. and Chhabra J.K., 2002: An integrated measure of software maintainability, Proceedings of the 2002 Reliability and Maintainability Symposium, pp. 235-241.

Ali H.O., Abd Rozan M.Z.A., Sharif A.M., 2012: Identifying Challenges of Change Impact analysis for software projects, International Conference on Innovation Management and Technology Research, pages 407-411.

Arnold R.S. and Bohner S.A., 1993: Impact analysis - towards a framework for comparison, In Proceedings of the International Conference on Software maintenance (CSM '93), pp. 292-301, Canada.

Badri L., Badri M., and St-Yves D., 2005: Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique, In Proceedings of the 12th Asia-Pacific Software Engineering Conference (Apsec'05) - APSEC. IEEE Computer Society, 167-175.

Badri L., Badri M., Joly N., 2015: Towards a Change Impact Analysis Model for Java Programs: An Empirical Evaluation, Journal of Software, vol. 10, no. 4, pp. 441-453.

Baggen R., Correia J. P., Schill K and Visser J, 2011: Standardized code quality benchmarking for improving software maintainability, Software Quality Journal - © Springer Science+Business Media.

Barros S., Bodhun Th., Escudie A., and Voidrot J.P., 1995: Supporting Impact Analysis: A semi-automated technique and associated tool, Proceedings of the 1995 IEEE Conference on Software Maintenance, pp. 42-51, Piscataway, NJ.

Bennett K. and Rajlich V., 2000: Software maintenance and evolution: a roadmap, in Proceedings of the Conference on the Future of Software Engineering, pp. 73–87, ACM, New York, NY, USA.

Bernardi M. L., Di Lucca G. A., Ceccato M., 2009: Workshop on Maintenance of Aspect Oriented Systems, 13th European Conference on Software Maintenance Reengineering, pages 273-274, Kaiserslautern, Germany.

Bernardi M.L., Di Lucca G.A, 2007: An Interprocedural Aspect Control Flow Graph to Support the Maintenance of Aspect Oriented Systems. IEEE International Conference on Software Maintenance, pages 435-444, Paris, France.

Bohner S.A. and Arnold R., 1996: Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA.

Briand L.C., Wust J., and Lounis H., 1999: Using coupling measurement for impact analysis in object-oriented systems, Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99), pp. 475 – 482.

Burrows R., Ferrari F. C., Lemos O. A.L., Garcia A., Taïani F., 2010: The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study, IEEE 21st International Symposium on Software Reliability Engineering, pages 329-338, San Jose, CA, USA.

Chaumun M. A., Kabaili H, Keller R. K., Lustman F. and Saint-Denis G., 2000: Design properties and object-oriented software changeability, in Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering, pages 45-54, Zurich, Switzerland, IEEE.

Chaumun M. A., Kabaili H., Keller R. K. and Lustman F., 2002: A change impact model for changeability assessment in object-oriented software systems, Science of Computer Programming, Volume 45, Issues 2-3, pp. 155-174.

Cho H., Gray J., Cai Y, Wong S. and Xie T., 2011: Model-Driven Impact Analysis of Software Product Lines, in Model-Driven Domain Analysis and Software Development: Architectures and Functions, Chapter 13, IGI Global.

Déhoulé F., 2014: Analyse de l'impact dans les systèmes orientés aspect (SOA) : Élaboration d'un modèle d'impact, M. Sc., Université du Québec à Trois-Rivières, Québec, Canada.

Dong Z., 2011: Aspect Oriented Programming Technology And The Strategy Of Its Implementation, International Conference on Intelligence Science and Information Engineering, pages 457-460, Wuhan, Chine.

Ebert C., and De Man, J., 2005: Requirements uncertainty: Influencing factors and concrete improvements, Proceedings of the 27th International Conference on Software Engineering.

Gethers M., Poshyvanyk D., 2010: Using relational topic models to capture coupling among classes in object-oriented software systems, In Proceedings of the 2010 IEEE International Conference on Software Maintenance, pp. 1–10.

Gradecki J. D., Lesiecki N., 2003: Mastering AspectJ, Aspect Oriented Programming in Java. Indianapolis, Indiana, USA.

Grubb P., Takang A. A., 2003: Software Maintenance: Concepts and Practice, Publisher: World Scientific Publishing Company. ISBN: 9812384251.

Harrold M. J., Jones J. A., Li T. et al., 2001: Regression test selection for Java software, in Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '01).

