Noname manuscript No.
(will be inserted by the editor)

A Model for Estimating Change Propagation in Software

Kecia A. M. Ferreira · Mariza A. S. Bigonha · Roberto S. Bigonha · Bernardo N. de Lima · Bárbara M. Gomes · Luiz Felipe O. Mendes

Received: date / Accepted: date

Abstract A major issue in software maintenance is *change propagation*. A software engineer should be able to assess the impact of a change in a software system, so that the effort to accomplish the maintenance may be properly estimated. We define a novel model, named K3B, for estimating *change propagation* impact. The model aims to predict how far a set of changes will propagate throughout the system. K3B is a stochastic model that has as input parameters about the system and the number of modules which will be initially changed. K3B returns the estimated number of *change steps*, considering that a module may be changed more than once during a modification process. We provide the implementation of K3B for object-oriented programs. We compare our implementation with data from an artificial scenario, given by simulation, as well as with data from a real scenario, given by historical data. We found strong correlation between the results given by K3B and the results observed in the simulation, as well as with historical data of change propagation. K3B may be used for comparing software systems from the viewpoint of change impact. The model may aid software engineers in allocating proper resources to the maintenance tasks.

Keywords change propagation · change impact analysis (CIA) · software maintenance · software metrics · comparing programs · stochastic process · Markov Chain

The authors would like to thank FAPEMIG - Fundação de Amparo à Pesquisa do Estado de Minas Gerais, which has sponsored Bárbara M. Gomes (CEFET-MG) and Luiz F. O. Mendes (UFMG) as undergraduated students.

F. Author
Department of Computing, CEFET-MG, Belo Horizonte-MG, Brazil
Tel.: +55-31-33196870
Fax: +55-31-33196722
E-mail: kecia@decom.cefetmg.br

S. Author
Department of Computer Science, UFMG, Belo Horizonte-MG, Brazil
Tel.: +55-31-34095860
Fax: +55-31-34095858
E-mail: mariza@dcc.ufmg.br

1 Introduction

Software maintenance is the most costly activity in the software life-cycle. About 2/3 of the total cost of a software system is due to maintenance activities, whereas only 1/3 is due to the development of the system (Sommerville 2011). One of the most important problems in software maintenance is the *change propagation* effect (Hassan and Holt 2004), which refers to the process in which a change in a particular *module* of the system, or in a set of modules, causes changes in other modules of the system successively. A module, in this context, is a piece of software. In a object-oriented project, for instance, a class can be considered as a module. When performing a change in a set of modules of a system, the developer should be able to identify what else must be changed in order to keep it consistent with the initial change. This problem is also referenced in the literature as *change impact analysis* (CIA) or *ripple effect* (Li et al 2012). Due to the importance of this issue, some works have been carried out in the last decades with the aim to investigate how changes propagate throughout a software system (Hassan and Holt 2004; Zimmermann et al 2005; Li et al 2009; Herzig 2010; Herzig and Zeller 2011; Geipel and Schweitzer 2012; Li et al 2012).

Ideally, a small change performed in a module should impact a very few other modules (Meyer 1997). The easiness to perform changes in a software system is affected by the internal characteristics of the system, such as information hiding, coupling and cohesion (Pressman 2009). Therefore, when facing a high pervasive change propagation, the developer should identify the causes of this effect, and, hence, refactor the software structure in order to improve its internal quality so as to lower the change impact rate. However, we still do not have resources to perform such analysis in software systems properly. There is not a consolidated method to estimate or to evaluate the effect of change propagation in software.

This work presents a novel model, named K3B, for estimating change propagation in software systems. The aim of the model is to predict the potential *change propagation* effect having as basis the structural characteristics of the software system, which is assessed by means of software metrics.

K3B represents the change propagation process as a stochastic process. In short, the main idea of the model is described following. The modification process starts with the changes performed in a set of modules. From these initial changes, other modules can be affected and, then, can be also changed successively. The process of changing a module occurs as follows: i) change starting: open the module source file for editing; ii) code changing: perform the necessary changes in the module source code to bring it back to a consistent state; iii) change propagation: propagate the impact of the changes, if any, to each module connected to it and immediately start the process of changing in each connected module accordingly; iv) change finishing: close the module source file. We call *change step* the event of starting or finishing a change in a module of the system. The idea of *change step* introduced in the proposed model considers the fact that in a real scenario of maintenance, a module is taken to be changed in a given moment and its changing process must be declared as concluded in a given moment too. The whole modification process is accomplished when all the necessary changes are finished, i.e, when there are no more modules to change in the system relating to the initial set of changes.

The model proposed in this paper aims to estimate the number of *change steps* which will be necessary to modify a system so that all changes resulting from a original set of changes are performed. The main application of this model is, then, a resource for estimating the cost to keep the system consistent with the initial changes.

The K3B model has as entry the size of the software, given in number of modules, the number of modules which will be initially changed, and software metrics regarding the internal structure of the system. The model is generically defined in terms of modules, which are separate units of software. In this work, we propose an implementation of K3B for object-oriented programs.

We evaluated our implementation of K3B in two scenarios: an artificial one, by comparing the theoretical results of K3B with data from simulation, and in a real scenario, by comparing the theoretical results of K3B with historical data of open source projects. For this purpose, we developed a tool to simulate change propagation in Java programs and to count the resulting number of change steps. In this evaluation, we used data from 37 releases of 11 Java programs. The results have shown a strong correlation between the number of change steps given by K3B and those observed in the simulation. The data of K3B were also compared with the historical data of change propagation from 10 open-source Java projects (Geipel and Schweitzer 2012). We have also found a positive correlation about the results of K3B and the probabilities of change propagation in real scenarios.

The main contributions of our work are the following:

– the definition of a stochastic model, named K3B, for predicting change propagation in software modules (Section 2);
– an implementation of K3B for object-oriented programs (Section 3);
– the empirical evaluation of K3B by means of simulation (Section 4.1), and by comparing the results produced by K3B with historical data (Section 4.2).

2 The K3B Model

The modification of a program can be viewed as a process described as follows. During a modification process, a module can be *consistent* or *inconsistent*. If an interface element of a module is changed, all modules that use it will become *inconsistent* at the moment the modifications performed on the module are concluded. Since more than one interface element of a module can be modified, each module changing process may cause several sets of modules to be considered *inconsistent*. A module that is declared *inconsistent* needs to be checked for its conformance to the proper use of the resources provided by modules whose interfaces have been modified. It will return to a *consistent* state when it has been properly changed so that all of its imported resources are used accordingly to their exporting interfaces. On the other hand, if the modification process changes its public interface, all modules that depend on it will become *inconsistent*, and the propagation of changes continues.

Before the modification process, all modules of the program are supposed *consistent*. The process starts when a set of modules are taken to be changed. In this situation, these modules become *inconsistent*. The changes of these modules may demand changes in other ones successively. During this modification process, the changes of some modules may be started, and the changes of other modules may be declared as finished, iteratively. When a module of the system become *consistent* or *inconsistent*, an event is generated. We call this event as a *change step*. A *change step*, then, represents the situation when a change has to be performed or finished in a module. Indeed, during a modification process, the number of modules to be changed can increase and decreased, iteratively, and the notion of *change step* captures this behavior of the modification process. The process is said accomplished when all modules are *consistent* again, i.e., when the number of *inconsistent* modules is zero.

The model proposed in this work aims to estimate the number of change steps for a modification process of a system with n classes, when i modules will be initially changed, until the modification process ends, i.e., until all the changes resulting from the initial ones are completely finished and all the modules are *consistent*. The effort needed to update a module, although important for estimating the maintenance cost, is not considered in the K3B model, because our purpose is to predict the number of change steps the modification task will take to be completed.

