Towards a Change Impact Analysis Model for Java Programs: An Empirical Evaluation

Linda Badri*, Mourad Badri, Nicolas Joly

Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Québec, Canada G9A 5H7

* Corresponding author. Email: Linda.Badri@uqtr.ca
Manuscript submitted November 18, 2014; accepted March 3, 2015.
doi: 10.17706/jsw.10.4.441-453

Abstract: Software change impact analysis (or impact analysis) plays a crucial role in software maintenance and evolution. Impact analysis 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 impact analysis to predict the effects of a change before it is implemented. In this way, impact analysis 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 Java programs. The model was designed to support predictive impact analysis. It includes several impact rules based on the Java language constructs. We performed an empirical evaluation of the model using several Java programs. In order to assess the model prediction quality, we used two measures (precision and recall). The reported results show that the model is able to achieve high accuracy.

Key words: Software maintenance, software evolution, change, ripple effect, impact analysis, predictive analysis, model, impact rules, and empirical analysis.

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 [1]-[3]. 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 [4].

Software evolution faces many challenges [5]-[8]. Software maintenance is, in this context, a vital activity [9]. It is, however, costly [10], [11]. 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 [12]-[16]. 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 [17]. The software design, particularly the dependencies between its components, can make this task difficult. A change to a system, however 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 (or impact analysis) plays a crucial role in software maintenance and evolution. Bohner [18] defined change impact analysis as "the process of identifying the potential consequence of a change, or estimate what need to be modified to accomplish a change". Impact analysis allows, indeed, developers assessing the possible effects of a given source code modification [17]-[27]. Impact analysis 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 [20], [28], [29].

We present, in this paper, a new static change impact analysis model for Java programs. The model, including several impact rules based on the Java language constructs, was designed to support predictive impact analysis. We performed an empirical evaluation of the model using several Java programs. We used, in fact, two types of programs: simple Java examples (programs developed by a group of students) and design patterns. We considered in the study different changes. In order to assess the model prediction quality, we used two measures (precision and recall). In addition, we evaluated the proposed approach using the properties of the Framework proposed by Li *et al.* [23] characterizing impact analysis techniques.

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

2. Related Work

Many criteria have been proposed in literature for classifying existing impact analysis techniques (e.g., [23, 30-33]). These techniques addressed, in fact, various specific tasks of software maintenance. Existing impact analysis techniques can be static and/or dynamic (e.g., [17, 20, 23, 34]), based on the source code of the program and/or on models (e.g., [34], [35]). Static impact analysis techniques include structural static analysis, textual analysis, and historical data analysis [33], [35], [37], [38]. Impact analysis techniques can be divided in two major classes: impact analysis techniques that support predictive analysis - pre-change (e.g., [34], [39], [41]) and impact analysis techniques that support retrospective analysis - post-change (e.g., [24], [42]). 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 (e.g., [23], [24], [28], [29], [42]).

Lee *et al.* [17] proposed a static impact analysis technique for object-oriented systems (OOS). The technique is based on control flow and data flow graphs. Horwitz *et al.* [43] used a system dependencies graph. Law *et al.* [20] proposed a dynamic impact analysis technique (PathImpact) based on execution paths (whole path profiling). This technique requires an instrumented code to capture execution traces. Compared to other techniques, such as the transitive closure of calls and static slicing graphs, this technique gives more accurate results. Orso *et al.* [29] present a comparative experimental study between dynamic impact analysis techniques CoverageImpact [28] and PathImpact [20]. Weiser [44] introduced a slicing technique based on call graphs. St-Yves *et al.* [34] proposed a predictive impact analysis technique based on control call graphs, which are a reduced form of traditional control flow graphs. Control call graphs are, in fact, more precise models than traditional call graphs. The authors also showed the limitations in terms of accuracy of impact analysis based solely on direct call graphs. Abdi *et al.* [45] proposed an approach based on Bayesian networks to analyze and predict change impact in OOS. Li *et al.* [23] proposed a Framework for characterizing code-based impact analysis techniques. The Framework includes seven main properties including the objective of the impact analysis, the type of the supported analysis, the language support, and so forth. This Framework has been designed from a survey on several studies on the subject.