Hunt B., Turner B., McRitchie K., 2008: Software Maintenance Implications on Cost and Schedule, Aerospace Conference, Big Sky, MT.

Joly N., 2010: Towards a Change Impact Analysis Model for Java Programs: An Empirical Evaluation, Mémoire de maîtrise, Université du Québec à Trois-Rivières, Canada.

Kabaili H, Keller R. K. and Lustman F., 2001: A change impact model encompassing ripple effect and regression testing, in Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering, pages 25-33, Budapest, Hungary, June.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. G., 2001: An Overview of AspectJ, In J. Lindskov Knudsen, editor, European Conference on Object-oriented Programming, volume 2072 of Lecture Notes in Computer Science, pages 327–353. Springer.

Kilpinen M., 2008: The Emergence of Change at the Systems Engineering and Software Design Interface – An Investigation of Impact Analysis, PhD thesis, Cambridge University, Engineering Department.

Kumar A., Kumar R., Grover P.S., 2007: An Evaluation of Maintainability of Aspect-Oriented Systems: a Practical Approach, International Journal of Computer Science and Security (IJCSS), Pages - 1 - 9, Kuala Lumpur, Malaysia.

Kung D. C., Gao J., Hsia P., Lin J., and Toyoshima Y., 1995: Class firewall, test order, and regression testing of object-oriented programs, Journal of Object-Oriented Programming, vol. 8, no. 2, pp. 51–65.

Law J., Rothermel G., 2003: Whole Program Path-Based Dynamic Impact Analysis, Proc. of the International Conference on Software Engineering, pp. 308-318.

Lee M, Offutt A. J. and Alexander R. T., 2000: Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software, 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '00), pages 61-70, Santa Barbara, CA, August.

Lehman M. M., 1980: On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle, Journal of Systems and Software, vol. 1, no. 3.

Lehman M. M., Ramil J. F., Wernick P. D., Perry P. E., and Turski W. M., 1997: Metrics and Laws of Software Evolution – The Nineties View, Proceedings of the 4th International Software Metrics Symposium, pp. 20-32.

Lehnert S., 2011: A Taxonomy for Software Change Impact Analysis, IWPSE-EVOL'11, September 5–6, Szeged, Hungary, ACM.

Li W. and Henry S., 1995: Maintenance support for object-oriented programs, The Journal of Software Maintenance, Research and Practice, 7(2):131-147, March-April.

Li B., Sun X., Leung H. and Zhang S., 2012: A survey of code-based change impact analysis techniques, Software testing, Verification and Reliability; 23:613-646, Wiley Online Library.

Li L. and Offutt A. J., 1996: Algorithmic analysis of the impact of changes to object-oriented software, Proceedings of the IEEE International Conference on Software Maintenance, CA, USA, pp 171-184.

Mens T., Fernandez-Ramil J., and Degrandsart S., 2008: The Evolution of Eclipse, Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pp. 386-395.

Orso A., Apiwattanapong T., and Harrold M.J., 2003: Leveraging field data for impact analysis and regression testing, Proceedings of the European Software Engineering Conference, and ACM SIGSOFT Symposium on the foundations of software Engineering (ESEC/FSE'03), Helsinki, Finland.

Orso A., Apiwattanapong T., J. Law, Rothermel G., and Harrold M.J., 2004: An Empirical Comparison of Dynamic Impact Analysis Algorithms, Proceedings of the International Conference on Software Engineering (ICSE'04), pp. 491-500, Edinburg, Scotland.

Petrenko M., Rajlich V., 2009: Variable granularity for improving precision of impact analysis, Proceedings of the International Conference on Program Comprehension, pp. 10–19.

Przybylek A., 2011: Impact of aspect-oriented programming on software modularity, 15th European Conference on Software Maintenance and Reengineering, pages 369-372, Oldenbourg, Allemagne, 1-4.

Ren X., Shah F., Tip F., Ryder B. G. and Chesley O., 2004: Chianti: A tool for change Impact analysis of Java Programs, OOPSLA'04. Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.

Riaz M., Mendes E. and Tempero E., 2009: A systematic review of software maintainability prediction and metrics, Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement.

Rothermel G. and Harrold M. J., 1996: Analyzing regression test selection techniques, IEEE Transactions on Software Engineering, vol. 22, no. 8, pp. 529–551.

Rothermel G. and Harrold M. J., 1997: A safe, efficient regression test selection technique, ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2.