2.1 Scenario Examples

Consider that a user of an object-oriented system reported an issue to the project team. After reading the issue report, one of the project developers concluded that there is an error in the system. Initially, he or she identifies that the error is related to three classes which need to be changed. The developer takes the three classes to be changed, starting the changing process. These three classes, then, go to the *inconsistent* state and, hence, three change steps are performed. The developer finishes the maintenance in one of the classes and; so, this class turns into *consistent* and one more change step is performed. Let's call this class `A`. In this situation, there are still two classes in the *inconsistent* state. However, due to the change in `A`, the developer identifies that other two classes of the system need to be changed too. Then, these classes turn into *inconsistent* and two more change steps are counted. At this point, there are four classes in the *inconsistent* state. The developer finishes the maintenance in one of the four classes, which turns into *consistent*. Let's call this class B. Therefore, one more change step is counted and there are now three classes in the *inconsistent* state. However, the developer notes that `A` needs to be changed again as a consequence of the change performed in B. The class A is taken to be changed, turning into *inconsistent*, and one more change step is counted. In this situation, there are four classes in the *inconsistent* state. The developer finishes the maintenance on them, noting that there is no more need of change in the system regarding the initial changes, i.e., changing those four classes will not impact other classes in the system. In this situation, the four classes turn into *consistent* and four more change steps are counted. At this point, all the classes of the system are *consistent* and the changing process is over. In the total, five classes were changed and twelve change steps were carried out.

This example describes a maintenance scenario started due to an error in the system. A similar scenario can occur with other kinds of maintenance. For instance, consider that a new feature will be included in the system. After designing the new feature, the software engineer concludes that to include the new feature in the system, three classes of the current system need to be changed. The process of changing the current system starts by taking the three classes to be changed. A similar change process, then, may be followed in the same way of the first scenario.

The proposed model aims to predict how many change steps the whole change process will take. Of course, other problems are involved in a change process. For instance, the model is not intended to identify the classes that need to be changed, though this is an important problem in this context too. Nevertheless, the model aims to aid to estimate the amount of task to be done in order to accomplish a whole change process, as described in these examples.

2.2 The Model Definition

To define the change propagation model, we represent a software system as a directed graph in which the nodes represent the modules, and the edges represent the *connections* between the modules. A *connection* from a module A to a module B represents a dependence relationship in such a way that a change in B may require a change in A.

Consider a strongly connected software system with n modules and let $S = \{0, 1, 2, ..., n\}$ denote the set of possible states of a change process in the system. In the model, a **state** corresponds to the number of modules which are *inconsistent* in a given moment. A change process ends when the number of modules that are *inconsistent* is equal to 0, i.e., when the change process is in state 0. This is the *equilibrium state* of the process.

In a real modification process scenario, many modules could simultaneously have their status switched (to *inconsistent* or *inconsistent*). That is, a team may start changing more than one module a time, as well as, it may conclude the changes in more than one module a time. The main problem with such assumption is the unit of time to be considered. If we consider a small unit of time, this assumption becomes unrealistic. For this reason, we do not define the unit of time in our model. The unit of time can be a second, an hour or even a month. The model is designed to count the total number of changes that will be necessary to accomplish the whole maintenance task. Therefore, the time in which each change is performed does not matter to the final result. Besides, considering parallel changes will introduce unnecessary complexity to the model.

Thus, for simplicity, we assume that changes are carried out singly and sequentially, in order to compute the number of change steps. Therefore, when there are i modules *inconsistent*, $i \epsilon \{1, 2, 3, ..., n - 1\}$, it is only possible to go to a state in which there are either $i + 1$ or $i - 1$ *inconsistent* modules. This problem can be modeled as a Markov Chain. In a Markov Chain, when i is the present state, the probability of going to future states depends only on the present state, regardless of the previous states. Figure 1 shows the Random Walk (Petrov and Mordecki. 2003) for this problem. In state i, the probability that the system goes to state $i + 1$ is q_i, and the probability that the system goes to state $i - 1$ is $1 - q_i$. The Random Walk is represented by a probability matrix $P = (p_{ij})$. Each position ij in the matrix represents the probability that the system goes from state i to state j. These probabilities are defined as follows.

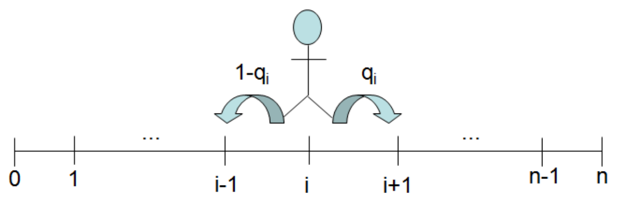


Fig. 1 The random walk of the change propagation process: i is the number of modules in the *inconsistent* state in a given moment.

- State 0 is the absorbing state, i.e., the *equilibrium state*. In this state, the change process ends. The probability of leaving this state is 0, and the probability of staying in it is 1. Hence, row 0 of the matrix is filled as follows.

$$p_{0j} = \begin{cases} 1 \text{ if } j = 0 \\ 0 \text{ if } j \neq 0 \end{cases}$$

- In state n, all modules are *inconsistent*. In this situation, the only possibility is to go to state $n-1$. Thus, row n of the matrix is filled as follows.

$$p_{nj} = \begin{cases} 1 \text{ if } j = n-1 \\ 0 \text{ if } j \neq n-1 \end{cases}$$

/;

- In the intermediate states i, $i \in \{1, 2, 3, ..., n-1\}$, it is only possible to go to state $i+1$ or $i-1$. The probability of going to state $i+1$ is q_i, and the probability of going to state $i-1$ is $1-q_i$. In such states, the matrix is filled as follows.

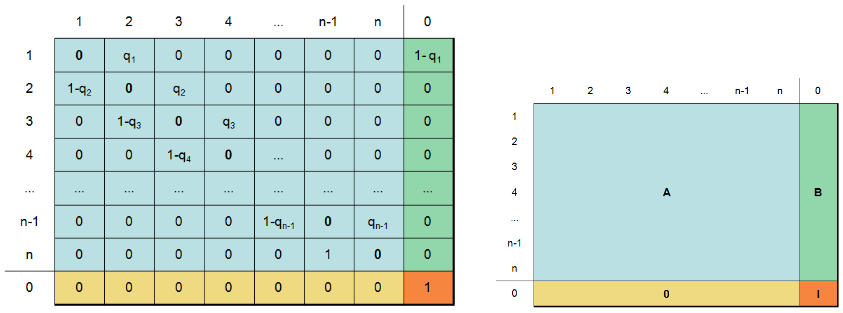
$$p_{ij} = \begin{cases} 0 & \text{if } j \neq i+1, j \neq i-1 \\ q_i & \text{if } j = i+1 \\ 1-q_i & \text{if } j = i-1 \end{cases}$$

We use the term "contamination rate" to designate the rate with which a change in a module will demand changes in other modules. We refer the parameter which indicates this rate as α. This parameter should be a factor in such a way the higher its value, the higher the "contamination rate".

We use the term "decontamination rate" to designate the rate with which a change in a module will be carried out without demanding changes in other modules. We refer the parameter which indicates this rate as β. This parameter should be a factor in such a way the higher its value, the higher the "decontamination rate".

In state i, the probability of going to state $i+1$ is proportional to:

- the "contamination rate", α.
- the number of modules which are *consistent*, $n-i$, because the higher the number of *consistent* modules, the higher the chance of at least one of them turns *inconsistent*.
- the number of modules which are *inconsistent*, i, because the higher the number of *inconsistent* modules, the higher the chance of other module in the system turns *inconsistent* too.



	1	2	3	4	...	n-1	n	0
1	**0**	q_1	0	0	0	0	0	$1-q_1$
2	$1-q_2$	**0**	q_2	0	0	0	0	0
3	0	$1-q_3$	**0**	q_3	0	0	0	0
4	0	0	$1-q_4$	**0**	...	0	0	0
...	...	...	...	...	...	...	...	...
n-1	0	0	0	0	$1-q_{n-1}$	**0**	q_{n-1}	0
n	0	0	0	0	0	1	**0**	0
0	0	0	0	0	0	0	0	1

1 2 3 4 ... n-1 n 0
1 2 3 4 ... n-1 n 0
A
B
0
I

Fig. 2 The probability matrix.

- the *connectivity* of the system. In this context, *connectivity* is referred to as the rate of existing connections among modules in the software system. The higher the connectivity of the system, the higher the chance of a change in a module affecting other modules. We refer the parameter which represents the connectivity among modules within a system as ϕ. For instance, in a system with $\phi = 0.5$, there are 50% of the possible connections between modules, whereas a system with $\phi = 1$ is a totally connected system.