Chaumun *et al.* [27, 39] developed an impact analysis technique for C ++ programs. The authors identified a set of unit changes, that can be made on the code of a C ++ program, affecting the structure of classes. The considered unit changes may occur at different levels: class level (e.g., adding a class, deleting a class, etc.), method level (e.g., adding a method, deleting a method, etc.) and attribute level (e.g., adding an attribute, deleting an attribute, etc.). Chaumun *et al.* established a list of 19 class-level changes, 35 method-level changes and 12 attribute-level changes for a total of 66 changes. For each change, the authors proposed an impact rule defined according to the relations between classes. The considered relations are aggregation (G), association (S), invocation (I) and inheritance (H). The authors also included the specific relationship "friendship" (F) of the C ++ language and the local self-reference (L) when the impact is found within the class where the change occurred. Kabaili *et al.* [42] have extended the work of Chaumun *et al.* by adapting the model for the Java language. This resulted in 15 class-level changes, 25 method-level changes and 12 attribute-level changes for a total of 52 changes. The 14 unit changes that have been removed from the original model correspond, in fact, to the notion of virtual class of the C++ language and the relationship "friendship" (F); notions not existing in Java. Kabaili *et al.* also discussed the use of Chaumun's impact analysis model, adapted to Java, to support ripple effect analysis (several levels of impact: direct and indirect). As Chaumun's model, adapted for Java by Kabaili *et al.*, is the most complete and the most similar to the model presented in this work, we will use it as a baseline.

3. Change Impact Analysis Model

We present in this section the IMC ((Impact analysis Model for Changes in Java) model that we propose in this paper. The model specifies for each type of change a set of impacts, providing useful information for analyzing the impact on several levels (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.

3.1. Types of Change: Structural and Non-Structural Changes

A structural change is a change that affects the structure of the class and is visible in a class diagram; for example the addition or removal of a method. A non-structural change is a change that does not affect the structure of the class and is not visible in a class diagram. Non-structural changes are, in fact, possible within the method bodies; for example, adding or removing a method call. The IMC model includes 44 structural changes divided into three levels (for more details, see Appendix A): 3 class-level changes (e.g., adding a class, deleting a class, etc.), 23 method-level changes (e.g., adding a parameter, changing a return type, etc.) and 13 attribute-level changes (e.g., adding an attribute, changing an attribute visibility, etc.). The IMC model includes also a total of 26 non-structural changes (adding a declaration, removing a declaration, adding an attribute initialisation, removing an attribute initialisation, adding a call to an inherited method, removing a call to an inherited method, etc.). For the complete list of non-structural changes, see Appendix B.

3.2. Concept of Certainty

The IMC model uses the notion of certainty, a concept which does not exist in the Chaumun model (MC). The notion of certainty allows basically mitigating the information provided by the impact analysis using the IMC model. To illustrate this concept, let us consider two simple examples. As a first example, let us consider the atomic change "removing a class attribute". This removal will impact all uses of this attribute. To compile the code after removing the attribute, we must also remove all its uses. 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 a class attribute". Normally, if we add an attribute to a class, 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 attribute. In this case, we are talking about an impact that is uncertain (uncertainty of the impact). The IMC 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 IMC model makes a difference between what will be impacted and what

could possibly be impacted. The MC model does not make this distinction in its impact rules.

3.3. Relationships between Classes

For the IMC model, the relationships between classes are the basis for the definition of the change impact rules. These rules aim at specifying each impact and its location in the code. Four relationships between classes were considered: the association relationship (noted A), the inheritance relationship (noted H), the ancestor-descendent relationship (noted Hs - when the ancestor is impacted by the descendent) and the pseudo-relationship L (class in question). Classes can have more than one relationship between them.

3.4. Impact rules

The IMC model has been designed to support both predictive and retrospective analyzes. It is important to notice, however, that some impacts are only identifiable in retrospective analysis. Also, in predictive analysis, there are some impacts that cannot be evaluated because the model does not take into account the intentions of the change. This is due primarily to the concept of certainty of the model. For example, let us consider the change "add a class", which has the impact rule " [Mpa (A)] + [Aa (A)] + [Nad (A)] " that is, adding parameters (class type), adding class attributes and adding object class declarations in associated classes. If we add a class, it is that we have the intension to use it (in general). In predictive analysis, however, we cannot apply this rule because the only premise that we have is the addition of the class. No further information on classes that will eventually have a relationship with the added class is provided (available), which prevents the prediction of the slightest impact. It is in that sense that some impacts (all impacts in the case of adding a class) cannot be identified in predictive analysis. To better understand the structure and the use of impact rules, we give in what follows a few simple examples (see Appendix C for the rest of the rules).