Ryder B. G. and Tip F., 2001: Change Impact Analysis for object-Oriented Programs. In ACM SIGPLAN-SIGSOFT.

Sabbah D., 2004: From Promise to Reality, Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04).

Shinomi I., Tamai T., 2005: Impact Analysis of Weaving in Aspect-Oriented Programming, Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 357-660.

Störzer M., 2007: Impact Analysis for AspectJ, A Critical Analysis and Tool-Based Approach to AOP, Dissertation, Passau.

St-Yves D., 2007: Dépendances et gestion des modifications dans les systèmes orientés objet: utilisation des graphes de contrôle, Thèse de maîtrise, Université du Québec à Trois-Rivières, Canada.

Sun X., Leung H., Li B., Li B., 2014: Change impact analysis and changeability assessment for change proposal: An empirical study, Journal of Systems and Software, 96, pp. 51-60, Elsevier.

Sun X., Li B., Tao C., Wen W., Zhang S., 2010: Change Impact Analysis Based on a Taxonomy of Change Types, 2010 IEEE 34th Annual Computer Software and Applications Conference, pages 373-382, Seoul.

Yau S.S., Collofello J. S., 1980: Some Stability Measures for software maintenance. IEEE Transactions on Software Engineering, 6(6): pp. 545-552.

Zhang S, Gu Z, Lin Y, Zhao J., 2008: Change Impact Analysis for AspectJ Programs, IEEE International Conference on Software Maintenance, pages 87-96, Beijing.

Zhao J., 2002: Change Impact Analysis for Aspect-Oriented Software Evolution, Proceedings of the International Workshop on Principles of Software Evolution, New York, NY, USA.

Zhao J., 2004: Measuring Coupling in Aspect-Oriented Systems, Information Processing Society of Japan (IPSJ), pages 14-15, Japon.

Zhou X., Jiang Y., Wang H., 2011: Method on change Impact Analysis for Object-oriented Program, 2011 Fourth International Conference on Intelligent Network and Intelligent Systems, IEEE.

Zimmermann T., Zeller M.A., Weissgerber, P., Diehl, S., 2005: Mining version histories to guide software changes. IEEE Transactions on Software Engineering, 31 (6), pp. 429–445.

ANNEX 1: PARTIAL LIST OF CHANGES (DÉHOULÉ, 2014).

Change	Meaning
Aspect level	
ASr	Remove an aspect
ASnm	Change name of an aspect
AST_{an}	Type : abstract to non-abstract
ASha	Inheritance : add
AShr	Inheritance :remove
Asia	Interface : add
Asir	Interface : remove
IMPr	Import : remove
Class level	
Cr	Remove a class
Cnm	Change name of a class
Chr	Remove of inheritance
Cir	Remove of interface
Mr	Remove a method

ANNEX 2: PARTIAL LIST OF IMPACT RULES (DÉHOULÉ, 2014).

Change	Impact rule
Cr	Cr(Class) -> JPr + [PCm || PCr] + [ADVr ||
Cnm	Cr -> JPr + [PCr] + [ADVr || ADVm] + I-
Chr	Chr -> Mpm(L) + Mm(L) + JPm + PCm +
Cir	Cir -> [Mr + JPr + PCr || PCm + ADVr ||
Mr	Mr -> JPr + PCm + ADVm
Mnm	Mnm -> JPm + PCm + ADVm
Mpa	Mpa | Mpm | Mpr -> JPpm + PCm +ADVm
Mpm	
Mpr	
Mtr_m	Mtrm -> JPm + PCm + ADVm
Mv_m	Mv_m-> [JPm + PCm + ADVm]
Mt_{sn}	Mt_{sn} | Mt_{ns} -> [JPm + PCm + ADVm]
Mt_{ns}	
Ar	Ar -> JPr{dg} || JPr{mu} +PCm + ADVm
Anm	Anm -> JPm{dg} || JPm{mu} +PCm +
Atm	Atm ->Mpm{mu} + Mtrm{ac} + JPm +
Av_m	Av_m -> JPm + PCm + ADVm + Mm{AS}
At_{sn}	At_{sn} | At_{ns} -> JPm{dg} || JPm{mu} +PCm + ADVm
At_{ns}	
At_{fn}	At_{fn} | At_{nf} -> JPm{dg} || JPm{mu} +PCm + ADVm
At_{nf}	
Nrtc	Nrtc -> JPr + PCr + ADVr