Thus, from a state i, the rate of changing to state $i + 1$ is proportional to $\phi\ \alpha\ (n - i)i$.

In state i, the rate of going to state $i - 1$ is proportional to:

- the "decontamination rate", β.
- the number of *inconsistent* modules, because the higher the number of *inconsistent* modules, the higher the probability that one of them turns *consistent*;

Thus, from a state i, the rate of changing to state $i-1$ is proportional to $\beta\ i$.

As the sum of $p_{i,i+1}$ and $p_{i,i-1}$, $i \ \epsilon\ \{1, 2, 3, ..., n-1\}$, must be equal to 1, Equations 1 and 2 are, then, defined.

$$p_{i,i+1} = \frac{\alpha\phi(n-i)i}{\alpha\phi(n-i)i + \beta i} \tag{1}$$

$$p_{i,i-1} = \frac{\beta i}{\alpha\phi(n-i)i + \beta i} \tag{2}$$

The purpose of this work it to define a method to compute the number of steps a process will take to go from state i to the absorbing state, i.e., the number of steps it will take until the change process reaches the equilibrium.

Theorem 1, described by Grinstead and Snell. (1991), defines the estimated number of steps a Markov Chain will take to reach the absorbing state when the chain is in state i. In order to apply this theorem, a $(n + 1) \times (n + 1)$ matrix of probabilities, where n is the number of modules in the system, must be constructed. In this matrix, the absorbing state corresponds to the last line and to the last column of the matrix. Hence, the matrix of probabilities, $P = (p_{ij})$, for the problem of change propagation in software is filled as shown in Figure 2. The resulting matrix has four regions: A is a $n \times n$ matrix; *0* is a row vector with all elements equal to zero; I is the identity matrix; and B is a column vector whose first element can be nonzero, but the others can not. The theorem is defined as follows.

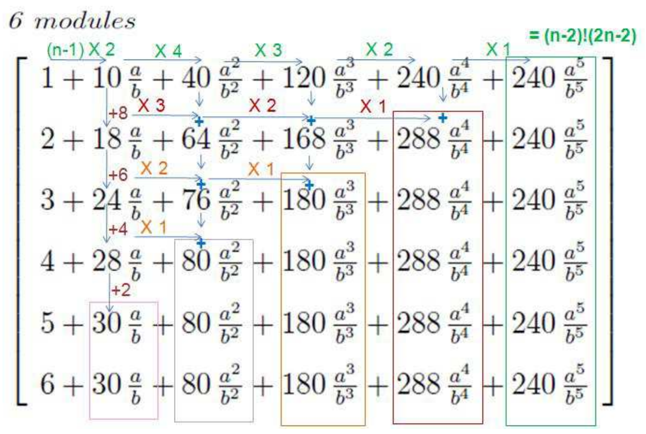

6 modules
(n-1) X 2
X 4
X 3
X 2
X 1
= (n-2)!(2n-2)
+8
+6
+4
+2
$1 + 10\frac{a}{b} + 40\frac{a^2}{b^2} + 120\frac{a^3}{b^3} + 240\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$
$2 + 18\frac{a}{b} + 64\frac{a^2}{b^2} + 168\frac{a^3}{b^3} + 288\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$
$3 + 24\frac{a}{b} + 76\frac{a^2}{b^2} + 180\frac{a^3}{b^3} + 288\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$
$4 + 28\frac{a}{b} + 80\frac{a^2}{b^2} + 180\frac{a^3}{b^3} + 288\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$
$5 + 30\frac{a}{b} + 80\frac{a^2}{b^2} + 180\frac{a^3}{b^3} + 288\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$
$6 + 30\frac{a}{b} + 80\frac{a^2}{b^2} + 180\frac{a^3}{b^3} + 288\frac{a^4}{b^4} + 240\frac{a^5}{b^5}$

Fig. 3 The pattern of the expressions.

Theorem 1 *Let i be a state of the chain. Let $E(t_i)$ be the number of estimated steps before the chain be absorbed, when the chain starts in state i. Let E be the column vector whose i-th entry is $E(t_i)$. Thus, $E = Nc$, where $N = (I - A)^{-1}$, and c is a column vector whose entries are equal to 1.*

Applying the theorem, we have Equation 3 as result. The i-th position of the vector E, $E(t_i)$, corresponds to the mean number of steps which the chain will take until it reaches the absorbing state, when the chain starts in state i. In the context of change propagation, $E(t_i)$ represents the number of *change steps* that a change process will take to reach the *equilibrium state* when i modules of a software system are initially changed.

$$\begin{pmatrix} E(t_1) \\ E(t_2) \\ E(t_3) \\ \vdots \\ E(t_n) \end{pmatrix} = (\mathbf{I} - \mathbf{A})^{-\mathbf{1}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{3}$$

$$E(n,1) = 1 + 2(n-1)(n-2)!\,\frac{\alpha^{n-1}\phi^{n-1}}{\beta^{n-1}} + \sum_{k=1}^{k=n-2} \frac{2(n-1)(n-2)!}{k!}\,\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}} \tag{4}$$

$$E(n,i) = 1 + E(n,i-1) + \sum_{k=i-1}^{k=n-2} 2(n-i)\,\frac{(n-i-1)!}{(k-i+1)!}\,\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}} \tag{5}$$

Using Matlab, we calculated the vector E for all $n \,\epsilon\, \{4, 5, 6, ..., 11, 20, 25\}$. The results of this computation are shown Appendix A. From the analysis of these results, we observed a pattern formation in the polynomials. Figure 3 shows the polynomials for software system with six modules, $n = 6$, and indicates the pattern of the expressions. This pattern was also observed in the polynomials for all $n \epsilon \{4, 5, 6, ..., 11, 20, 25\}$. From this observation, we analytically defined a generic formula which represents the pattern formation of those polynomials, that are the solution for Equation 3. The generic formula is given by Equations 4 and 5. Those equations are, then, the solution of the proposed model[1]. For simplicity, we call them *the K3B model*.

The K3B model show that the number of change steps is given as a function of the relation $\frac{\phi\alpha}{\beta}$. When $\phi = 0$, there are no connections between modules within the software system. In this case, the resultant number of change steps will be i, which is the number of modules that will be initially changed. This is the expected result, because, in that situation, there will be no change propagation. The same occurs when $\alpha = 0$, which means that the "contamination rate" is zero.

In a system with connections between its modules, i.e., when $\phi > 0$, the best case occurs when α and ϕ have a very low value, and β has a high value. In this case, the number of change steps tends to i.

The worst case occurs when α and ϕ have high values, and β has a value near to 0. In this case, the number of change steps is explosive, which means that the problem of changing a software system may be intractable.

The K3B model are given in terms of factorial. Logarithmic transformation of the formula is a way to overcome overflow problems in the implementation of K3B. The factorial function can be generalized to real numbers by the Gamma function (represented by Γ) as shown in Equation 6. The natural logarithmic of $n!$ corresponds to the natural logarithmic of $\Gamma(n+1)$, referenced as *lngamma*, which can be directly calculated. There are some available implementations of *lngamma* for different programming languages. An implementation of *lngamma* in Java is provided by the Weka project[2].

$$n! = \Gamma(n+1) = \int_0^\infty t^n e^{-t} dt \tag{6}$$

3 An Implementation of K3B for Object-Oriented Programs

In this section, we present an implementation of K3B for object-oriented programs[3]. Object-oriented systems are made up of classes which relate one to another. We shall refer to classes as modules.

K3B has as entry five parameters: ϕ, the connectivity of the system; α, the "contamination rate" in the system; β, the "decontamination rate" of the system; n, the total number of classes; and i, the number of classes that will be initially modified. The definition of K3B does not specify which metrics should be represented by parameters ϕ, α and β, however, we elected three well-known software metrics to represent those parameters.

[1] The formula was defined based on the resultant polynomials for $n \epsilon \{4, 5, 6, ..., 11, 20, 25\}$. Analyzing data for values of $n > 25$ manually was not viable.