As a first example, let us consider the addition of a method: «Ma ▶ Ma {Rr} (O: Sab) (H) + [Nam (L, A, H)],» «Ma» is the element of change and arrow «▶» is the impact rule. In the impact rule, each impact member is separated by the plus sign «+». To add a method, we have two impact elements: adding a method «Ma» and adding call (s) to this method «Nam». Brackets «[]» around the impact element «Nam» mean uncertainty. For the impact element «Ma», the «Rr» indication is found between braces «{}». Braces mean a specification of the impact element, «Rr» means «redefinition». Therefore, the impact element «Adding a method» will be a redefinition of the method added. «O: Sab» indicates that this impact element will be mandatory «O» (certain) if the added method is abstract, «Sab» otherwise optional (uncertain). «(H)» and «(L, A, H)», corresponding respectively to the impact elements «Ma» and «Nam», are indicators of relationships between classes. In this case, the impact will correspond to adding a method in classes that inherit «H», and to adding method calls in classes that inherit «H», classes that are associated «A», and the class itself «L» (local). If we take a complete reading of the rule, this means that when adding a method the impact will be: (1) to redefine this method, in classes that inherit, with certainty only if the added method is abstract, and (2) to add call (s) to the method added with uncertainty in the class itself, the associated classes and inheriting classes.

As a second example, we will partially interpret the impact rule of an attribute that goes from static to non-static «At_{sn} ▶ Mt_{sn} {Ru} (L) || Nra {Rms} (L) + Nra (A)». The impact «Mt_{sn} {Ru} (L) || Nra {Rms} (L)» means that methods using the class attribute will go from static to non-static «Mt_{sn}» or «||» the attribute is removed in static methods «Nra» class. In some rules, we find the sign «&&» instead of «||», which means that we have two changes in the impact element, not one or the other. The complete list of structural and non-structural changes is given in Appendix A and Appendix B. The list of impact rules is given in Appendix C.

4. Empirical Evaluation

In order to evaluate the ability of the IMC model to accurately predict the impact of changes, we conducted an empirical study in two stages. As a first stage, we evaluated the model based on simple examples. This first series of experiments was essentially designed to make a comparison between our model and the MC model. We used simple Java programs. As a second stage, we evaluated the IMC model using some design patterns. The goal this

time was to evaluate the performance of the IMC model. A part of these experiments were performed by including the MC model. In order to measure and compare objectively the performance of both models, we used in this study two measures: precision and recall. In addition, we considered the following six changes: removing an attribute, changing an attribute from public to a private, changing an attribute from static to non-static, changing an abstract class to a non-abstract class, changing a static method to a non-static method and changing the type of a parameter of a method.

4.1. Metrics

To evaluate the quality of the prediction of the impact analysis models, we used two well-known measures [46, 47]: recall and precision. Impact analysis may, in fact, have some false positives (elements in the impact set that aren't really impacted) and false negatives (elements really impacted that aren't identified in the impact analysis) [23]. In the evaluation, we obtained three types of results: the number of actual impacts due to a change, the number of impacts predicted by a model and the number of impacts correctly predicted by a model. The recall, which is an inverse measure of false negatives, is the ability of a model to predict all real impacts (percentage 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 the impacts correctly (percentage 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.

4.2. Simple Experiments

In this section, we used a set of similar programs, which have been developed by students in one of the programming courses of the Bachelor of computer science in our department. We selected, in fact, a group of six students who achieved the best ratings. We had 39 Java classes in total.

At first, we evaluated the impact of 21 different changes on the first version of the code (compilable), using the two models IMC and MC. Then, a second version of compilable source code is obtained from the first version by instantiating the considered changes. Finally, we compared the two versions of the programs in order to determine the actual impacts due to the changes we applied. For purposes of comparison of the two models, the results for this series of experiments were calculated in terms of number of impacted classes, given that the MC model focusses only on impact at class level. Results are given in Table 1. CP indicates the number of classes predicted by a model, CR indicates the number of really impacted classes and CC indicates the number of classes correctly predicted by a model. As it can be seen from Table 1, the recall and the precision of the IMC model are both equal to 100%, meaning that our model predicted only real impacts and has predicted them all. The MC model obtains a recall of 57% and a precision of 48%.

Table 1. Results of the Simple Experiments.

	CP	CC	CR	Recall	Precision
IMC	35	35	35	100%	100%
MC	42	20		57%	48%

4.3. Advanced Experiments

In this section, we used the code of some design patterns to evaluate the IMC model. We considered particularly the following design patterns: abstract factory, adapter, bridge, builder, chain of responsibility, command, composite, decorator, façade, interpreter and memento. The code of the selected design patterns varies in terms of number of classes between 4 and 10 classes. We considered this time 35 different changes in

total. As for the first series of experiments, the considered changes have been instantiated. In some cases, there has been a ripple effect of changes. For purposes of comparison between the two models, each impact causing a ripple effect was treated as a new change. This second series of experiments was conducted in two steps.

Table 2. Results of Advanced Experiments (Step I).