[2] http://www.cs.waikato.ac.nz/ml/weka/

[3] Our implementation of K3B was introduced in Connecta Project and it is available in http://homepages.dcc.ufmg.br/˜ kecia/connecta.htm.

The parameter α must be a factor in such a way as higher the factor, as higher the "contamination rate", i.e, the change propagation. The parameter β must lead to the opposite effect. To choose the metrics for α and β, we considered modularity as one of the most relevant factors of software quality and software maintenance (Meyer 1997). Moreover, coupling among modules and module cohesion are considered the main forces that govern modularity (Myers 1975; Pressman 2009). Thus, based on those assumption, we consider that coupling metrics play the role of α, and cohesion metrics of β. Nevertheless, K3B does not exclude the use of other metrics for α and β, provided the chosen metric for α corresponds to a factor which may contribute to favor change propagation, and the chosen metric for β corresponds to a factor which may contribute to prevent change propagation.

3.1 A metric for ϕ

The ϕ parameter represents the *connectivity* of the system. In the K3B model, the software system is represented as a directed graph, i.e, a network. The *connectivity* refers to the density of the network, that is, the percentage of existing connections on it. Abreu and Carapuça (1994) defined a software metric that aims to measure such property in object-oriented software systems. This metric, called COF (coupling factor), is given by $c/(n^2 - n)$, where c is the number of connections among the classes, and n is the number of classes in the system. In our implementation of K3B, we have calculated ϕ according to the metric COF. To generate the graph which represents the object-oriented program, we considered the following types of connections between classes: inheritance, use of method and use of fields. However, the definition of the model does not limit the types of connections to be considered in the implementation of the model.

3.2 A metric for β

In our implementation of K3B, we used the metric Cohesion by Responsibility (COR) (Ferreira 2011) for β. This metric is given by $1/C$, where C is the number of disjoint sets of methods within the class. Each of these sets consists of similar methods. Two methods are similar when they use a common field or a common method of the class. The similarity relation is transitive in the sense that if a method a is similar to a method b, and b is similar to a method c, then a is also similar to c. Each set defines a class responsibility. For example, when there are only two sets in a class, COR is 0.5. This indicates that this class has two responsibilities. If there is only one set in a class, COR will result in 1, which indicates a high cohesion. Ferreira et al (2011) carried out an experimental evaluation of four cohesion metrics: LCOM (*lack of cohesion in methods*) (Chidamber and Kemerer 1994), LCOM4 (Hitz and Montazeri 1995), TCC (*tight class cohesion*) (Bieman and Kang 1995) and COR. Findings of that work showed that COR may be used as a good indicator of class cohesion and it is useful in identifying classes with design deviance. Based on these findings, we have chosen to use COR in our implementation of K3B.

Cohesion metrics are usually defined to evaluate a single class. As β is a value taken for the system as whole, we computed the mean cohesion in the system as β. We applied this approach in our implementation of K3B.

3.3 A metric for α

Coupling is the level of interdependence among modules of a system. If a class depends on another, there is a connection between them. The nature of this connection may define the coupling level between the classes. For instance, the use of public fields may cause strong coupling between classes.

We consider four main types of dependence among classes. To each of them, we associated a weight from 0 to 1, because the value is supposed to represent α, that is a rate. The weights of this scale were set based on the idea that the higher the coupling, the higher should its weight be. The highest weight we used were 0.2. We carried out previous tests with K3B formulae in order to calibrate those weights. The value 0.2 was selected to be used in our implementation as the highest value for α because bigger values had led to huge values of K3B, what we assumed to be unrealistic[4]. The types of connections and its weights are described as follows.

- Use of field: this type of coupling occurs when a class uses a field of another, what might lead to strong coupling between the classes. For this reason, the weight associated to this type of coupling is the highest in this scale: 0.2.
- Inheritance: it occurs when a class extends another. In languages such as Java, when a class implements an interface, it could also be considered an instance of this type of coupling. This kind of relationship leads to a strong dependence among classes, but not as high as the *connection by the use of field.* The weight associated to this type of connection should be less than that associated to *connection by the use of field.* We used 0.1 as the weight for *connection by inheritance.*
- Reference: when a class uses a method of another and passes a reference to some object to this method as parameter, there is a relevant degree of dependence among these classes, because changing the object inside the called method would impact on the caller. We used the weight 0.1 for this kind of coupling.
- Use of method: it occurs when a class uses a method of another and only passes non-object data to this method as parameter. This is the lowest level of coupling among classes, but it still represents some level of dependence. We used 0.05 as the weight for this level of coupling.

A class may use another one in many ways, resulting in different types of connections. In this case, we considered the connection with the highest weight. As α is a single value, which represents a property of the system as whole, we used the mean value of the weights of all the connections as α.

[4] Geipel and Schweitzer (2012) have found that the probability that two interconnected classes have been modified together is at least once is 0.33, with a standard deviation of 0.12. The weights we used in our study are near to those.

3.4 Computing K3B

To implement K3B, we represent an object-oriented program as a weighted directed graph, where: a node represents a class with its corresponding metric COR; an edge corresponds to a connection between two classes, and it has a weight that represents the type of the connection. We compute COF as ϕ, the mean weights of the edges as α, and the mean value of COR of the nodes as β. The evaluation of K3B performed in this work is based on the implementation of K3B described in this section.

4 Evaluation of K3B

In this section we evaluate the proposed model. We compare the results generated by our implementation of K3B with: (1) data from simulation of change propagation in Java programs; and (2) data of a previous study that empirically investigated the change propagation in real scenarios of modification process of Java projects.

4.1 K3B $\times$ Simulation

We developed a tool (Ferreira 2011) which simulates a given type of change the classes of a program, and counts the number of change steps resulting from each initial change. The tool analyzes Java software given its bytecode, the type of change which will be simulated, and the number of classes which will suffer changes initially. The tool simulates all the possible changes in the program with the given pattern and, for each of these instances of change, it counts the number of change steps and returns the average number of change steps.

The evaluation of K3B was performed according to the following steps. Initially, for a given program, K3B estimates the number of change steps for $1 \leq i \leq n$, where i is the number of classes which will be initially changed, and n is the total number of classes that compose the program. For each value of i, we performed the simulation of a given type of change. We, then, compared the value computed by K3B with the resultant value of the simulation. The hypothesis investigated in this evaluation is that K3B estimates the mean number of change steps in object-oriented software systems. In order to verify this hypothesis, we investigated if there is correlation between the values generated by K3B and the results of the simulation. We used the Pearson correlation to perform such analysis. In this experiment, we used 37 versions of 11 open-source software systems.

4.1.1 The Evaluated Type of Change

The aim of this evaluation is to verify the accuracy of K3B in estimating change propagation. A change in a module can "contaminate" the other ones to which it is connected. This process dynamics is applicable to any modification, regardless of the type of the modification performed in the module interfaces. Hence, for the purpose of simulation, it is necessary just a situation which represents this

```
Require: $i \geq 1$ {the number of classes which will be changed initially}
Ensure: $meanStepChange \geq 0$ {the mean number of steps}
  $visited$ = empty {A set of classes.}
  $P$ = empty {A set of methods and its resulting number of change steps}
  for all class C do
    for all method M defined in C do
      visited.insert(C);
      result = countStepsMethod (M, visited);
      P.insert(M, result);
    end for
  end for
  for all combination Comb of i methods from distinct classes do
    for all method M of Comb do
      sumStepsComb += P.getSteps(M);
    end for
    totComb++;
    sumSteps += sumStepsComb;
    sumStepsComb = 0;
  end for
  meanStepChange = sumSteps / totComb;
  return meanStepChange
```

Fig. 4 Simulation of Change Propagation - *meanStepChange*

process, i.e., it is enough to perform a change in a module and see how it ripples successfully to other ones.

Changing the parameter list of a given public method implies that the contract (Meyer 1997) of the class will change necessarily. In particular, when a parameter is added to the parameter list of a method, there are two main ways to generate the new method's argument in the caller methods: the own caller method is able to provide the new data, or it may demand this new data will be passed to it as a parameter. Hence, a change in a parameter list of a method may be propagated to other ones.