Number of changes	Accuracy		Reality
	IMC	MC	
35	55/ 71	27/41	57
Recall	96,49%	47,93%	
Precision	77,46%	65,85%	

In a first step, given that the evaluation also focused on the MC model, and to facilitate comparison, the evaluation of the IMC model was also performed by considering only the impact at class level. Table 2 shows the results obtained. For the 35 changes we considered, there were 57 impacted classes. The IMC model predicted 71 impacted classes of which 55 were actually impacted. In fact, 16 predicted classes were not really impacted. The IMC model in this series of experiments has a precision of 77.46% and a recall of 96.49%. The MC model, meanwhile, predicted 41 impacted classes of which 27 were actually impacted. The MC model, in this series of experiments, has a precision of 65.85% and a recall of 47.93%. As it can be seen from Table 2, results indicate better overall performance of the IMC model compared to the MC model. Until now, for comparison purposes, the results were obtained according to the level of precision of the MC model (class level). We wanted to know particularly if the models predict correctly the impacted classes.

Table 3. Results of Advanced Experiments (Step II).

Number of changes	IMC	
	Precision	Real
35	50/51 - 18/56	71
Precision (certain)	98,04%	
Precision (uncertain)	32,14%	
Recall	95,77%	

In a second step, we focused on the evaluation of the IMC model alone. Table 3 shows the results obtained. The impacts predicted with certainty by the IMC model are 50/51 (predicted 51 impacts of which 50 were actually impacted), with a precision of 98.04%. It is in the uncertain impacts predicted by the IMC model where we found erroneous predictions. Only 32.14% (18/56 -56 predicted impacts of which 18 were actually impacted) of predicted impacts with uncertainty have actually occurred. Uncertain impacts represent for almost half of the total predicted impacts (56 of 107 (51+56)). All the erroneous predictions are made by the prediction of uncertain impacts and only one by the prediction of impacts that are certain. In total, 26% of the impacts predicted correctly by the IMC model were predicted by the uncertain impacts (18 of the 68 (50+18) correctly predicted impacts).

4.4. Discussion

Table 4 summarizes the differences between the two models IMC and MC. From Table 4, it can be seen that the models have some differences according to the four criteria: structural changes, non-structural changes, impact rules and impact level. The non-structural changes are absent from the MC model, which is due to the fact that the MC model focuses only on the class-level, contrary to the IMC model which focuses on the two levels: class and method. Moreover, various changes (such as adding an attribute, changing the visibility of an attribute from protected to public, private to public and private to protected, etc.) has no impact in the case of the MC model.

The other weakness of the MC model is in relation with its interpretation. The MC model indicates, in fact, only which classes will be impacted after the instantiation of a change. The model does not give any detail on the nature of the impact. According to the model, if a change made in class A has an impact on a class B, it does not tell if there is one or more changes to be made in class B.

Table 4. General Comparison of Models.

Criterion	IMC	MC
Structural changes	41	52
Non-structural changes	26	0
Impact rules	41	33
Impact (accuracy) level	Method and Class	Class

Furthermore, according to its authors, the MC model predicts also the impacts that are uncertain. If we combine, for the IMC model, the results corresponding to the impacts that are certain and the results corresponding to the impacts that are uncertain, we obtain a precision of 63.55%, slightly lower than the precision of the model MC. It should nevertheless understand that the imprecision comes from the fact that the IMC model makes the difference between the impacts that are certain and those that are not. So, overall, the IMC model is not more accurate than the MC model (if we consider that the MC model predicts also the impacts that are uncertain), but it allows contrary to the MC model making the difference explicitly between what it is certain and what it is uncertain. It is, however, important to notice that the slight different in terms of precision between the two models may vary according to the performed tests. The IMC model, however, presents a recall (99.77%) that is much higher than the recall of the MC model (47.93%). This means that the IMC model identifies almost all the real impacts and the double of the real impacts identified by the MC model. So, we can say that despite the fact that the overall precision of the two models are comparable, the IMC model is superior in its ability to identify the real impacts.

The different experiments we conducted show that the IMC model achieves results close to reality. However, these experiments also showed that the model has some limitations, in particular when adding a new class. The reason of this weakness is simply due to the fact that the IMC model (the MC model also) seeks impacts in classes that have any relationship with the modified class (added class). But, in this case, when adding a new class (with no relationships with any classes), it is necessary to know with what classes it will create links to make prediction.

In addition, the IMC model as mentioned above offers a greater accuracy in the impacts (in terms of impacts and their location). This allows a better support for the different tasks that can be associated with a given change. Moreover, with the MC model, it is difficult to analyze the ripple effect (cascading impact) efficiently because, even if it correctly predicts that a class will be impacted, the model do not give any information on the nature of the impact. In other words, if for example the real impact was the change in a signature of a given method, the model only precise the impacted class, class containing the impacted method, without any other information (in this case the method impacted and how it is impacted). The IMC model, by clearly specifying what change is expected after a given initial change, can make the link between the different impacts and predict the ripple effect. To better illustrate this dimension, let us consider the following example given in Fig. 1.