Adding private fields may demand accessor methods. In the case of adding public fields, accessor methods are not needed. In both cases, the classes which use the changed class also may have to be changed in order to provide data for the new fields. In these situations, there are two main possibilities: the own class is able to provide the data, or it may demand the data be generated by other classes and, then, sent to it as a parameter. The effect of this type of change and the effect of changing parameter list are, hence, similar.

Fowler (1999) describes a set of changes that can be performed in a object-oriented software system, that is known as *code refactoring*. For instance, *add parameter*, *parametrize method*, *collapse hierarchy* and *extract class* may lead to change in the contract of the affected class. The propagation effects of those types of change are similar to *change of parameter list*. Therefore, for the purpose of simulating the effect of a change propagating throughout the system, a single type of change is sufficient. We have chosen the *change of parameter list* to simulate change propagation. Although this is not the only possible change in object-oriented software, it is good enough to represent the change propagation effect.

```
Require: $M$ {the method which will be changed initially}
Require: $visited$ {a set of classes}
Ensure: $p \geq 0$ {the total number of change steps}
  p = 0;
  for all class C which uses M do
    if C is not the class of M then
      if C contains a method X which uses M then
        p++;
        if C is not in visited then
          visited.insert(C);
          for all method X in C which uses M do
            p+= countSteps(X, visited);
          end for
        end if
      end if
    else
      for all method X in C which uses M do
        p += countSteps(X,visited);
      end for
    end if
  end for
  return p
```

Fig. 5 Simulation of Change Propagation - *countStepsMethod*

4.1.2 The Algorithm of the Simulation of Change Propagation

The simulation of the change propagation is based on the call flow among methods of a program. When a method $m2$ calls a method $m1$, and $m1$ has its parameter list changed, it is possible that $m2$ will have its parameter list also changed. This might occur because there are two main ways for $m2$ produce the new parameters to $m1$: $m2$ will produce the new parameter itself, or $m2$ also will need to receive the new parameter from some other object. As in the simulation it is not possible to define how $m2$ can produce the new parameter itself, we consider the worst case in the simulation: the parameter list of $m2$ also will be changed. Therefore, the propagation of a change in $m1$ is counted until a method which is not called by any other method is reached. The algorithms of Figures 4 and 5 describe the computation of the number of change steps when the parameter list of a method is changed.

The algorithm simulates all the possibilities of change propagation in the software system. To compute the change propagation for $i = 1$, i.e, when one class is changed, the simulation of change propagation is carried out for each method of the classes. At the end of the computation, the mean value of the partial results is calculated. This result represents the mean number of change steps when one class is changed. To simulate the effect of change propagation for $i > 1$, the algorithm takes i classes from the software program a time, making a combination of i methods from them. The result of each partial simulation is the sum of the change impact of the methods. After considering all the combination of i classes, the simulation ends. The final result of the simulation is the average of the partial results.

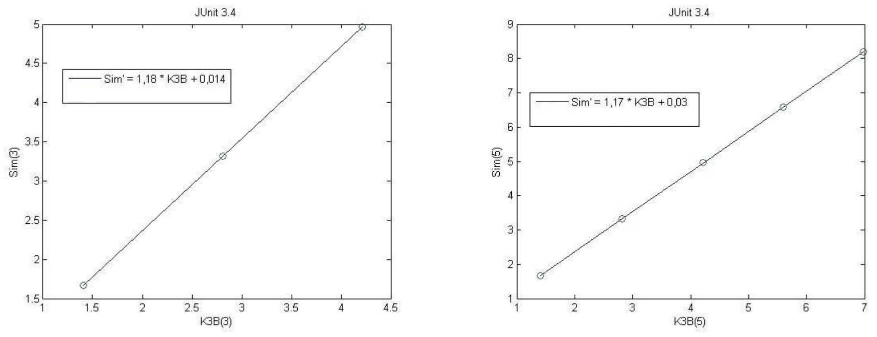


Fig. 6 Correlation between K3B and the data from simulation, for $i = 3$ and $i = 5$ - JUnit 3.4

4.1.3 Method of the Evaluation

To evaluate K3B, we verified if there is correlation between the values produced by K3B and the resultant values of the simulation. The hypothesis investigated is that K3B estimates the real value, hence, there is a positive correlation between K3B values and the data from simulation. We performed this analysis for the 37 software versions used in this experiment. For each of these programs, we computed the K3B value for different values of i (the initial number of modules in *inconsistent* state), and we collected the corresponding values in the simulation. Each of these observations resulted in a regression line $Simulated = m * K3B + b$.

If there is a positive correlation between K3B and the values given by the simulation, and if the values of m and b are slightly different, or invariant, among the resultant regression lines for distinct software systems, it is possible to conclude that, regardless the software system, K3B estimates the real number of change steps properly. Moreover, the simulation value can be calculated by multiplying K3B values by m and, then, adding the result with b. On the other hand, if there is a significant difference among the values of m and among the values of b in the resultant regression lines, it means that, though K3B can predict the real value, the regression line is sensitive to the evaluated software. In this situation, it would not be possible to define general values for m and b to obtain the real value using the K3B formulae.

Therefore, besides the verification of correlation between K3B and the values given by the simulation, we analyzed the behavior of the values of m and b among the regression lines. The hypothesis investigated in this analysis is that the value of K3B multiplied by a known constant will result in the real value, regardless the software system.

4.1.4 Results

In this section, we present and discuss the results of the evaluation of K3B. For simplicity, we use the term $K3B(i)$ to denote the value given by K3B for a program when i of its classes are initially changed. In the same way, we use the term $Sim(i)$ to denote the value given by the change simulation algorithm considering that i classes of the program are initially changed.

Table 1 Comparison between K3B and data from the simulation – $1 \leq i \leq 3$

Program	Version	α	β	ϕ	Classes	Methods	K3B(1)	K3B(2)	K3B(3)	Sim(1)	Sim(2)	Sim(3)	r	m	b
Hibernate	3.0	0.083	0.584	0.003	956	10358	2.33	4.66	6.99	2.67	5.34	8.01	0.99999997	1.15	0
	3.1	0.085	0.584	0.003	1118	13683	2.38	4.76	7.14	2.73	5.47	8.01	0.99978741	1.11	0.12
Jasper Reports	0.4.0	0.095	0.720	0.009	242	1790	1.75	3.49	5.23	2.73	5.47	8.20	0.99999998	1.57	-0.02
	1.0.0	0.083	0.639	0.004	574	6094	1.88	3.77	5.65	2.79	5.58	8.38	1.00000000	1.48	0.00
	2.0.0	0.081	0.617	0.002	1104	13196	1.97	3.94	5.90	2.82	5.65	8.79	0.99953185	1.52	-0.23
Java Groups	1.0	0.109	0.660	0.007	415	3331	3.01	6.01	8.99	3.73	7.44	11.14	0.99999996	1.24	0.00
	2.1.1	0.105	0.699	0.004	696	8638	2.46	4.92	7.37	4.51	9.02	12.37	0.99633703	1.60	0.76
JML	10a1	0.085	0.535	0.017	171	1167	2.61	5.20	7.77	2.33	4.66	6.98	0.99999954	0.90	-0.02
	10a2	0.079	0.580	0.015	186	1652	2.18	4.35	6.51	2.01	4.01	6.02	0.99999818	0.93	-0.01
	10b1	0.081	0.582	0.015	203	1808	2.42	4.82	7.21	2.13	4.27	6.41	0.99999960	0.89	-0.03
	10b3	0.081	0.576	0.014	218	1956	2.50	4.99	7.47	2.16	4.33	6.49	0.99999949	0.87	-0.02
	10b4	0.083	0.592	0.012	270	2319	2.74	5.48	8.20	2.59	5.18	7.77	0.99999889	0.95	-0.01
JSCH	0.1.14	0.111	0.747	0.032	80	671	2.17	4.32	6.44	2.30	4.60	6.89	0.99999908	1.08	-0.04
	0.1.20	0.108	0.668	0.028	83	740	2.17	4.31	6.43	2.26	4.51	6.75	0.99999839	1.05	-0.03
	0.1.26	0.104	0.681	0.024	94	823	2.03	4.05	6.04	2.15	4.29	6.43	0.99999829	1.07	-0.02
	0.1.34	0.105	0.690	0.023	109	941	2.17	4.32	6.45	2.26	4.53	6.80	0.99999901	1.06	-0.05
	0.1.42	0.103	0.659	0.020	117	1012	2.11	4.21	6.29	2.25	4.50	6.75	0.99999734	1.08	-0.02
JUnit	3.4	0.078	0.815	0.023	78	315	1.41	2.81	4.21	1.67	3.32	4.96	0.99999968	1.18	0.01
	3.8	0.078	0.803	0.018	101	595	1.41	2.81	4.21	2.35	4.70	7.04	0.99999996	1.67	-0.01
	4.0	0.073	0.763	0.020	92	924	1.42	2.82	4.23	2.27	4.53	6.80	0.99999944	1.61	-0.02
	4.5	0.075	0.742	0.010	188	1516	1.47	2.94	4.40	2.58	5.15	7.73	0.99999946	1.76	-0.01
	4.8.1	0.075	0.742	0.008	230	1694	1.46	2.92	4.38	2.68	5.35	8.02	0.99999968	1.83	0.01

Table 2 Comparison between K3B and data from the simulation – $1 \leq i \leq 3$

Program	**Version**	α	β	ϕ	**Classes**	**Methods**	**K3B(1)**	**K3B(2)**	**K3B(3)**	**Sim(1)**	**Sim(2)**	**Sim(3)**	**r**	**m**	**b**
KolMafia	0.2	0.091	0.785	0.056	39	239	1.65	3.27	4.88	2.83	5.69	8.58	0.99997862	1.78	-0.12
	1.0	0.093	0.760	0.025	143	634	2.47	4.93	7.37	5.53	11.02	16.46	0.99999996	2.23	0.01
Logsim	2.0.0	0.092	0.632	0.004	908	6738	3.17	6.34	9.51	3.86	7.72	11.58	0.99999997	1.22	0.00
	2.1.0	0.092	0.630	0.004	993	7508	3.32	6.64	9.95	3.90	7.79	11.68	0.99999980	1.17	0.00
Movie	1.6beta	0.080	0.767	0.037	64	364	1.64	3.26	4.87	2.93	5.75	8.46	0.99996704	1.71	0.13
Manager	1.7	0.080	0.832	0.032	73	418	1.56	3.12	4.66	2.88	5.64	8.30	0.99996424	1.75	0.16
	2.0	0.088	0.687	0.004	517	8097	1.66	3.31	4.96	6.53	13.04	19.53	0.99999959	3.94	0.00
	2.8	0.095	0.830	0.007	458	3495	2.23	4.46	6.69	8.43	16.77	25.02	0.99999604	3.72	0.14
	2.9.13	0.097	0.834	0.005	608	4655	2.27	4.54	6.81	7.26	14.48	21.64	0.99999818	3.17	0.07
Phex	0.6	0.098	0.648	0.007	393	2232	2.49	4.98	7.45	8.71	17.41	26.10	0.99999996	3.50	-0.02
	2.8.0	0.087	0.705	0.003	1205	9872	2.42	4.83	7.24	4.74	9.48	14.21	1.00000000	1.97	-0.01
	3.4.2	0.087	0.703	0.003	1352	14078	2.75	5.50	8.25	4.16	8.33	12.49	1.00000000	1.52	-0.01
Squirrel	1.0	0.072	0.785	0.004	424	1589	1.37	2.73	4.10	4.06	8.11	12.16	1.00000000	2.97	-0.01
	2.0	0.075	0.746	0.003	729	5103	1.46	2.92	4.37	3.33	6.67	10.00	0.99999997	2.29	-0.01
	2.6	0.079	0.799	0.002	940	6335	1.50	3.00	4.50	3.23	6.46	9.69	1.00000000	2.16	0.00

Table 3 Comparison between K3B and data from the simulation – $i = 4$ and $i = 5$

Program	**Version**	**K3B(1)**	**K3B(2)**	**K3B(3)**	**K3B(4)**	**Sim(1)**	**Sim(2)**	**Sim(3)**	**Sim(4)**	**r**	**m**	**b**
JML	10a1	2.61	5.20	7.77	10.33	2.33	4.66	6.98	9.29	0.999998535	0.90	-0.03
JSCH	0.1.14	2.17	4.32	6.44	8.54	2.30	4.60	6.89	9.17	0.999997453	1.08	-0.05
Kolmafia	1.0	2.47	4.93	7.37	9.79	5.53	11.02	16.46	21.85	0.999999882	2.23	0.02
Squirrel	1.0	1.37	2.73	4.10	5.46	4.06	8.11	12.16	16.20	0.999999996	2.97	-0.004

Program	**Version**	**K3B(1)**	**K3B(2)**	**K3B(3)**	**K3B(4)**	**K3B(5)**	**Sim(1)**	**Sim(2)**	**Sim(3)**	**Sim(4)**	**Sim(5)**	**r**	**m**	**b**
JUnit	3.4	1.41	2.81	4.21	5.60	6.99	1.67	3.32	4.96	6.58	8.19	0.999997486	1.17	0.03
Kolmafia	0.2	1.65	3.27	4.88	6.46	8.02	2.83	5.69	8.58	11.49	14.41	0.999905934	1.82	-0.23
MovieManager	1.6beta	1.64	3.26	4.87	6.46	8.05	2.93	5.75	8.46	11.09	13.66	0.999905934	1.67	0.25

The evaluation was limited to $1 \leq i \leq 3$ because the simulation is a high cost process. The algorithm simulates all the possibilities of change propagation. Therefore, its complexity is exponential. Some simulations for $i = 3$ took more than 24 hours of computer processing time. In four programs, the execution of the simulation was performed for $i = 4$, and in other three programs, the execution of the simulation was performed for $i = 5$. Some of these simulations have consumed about three days. For this reason, performing the simulation for values of i higher than five is impracticable.

Tables 1 and 2 show the data from the analysis for $i = 3$. Columns α e β correspond to the mean class coupling and the mean class cohesion of the system, respectively. Column ϕ is the COF metric of the system. Columns $K3B(i)$ and $Sim(i)$ are the values of K3B and the values given by the simulation, respectively. Column r is the correlation coefficient between K3B and the corresponding value observed in the simulation. Columns m and b are the coefficients of the regression line $Simulated' = m * K3B + b$.

The results for $1 \leq i \leq 4$ and $1 \leq i \leq 5$ are reported in Table 3. From the gathered data, we observed that these results are very close to those for $1 \leq i \leq 3$. An example of this situation is illustrated in Figure 6, which shows the result of the program JUnit, version 3.4, for $i = 3$ and $i = 5$. Although these results are not conclusive, they suggest that those for $1 \leq i \leq 3$ can be considered as significant.

The analysis of the data reported in Tables 1, 2 and 3 shows that:

- In all cases, correlation r is nearly 1. The mean of the values of r is 0.99992, with standard deviation of 0.0005. This result indicates a strong correlation between K3B and the observed data in the simulation. Despite the small number of points used in the correlation analysis, this finding reveals that K3B indeed predicts the values given by the simulation.
- The values of the coefficients m are very close. The same occurs with the coefficient b. The mean values of these coefficients and their respective 95% confidence intervals are such as:
  - the mean value of m is 1.7, with standard deviation of 0.82. The 95% confidence interval of the mean is [1.46; 1.92]. This indicates that the value observed in a simulation is 1.92 times larger than the value given by K3B at most;
  - the mean value of b is 0.02, with standard deviation of 0.14. The 95% confidence interval of the mean is [-0.02; 0.06]. This indicates that, in general, the value of b is near to 0. Thus, the value given by the simulation can be obtained by adjusting the value of K3B with the known mean of m.
- Equation 7 is the mean regression line between the values of K3B and the observed data from the simulation.

$$Simulated = 1.7 * K3B \tag{7}$$

This finding suggests that the value given by K3B can indeed be used to predict the value given by the simulation. Moreover, as the value of m is quite small, the value given by K3B and the value given by the simulation have the same magnitude. Assuming that the simulation is a good expression of the real scenarios, the K3B model can be, hence, used to predict the number of change steps in software systems.