If the « count » attribute of class A is removed, the MC model predicts that class A will be impacted (the rule is « S + L »). Knowing that there is an impact on A, the MC model does not predict the impact that will

take place later on classes B and C. In the case of the IMC model, if the « count » attribute is removed from the class A, the model predicts an impact: « the withdrawal of the accessor ». The impact being accurate, and not just the name of a class as predicted by the MC model, we can use again the IMC model and predict that class B will be impacted. The IMC model, by clearly specifying what change is expected after a given initial change, can make the link between the different impacts and predict the ripple effect.

```
public class B extends A
{
        public void count( )
        {  int val = 10 - getCount();}
}

public class C extends A
{}

public class A
{
        private int count;
        public A(int count)
        {  this.count = count;}
        public int getCount( )
        {  return count;}
}
```

Fig. 1. A simple example for cascading impact.

5. Conclusions and Future Work

Software change impact analysis plays a crucial role in software maintenance and evolution. Impact analysis can be used, in fact, to support various important tasks such as: planning changes, assessing the cost of changes, implementing changes, tracking the effects of changes and regression testing. We presented, in this paper, a new change impact analysis model for Java programs. The model includes several impact rules that are based on the Java language constructs. The model was designed to support in particular predictive impact analysis.

We performed an empirical evaluation of the model using several Java programs. We considered different types of (structural and non-structural) changes. The quality of the prediction of the model has been evaluated using two well-known measures: precision and recall. Results show that the model is able to achieve high accuracy. In particular, the proposed model presents a high recall (more than 95% in all cases), which means that the model identifies almost all the real impacts. Moreover, it allows a better support for cascading impact analysis. Furthermore, the proposed technique satisfies five of the seven properties of the Framework proposed by Li *et al.* [23] 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*.

The performed study should be replicated on many other Java programs in order to draw more general conclusions. The findings in this paper should be viewed as exploratory and indicative rather than conclusive. As future work, we plan: 1) to develop a tool supporting the proposed technique, which should allow us to experiment the model on large Java programs, 2) to improve the precision of the model, 3) to explore the integration of machine learning methods to improve the global accuracy of the model, and finally 4) to replicate the study on various Java programs to be able to give more general conclusions.

Appendix A Structural Changes

Change	Meaning	Change	Meaning
Class level		Mpc_t	Parameter: type change
Ca	Add a class	Mpc_n	Parameter: name change
Cr	Remove a class	Mwa	Throws : add
Cia	Interface : add	Mwr	Throws : remove
Cir	Interface : remove	Mr_{vo}	Return : void to Object
Cha	Inheritance :add	Mr_{oo}	Return ã „ Œ ‡ … – ‘ „ Œ ‡ … – ï
Chr	Inheritance :remove	Mr_{ov}	Return : Object to void
Ct_{na}	Type : non-abstract to abstract	Ma	Method : add
Ct_{an}	Type : abstract to non-abstract	Mr	Method : remove
Method level		Class attributes level	
Mv_{ui}	Visibility : public to private	Av_{ui}	Visibility : public to private
Mv_{uo}	Visibility : public to protected	Av_{uo}	Visibility : public to protected
Mv_{iu}	Visibility : private to public	Av_{iu}	Visibility : private to public
Mv_{io}	Visibility : private to protected	Av_{io}	Visibility : private to protected
Mv_{ou}	Visibility : protected to public	Av_{ou}	Visibility : protected to public
Mv_{oi}	Visibility : protected to private	Av_{oi}	Visibility : protected to private
Mt_{na}	Type : non-abstract to abstract	At_{sn}	Type : static to non-static
Mt_{an}	Type : abstract to non-abstract	At_{ns}	Type : non-static to static
Mt_{sn}	Type : static to non-static	At_{fn}	Type : final to non- final
Mt_{ns}	Type : non-static to static	At_{nf}	Type : non-final to final
Mt_{fn}	Type : final to non- final	Aa	Attribute : add
Mt_{nf}	Type : non-final to final	Ar	Attribute : remove
Mpa	Parameter: add	Ata	Attribute type change
Mpr	Parameter: remove		

Appendix B Non-Structural Changes

Change	Meaning
Nad	Adding a declaration
Nrd	Removing a declaration
Nai	Adding initialisation
Nri	Removing initialisation
Nas	Adding the «super» keyword
Naah	Adding the use of an inherited attribute
Namh	Adding call to inherited method
Nrah	Removing the use of an inherited attribute
Nrmh	Removing call to inherited method
Nrs	Removing the «super» keyword
Nrm	Removing method call
Nam	Adding method call
Nrcm	Removing method body (replaced by «;»)
Nacm	Adding method body (replacement of «;»)
Nrms	Removing call to static method
Nams	Adding call to static method
Natc	Adding a «try/catch»
Nrtc	Removing a «try/catch»
Nar	Adding a «return»
Nrr	Removing a «return»
Naa	Adding attribute use
Nra	Removing attribute use
Naf	Adding assignment statement
Nrf	Removing assignment statement

Appendix C IMPACT RULES.