The K3B model is based on the probability of a change be propagated in the system. In our implementation of K3B, such probabilities were computed in terms of factors such as level of class coupling, cohesion and connectivity. The simulation of change propagation used in the evaluation of K3B does not directly consider cohesion and level of coupling. In the simulation, a change in a method is supposed to be always propagated to all the other methods which use the changed method. The simulation is, hence, more pessimist than K3B. The achieved results are consistent with this, because they show that the values given by the simulation are larger than those given by K3B, as expected.

Simulation is a high-cost process. In some cases, the simulation time is infeasible. For instance, the simulation of change of four classes ($i = 4$) from JSCH, which has only 80 classes and 671 methods, took 34 minutes. The simulation of change of four classes from Squirrel, which has only 424 classes and 1589 methods, took more than 24 hours. As the implementation of K3B is simple and it computes the result promptly, K3B is an efficient candidate to predict the number of change steps in software systems.

The systems considered in this study have the following mean parameters for K3B: $\phi = 0.009$, $\alpha = 0.085$, $\beta = 0.700$. Hence, in general, there is less than 1% of the connections among the classes in the system. The coupling among classes is near to 0.1, which is the corresponding weight of coupling by inheritance or by reference. The mean cohesion of the classes is 0.7, which corresponds to a high cohesion. These values suggest that the software systems considered in this study have good internal quality in general, regarding coupling, cohesion and level of connectivity. This might be the reason why the K3B values obtained in this study are low. However, a deeper investigation would be necessary to state that these systems actually have high internal quality. Even in software systems with such internal characteristics, K3B may be of help because not always the developer will be able to estimate the impact of the changes he or she has to accomplish. The importance of predicting change impact would be even greater in systems with low internal quality.

4.2 K3B × Historical Data

In this section, we compare the results of K3B with historical data that consider the dependencies between modules. To carry out this comparison, we considered the results of the empirical study of Geipel and Schweitzer (2012) that investigated the role of class dependence in change propagation. They have shown that dependencies among classes significantly raise the chance of change propagation.

We decided to compare our model with the results of their study for many reasons: the sample they used is large; they analyzed open-source Java projects, that is the target programming language of our tool; the way they represented the dependencies between classes is similar to ours; and they provided detailed data about the results in their paper so that we could carry out a comparison with our proposal.

Following, we describe the terminology used by Geipel and Schweitzer (2012) in their study:

- co-change: it is an event that “comprises all classes whose changes have been committed at exactly the same time by exactly the same author”. This event is considered by them as a change propagation occurrence. The change data consist of CVS logs.

– Dependence: they consider that `I` depends on `J` when: `I` extends or implements `J`; `I` uses `J` as a member or a variable; or `I` references or calls a method of `J`. They refer those types of dependencies as A, B, and C, respectively.
– P_D: the probability that two classes have changed together at least once, given that a dependence exists between them.
– P_Da: the probability that two classes have changed together at least once, given that a dependence of type A exists between them.
– P_Db: the probability that two classes have changed together at least once, given that a dependence of type B exists between them.
– P_Dc: the probability that two classes have changed together at least once, given that a dependence of type C exists between them.

4.2.1 Method of the Evaluation

The results of the study of Geipel and Schweitzer (2012) are reported in terms of probabilities, whereas K3B gives the absolute number of *change steps*. Although those results are reported in different units, we consider comparing them may put K3B in perspective, since the data of the study of Geipel and Schweitzer (2012) are from real scenarios of modification process. For the purpose of comparison, we considered the value $(K3B(i))/i$, that is a *relative* K3B, instead of $K3B(i)$, which is the absolute value of K3B. We made this adjustment because K3B depends on the size of the system and, then, comparing it with a probability, which is a relative number, would be misleading.

The data analyzed by Geipel and Schweitzer (2012) are from 35 projects and were collected in 2008. Their paper reports only a date for each project, and it does not report the release numbers. We faced some problems when recovering the versions of the projects: some of them are not available anymore, and, for most of them, it was not possible to identify the proper files to analyze based only in the date reported in the study of Geipel and Schweitzer (2012). Aiming to mitigate the threats to our analysis, we only considered the projects for which we could clearly identify the release close to the date reported by Geipel and Schweitzer (2012). This selection resulted in ten projects. We, then, gathered the K3B values of them.

To compare the results of K3B with the historical data, we analyzed the correlation among the probabilities of co-change obtained by Geipel and Schweitzer (2012) and the values of $K3B(1)$, $K3B(2)$, $K3B(n/2)/(n/2)$, and $K3B(n)/n$. We used Pearson and Spearman correlation in our analysis, because the data distribution is unknown.

4.2.2 Results

Table 4 shows the data of the Java projects considered in this work, in which column n is the number of classes. Table 5 shows the Pearson correlation between K3B and the co-change data, and Table 6, the Spearman correlation. Both results indicate that $K3B(i)$ is related to changes performed in real scenarios of modification process.

Table 4 Historical co-change data and K3B.

Project	**n**	P_D	$P_D a$	$P_D b$	$P_D c$	**K3B(1)**	**K3B(2)**	**K3B(n/2)/(n/2)**	**K3B(n)/n**
aspectj	31	57.8	76.0	60.5	55.2	3.13	6.13	2.33	1.83
fudaa	15987	34.5	48.8	37.3	31.1	3.02	6.05	2.26	1.78
jpox	732	35.1	57.1	41.9	30.5	2.77	5.54	2.12	1.70
azureus	5446	38.2	45.6	38.2	36.9	1.81	3.62	1.56	1.36
rodin-b-sharp	166	21.8	47.0	18.8	18.2	1.75	3.51	1.53	1.34
university	388	26.8	38.6	12.0	28.6	1.73	3.46	1.51	1.33
jaffa	538	29.5	28.3	28.0	29.9	1.58	3.17	1.41	1.76
squirrel	832	27.4	34.7	28.9	25.3	1.47	2.96	1.34	1.22
xmsf	79	34.8	29.1	46.4	34.3	1.46	2.91	1.33	1.21
hibernate	251	45.6	72.9	56.9	39.4	1.99	3.98	1.56	1.55

Table 5 Pearson correlation among co-change data and K3B values.

	K3B(1)	**K3B(2)**	**K3B(n/2)/(n/2)**	**K3B(n)/n**
P_D	0.60	0.59	0.57	0.52
$P_D a$	0.70	0.69	0.66	0.51
$P_D b$	0.49	0.48	0.45	0.42
$P_D c$	0.52	0.51	0.50	0.47

The correlation between K3B and P_D, when performing a change in one class ($i = 1$), is 0.6, according Pearson, and 0.56, according Spearman. This result indicates a positive correlation between K3B and the co-change data, i.e, the higher the number of co-changes, the higher the K3B value. From the results, we observe that the correlation values slightly decrease as i grows. This result might indicate that K3B precision decreases a little for higher values of i.

Geipel and Schweitzer (2012) detailed the co-change data according three types of dependencies among classes: inheritance, use of variable, and use of methods. The correlations between K3B and the probability of changes due to use of variable (P_{Db}), and due to use of methods (P_{Dc}) are positive and relevant, near to 0.5. The correlations found for P_{Db}, however, are a little lower than P_{Dc}. This result indicates that K3B can predict the change propagation effect due to use of variables or methods.

The strongest Pearson correlation, 0.7, appeared between K3B and the probability of changes due to inheritance (P_{Da}). Using Spearman, the correlation between K3B and co-change due to inheritance is even higher, 0.9. These correlations were the highest we have found in our study. Geipel and Schweitzer (2012) has noticed inheritance leads to significantly strongest dependencies. This result, then, suggests that K3B is able to detect the main ripple effect in software systems.