Change	Impact Rules
Ca	[Mpa(A)]+[Aa(A,L)]+[Nad(A,L)]
Cr	Mpr(a,H)+Ar(a,H)+Nrd(a,H)+Nri(a,H)
Cia	Ma{Rr}(L)
Cir	Mr{Rr}(L)
Cha	Ma{Rmah}(L)+[Nas(L){Rc}]+[Ma{Rr}(L)+^Nas(L){Rr}]+[Naah(L,A,H)]+[Namh(L,A,H)]+[Av$_{uo}$(H^s)]+[Av$_{io}$(H^s)]
Chr	Nrah(L)+Nrmh(L)+Nrs(L)+[Mr{Rmah}(L)]+[Mr{Rr}(L)]+[Av$_{ou}$(H^s)]+[Av$_{oi}$(H^s)]
Ct$_{na}$	Nri(A,H)
Ct$_{an}$	Mt$_{an}${Rma}(L)+[Nai(L,A,H)]
Mv$_{ui}$	Nrm(A,H)
Mv$_{iu}$	*f* • ï
Mv$_{io}$	[Nam(A,H)]+[Nas{Rr}(H)]
Mv$_{oi}$	Nrm(A,H)+Nrs{Rr}(H)
Mt$_{na}$	Nrcm(L)+Ma{Rr}(H)+Ct$_{na}$(L)
Mt$_{an}$	Nacm(L)+[Mr{Rr}(H)]
Mt$_{sn}$	Nrms(A,H)+[Nam(A,H)]
Mt$_{ns}$	Nrm(A,G)+[Nam(A,G)]
Mt$_{fn}$	[Ma{Rr}(h)]
Mt$_{nf}$	Mr{Rr}(H)
Mpa	Mpa ||Ma(O:Sab){R}(H)+Nrm&&Nam(A)+[Ua](L)
Mpr	Mpr||Ma(O:Sab){Rr}(H)+Nrm&&Nam(A)+Ur(L)||Nad(L)
Mpc$_t$	Mpc$_t$||Ma (O:Sab){Rr}(H)+Nrm&&Nam{Rh}(A,L)
Mwa	Mwa{Rsm}(H^s)+Mwa||Natc{Rmc}(A,L,H)
Mwr	[Mwr{Rsm}(H^s)]+[Mwr||Nrtc{Rmc}(A,H,L)
Mr$_{vo}$	Nar(L)+Mr$_{vo}${Rr}(H)+[Nrm&&Nam{Rmc}(A,H,L)]
Mr$_{oo}$	Mr$_{oo}${Rr}(H)+Nrm&&Nam{o:Rh}(A,H,L)+Nrr&&Nar{Rr}(L)
Mr$_{ov}$	Nrr(L)+ Mr$_{ov}${Rr}(H)+Nrm&&Nam{Rmc}(A,H,L)
Ma	Ma{Rr}(O:Sab)(H)+[Nam(L,A,H)]
Mr	[Mr(H)]+Nrm(L,A)+[Nrm(H)]
Av$_{ui}$	Nra(H,A)+[Ma{Rmu}(L)]+[Ma{Rac}(L)]
Av$_{uo}$	Nra(A)+[Ma{Rac}(L)]+[Ma{Rmu}(L)]
Av$_{iu}$	[Naa(H,A)]+[Mr{Rmu}]+[Mr{Rac}(L)]
Av$_{io}$	[Naa(H)]+[Mr{Rmu}(L)]+[Mr{Rac}(L)]
Av$_{ou}$	[Naa(A)]+[Mr{Rmu}(L)]+[Mr{Rac}(L)]
Av$_{oi}$	Nra(H)+[Ma{Rmu}(L)]+[Ma{Rac}(L)]
At$_{sn}$	{Ru}(L)||Nra{Rms}(L)+Nra(A)
At$_{ns}$	[Mt$_{ns}${Ru}(L)]
At$_{fn}$	[Naf(L,H,A)]+[Ma{Rmu}(L)]
At$_{nf}$	Nrf(L,H,A)+Mr{Rmu}(L)
Ata	Mr$_{oo}${Rac}(L)+[Nra&&Naa{Rh}(A,H,L)]+[Mpc{Rh+Rmu}(L)]
Aa	[Ma{Rac}(L)]+[Ma{Rmu}(L)]+[Naa(L,H,A)]
Ar	Mr{Rac}(L)+Mr{Rmu}(L)+Nra(L,H,A)+[Mpr{Rc}(L)]

Acknowledgment

This project was financially supported by NSERC (National Sciences and Engineering Research Council of Canada).

References

[1] Kung, D. C., Gao, J, Hsia, P., Lin, J, & Toyoshima, Y. (1995). Class firewall, test order, and regression testing of object-oriented Programs. Journal of Object-Oriented Programming, 8(2), 51 65.

[2] Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. Proceedings of the ACM Transactions on Software Engineering and Methodology

[3] Harrold, M. J., Jones, J. A., Li, T. et al. (2001). Regression test selection for Java software. Proceedings of the Conference on Object ” ‹ ‡ • – ‡ † ” ‘ ‰ ” ƒ • • ‹ • ‰ á › • – ‡ • • á ƒ • ‰ — ƒ ‰ ‡ • ƒ. • † ’ ’ Ž ‹ … ƒ – ‹ ‘ • •

[4] Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. Proceedings of the IEEE Transactions on Software Engineering (pp. 529 551).

[5] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, P. E., & Turski, W. M. (1997). Metrics and laws of software evolution the nineties view. Proceedings of the 4th International Software Metrics Symposium (pp. 20-32).

[6] Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software, 1(3).

[7] Ebert, C., & De, M. J. (2005). Requirements uncertainty: Influencing factors and concrete improvements. Proceedings of the 27th International Conference on Software Engineering

[8] Mens, T., Ramil, J. F., & Degrandsart, S. (2008). The evolution of edipse. Proceedings of the IEEE International Conference on Software Maintenance (ICSM) (pp. 386-395).

[9] Bennett, K., & Rajlich, V. (2000). Software maintenance and evolution: a roadmap. Proceedings of the Conference on the Future of Software Engineering (pp. 73 87), ACM, New York, NY, USA.

[10] Grubb, P., & Takang, A. A. (2003). Software maintenance: Concepts and practice. World Scientific Publishing Company.

[11] Abran, A., April, A., & Reiner, R. D. (2004). Smcmm model to evaluate and improve the quality of the software maintenance process. Proceedings of the 8th European Conference on Software Maintenance and Reengineering.

[12] Barros, S., Bodhun, T., Escudie, A., & 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.

[13] Aggarwal, K. K., Singh, Y., Chhabra, & J. K. (2002). An integrated measure of software maintainability. Proceedings of the 2002 Reliability and Maintainability Symposium (pp. 235-241).

[14] Riaz, M., Emilia, M., & Ewan, T. (2009). A systematic review of software maintainability prediction and metrics. Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement

[15] Robert, B., José, P. C., Katrin, S., & Joost, V. (2011). Standardized code quality benchmarking for improving software maintainability. Software Quality Journal, Springer Science+Business Media.

[16] Hyun, C., Jeff, G., Yuan, F. C., Sonny, W. & Tao, X. (2011). Model-driven impact analysis of software product lines, Model-Driven Domain Analysis and Software Development: Architectures and Functions. IGI Global.

[17] Michelle, L., Jefferson, O. A., & Roger, T. A (2000). Algorithmic analysis of the impacts of changes to object-oriented software. Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (pp. 61-70).

[18] Bohner, S. A., & Arnold, R. (1996). Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA.

[19] Briand, L. C., Wust, J., & Lounis, H. (1999). Using coupling measurement for impact analysis in object-oriented systems. Proceedings of the IEEE International Conference on Software Maintenance (pp. 475 482).

[20] Law, J, & Rothermel, G. (2003). Whole program path-based dynamic impact analysis. Proceedings of the International Conference on Software Engineering (pp. 308-318).

[21] Wei, L., & Sallie, H. (1995). Maintenance support for object-oriented programs. Journal of Software Maintenance, Research and Practice, 7(2), 131-147.

[22] Li, L., & Offutt, A. J.(1996). Algorithmic analysis of the impact of changes to object-oriented software. Proceedings of the IEEE International Conference on Software Maintenance(pp. 171-184).

[23] Li, B. X., Sun, X. B., Leung, H., & Zhang, S., (2012). A survey of code-based change impact analysis techniques Software testing, Verification and Reliability

[24] Ren, X. X., Shah, F., Tip, F., Ryder, B. G., & Chesley, O. (2004). A tool for change impact analysis of Java Programs. Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications

[25] Barbara, G. R., & Tip, F. (2001). Change Impact Analysis for Object-Oriented Programs

[26] Yau, S. S., & Collofello, J. S(1980). Some stability measures for software maintenance. IEEE Transactions on Software Engineering, 6(6), 545-552.

[27] Chaumun, M. A., Hind, K., Rudolf, K. K., & Lustman, F. (2002). A change impact model for changeability assessment in object-oriented software systems. Science of Computer Programming, 155-174.

[28] Orso, A., Apiwattanapong, T., & Harrold, M. J. (2003). Leveraging field data for impact analysis and regression testing Proceedings of the European Software Engineering Conference, and ACM SIGSOFT ›•'•‹—• '• –Š‡ ˆ'—•†ƒ–‹•• '^ •'^–™ƒ"‡ •,‰elsinki, Finland. ïvy

[29] Orso, A., Apiwattanapong, T., Law, J, Rothermel, G., & Harrold, M. J. (2004). An empirical comparison of dynamic impact analysis algorithms. Proceedings of the International Conference on Software Engineering (pp. 491-500).

[30] Arnold, R. S., & Bohner, S. A. (1993). Impact analysis- towards a framework for comparison. Proceedings of the International Conference on Software Maintenance(pp. 292-301).

[31] 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.

[32] Steffen, L. (2011). A Taxonomy for software Change Impact Analysis, Szeged, Hungary, ACM.

[33] Sun, X. B., Leung, H., Li, B., & Li, B. X. (2014). Change impact analysis and changeability assessment for change proposal: An empirical study. Journal of Systems and Software, 86, 51-60.

[34] Daniel, S. Y. (2007). Dépendances et gestion des modifications dans les systèmes orientés objet : utilisation des graphes de contrôle, Thèse de maîrise, Université du Québec à Trois-Rivières, Canada.

[35] Petrenko, M., & Rajlich, V. (2009). Variable granularity for improving precision of impact analysis Proceedings of the International Conference on Program Comprehension(pp. 10 19).

[36] Acharya, M., & Robinson, B. (2012). Practical change Impact analysis Based on Static Program Slicing for Industrial Software Systems

[37] Gethers, M., & Poshyvanyk, D.(2010). Using relational topic models to capture coupling among classes in object-oriented software systems. Proceedings of the 2010 IEEE International Conference on Software Maintenance(pp. 1 10).

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

[39] Chaumun, M. A., Hind, K., Rudolf, K. K., François, L., & Guy, S. D. (2000). Design properties and object-oriented software changeability. Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering(pp. 45-54).

[40] Badri, L., Badri, M., & Yves, D.(2005) Supporting predictive change impact analysis: A control call graph based technique. Proceedings of the 12th Asia-Pacific Software Engineering Conference(pp. 167-175).

[41] Abdi, M. K., Lounis, H., & Sahraoui, H. (2007). •ƒŽ›•‡ ‡– '"±†‹…–‹•† ŽïŒ•'ƒ…– †‡ …Šƒ•système à objets : Approche probabiliste. In proceedings of LMO 2009.

[42] Hind, K., Rudolf, K. K., & François, L. (2001). A change impact model encompassing ripple effect and regression testing. Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering(pp. 25-33).

[43] Horwitz, S, Reps, T. & Binkley, W. D. (2004). Interprocedural slicing using dependence graphs. ACM SIGPLAN Notices, 39(4).
[44] Weiser, M. (1979). Program slices: formal, psychological, and practical investigations of program abstraction method, PhD thesis, University of Michigan, Ann Arbor.
[45] Abdi, M. K., Lounis, H., & Sahraoui, H. (2009). Predicting change impact in object-oriented applications with bayesian networks. Proceedings of the 33rd Annual IEEE International Computer Software and Application Conference
[46] Van, R. C.J. (1979). Information Retrieval. Butterworths Wiley, 2nd edition, London.
[47] Hassan, A.E., & Holt, R. C. (2004). Predicting Change Propagation in Software Systems, Software Maintenance. Proceedings of the 20th IEEE International Conference on Software Maintenance

Linda Badri is a full professor of computer science (software engineering) at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. Her main research interest include object and aspect-oriented software engineering, software quality attributes, web engineering, change impact analysis, regression testing, software maintenance as well as various topics of software engineering.

Mourad Badri is a full professor of computer science (software engineering) at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. His main research interest include object, aspect and agent oriented software engineering, software quality attributes, software quality assurance, program analysis, software maintenance and evolution as well as various topics of software engineering.

Nicolas Joly is a student of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He finished his master in computer science (software engineering) at the University of Quebec at Trois-Rivières. His main research interest include object-oriented programming, change impact analysis as well as various topics of software engineering.