These results refer to a sample of ten Java projects projects. To depict any conclusion about the population projects having as basis these results, it is necessary to use the hypothesis test for the population correlation, denoted by ρ. We, then, tested if $\rho = 0$, with a significance level of 0,05, i.e., 95%. The hypothesis are: H_0: $\rho = 0$, and H_1: $\rho > 0$. For the test purposes, we used the Pearson correlation in the case of K3B(1). The resulting test statistic is $t = 2.13$, and the critical value of the test is t_c=1,86. As $|t| > t_c$, H_0 is rejected. We may conclude, then, that $\rho > 0$. The *p-value* of the test is 0,033, i.e, 97%. The *p-value* of a test is the lowest significance level that will lead to reject the null hypotheses. This means that it is possible to state there is a positive correlation between K3B and co-change.

The co-change data of Geipel and Schweitzer (2012) reflect the change propagation effect in real scenarios. This study compared K3B with such data and found a positive correlation between them. A consequent conclusion is that K3B is, in fact, correlated with change propagation effect.

Table 6 Spearman correlation among co-change data and K3B values.

	K3B(1)	**K3B(2)**	**K3B(n/2)/(n/2)**	**K3B(n)/n**
P_D	0.56	0.56	0.55	0.48
$P_D a$	0.90	0.90	0.85	0.58
$P_D b$	0.44	0.44	0.40	0.32
$P_D c$	0.51	0.52	0.50	0.44

5 Application of K3B

The proposed model has the following main applications in real scenarios of software engineering.

5.1 Estimating change impact

Consider the following scenario. The client requests a change in the system. The developer performs a preliminary analysis on the system and finds out the classes that need to be changed in order to attend the request. Based on this evaluation, the developer defines how much time the activity will take and how much it will cost. However, during the change process the developer realizes that the activity is much more complex because changing these classes will impact other classes of the systems. Moreover, due to the high complexity and the large size of the system, the developer notices that it is very difficult, or even impossible, to manually determine the time and the cost of the activity. The K3B model provides an automatic and simple way for predicting change impact in software systems. The higher the value of K3B, the greater the number of steps in the modification process. Consequently, a high value of K3B indicates that high amount of time should be spent in the activity and that the cost of the software maintenance would be high. Therefore, both developer and manager may consider K3B to plan the modification process more properly.

5.2 Refactoring

A high value of K3B indicates that performing change in the software will be costly. The K3B formulae have as parameters α and β, which represent aspects that determine the tendency for change propagation in the software system. Thus, in order to reduce the cost of maintenance, the developer should control these aspects: reducing α and increasing β. In order to reduce the K3B value, and, hence, to improve the software structure, it is necessary to improve the factors which lead to high values of K3B. For instance, it could be necessary to improve the internal structure of the classes in order to increase their cohesion by means of a better application of the information hiding principle, or a more proper use of inheritance.

5.3 Comparing software systems

The model may be used to compare systems from the viewpoint of maintainability, in the aspect of change impact. The K3B curve of the software system is particularly important in this comparison. The K3B curve provides an overview of how

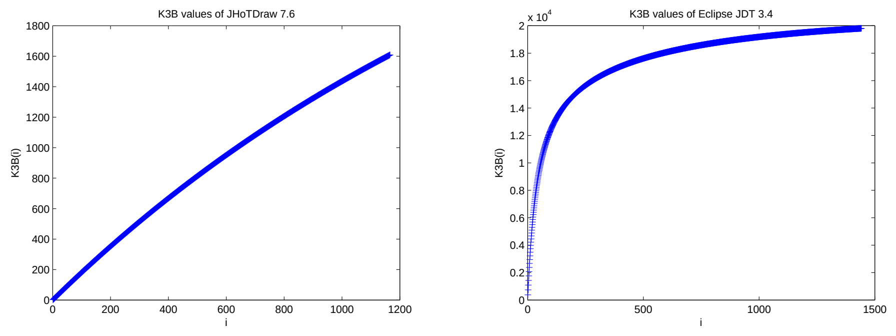


Fig. 7 The K3B curve of (a) JHotDraw 7.6 and (b) Eclipse JDT 3.4.

the change propagation can grow depending on the number of classes that will initially be changed.

Figure 7 shows the K3B curve of two Java projects with approximately the same size, about 1500 classes. By observing the curves, it is possible to notice that a change of a high number of modules in Eclipse JDT may cause a huge impact in the software system, when compared to JHotDraw.

6 Related Work

Changeability, the easiness of performing changes in a software system, is a relevant characteristic of maintainability, especially for environments in which changes are frequent. Controlling the pervasiveness of change propagation is notably important for the management of software maintenance. Some works have been carried out on the topic of change propagation since the early days of Software Engineering (Myers 1975; Rajlich 1997; Hassan and Holt 2004; German et al 2009; Li et al 2009, 2010). These works use different approaches, such as the analysis of dependence between modules, the analysis of historical data about changes performed, and simulation of change propagation (Li et al 2012). Following, we discuss some of them.

To demonstrate the explosive effect that changes may have on software, Myers (1975) uses the example of a circuit with 100 lamps, which correspond to modules in a system. The lamps can assume two states: *on* and *off*. A lamp in the *on* state corresponds to a module which is suffering a change. A lamp in the *off* state corresponds to a module which is not suffering a change. If a lamp is *on*, the probability it will be *off* in the next second is 0.5. An *off* lamp will keep this state while it is connected only to *off* lamps. An *off* lamp which is connected to an *on* lamp will be *on* in the next second with probability 0.5. The system reaches the equilibrium when all lamps are *off*. Starting with all lamps on the *on* state, Myers (1975) has determined the time of equilibrium of the circuit in three configurations: (1) seven seconds, when there are no connections between the lamps; (2) 20 minutes, when the circuit has 10 groups of 10 lamps, and there is no connections between the groups, but all lamps within a group are connected to each other; (3) 10^{22} years, when all lamps within the circuit are connected one to another. The way we define the modification process in K3B has been inspired by the circuit metaphor of Myers.

Myers (1975) proposed a model for stability of programs which aims to predict the number of modules which will be changed due to a change in a random module of the software system. The model has a complex algorithm which is based on probabilities, the internal cohesion of the modules and the coupling between modules. It was originally proposed for the structured programming paradigm. Ferreira et al (2008) adapted the model to the object-oriented paradigm.

The model proposed by Chaumun et al (1999) aims to evaluate the impact of changes in object-oriented software. Their model basically counts the number of classes which are directly affected by a change in a given class of the system. They performed a case study to evaluate the model in which they considered a single type of change: the change of method signature. The target program of the case study was developed in C++ and consists of 1044 classes. The aim of the case study was to investigate correlation between the results given by the model for each class of the system, and the number of methods of the class. However, they did not report strong correlation between these variables.

A set of heuristics was defined by Hassan and Holt (2004) to predict change propagation, including the following ones: the analysis of historical co-change data in order to identify the software entities that have been changed together at the same time; and the call graphs representing the static dependencies between software entities. A simplifying assumption made by them is that each heuristic will predict the ripple effect of a change carried out on a single software entity, i. e., the heuristics disregard that a change can start in more than one software entity. The proposed heuristics aim to identify the set of software entities which will be affected when a given software entity is changed. They have evaluated the proposed heuristics by using historical change data. They reported that the co-change data is useful in the analysis of change propagation, whereas the code structure is not a good indicator for this analysis.

Zimmermann et al (2005) have defined a tool to predict likely changes by mining related changes previously performed in the software system. When the developer made a change in a given software entity, the tool suggests possible further changes. They carried out an empirical evaluation of their approach with data from eight projects. They found that their approach was able to predict changes with a precision of 40% in stable software systems.

The predicting model defined by Mirarab et al (2007) aims to determine the chance that a particular module has to change when a given set of modules is initially changed. Their model is based on the analysis of dependence between modules and in the analysis of historical data, aiming to identify modules in the system which have been changed simultaneously. This information is used to define the probability of changing a module A when B is changed. They carried out a case study with the program Azureus[5]. In the case study, the results of the model were compared to historical data of changes. The results of the case study indicated a strong correlation between the variables.

Brudaru and Zeller (2008) have defined the concept of *genealogy of changes*, which is a directly acyclic graph whose nodes represent the changes performed in a software system during its lifetime, and whose edges represent the dependencies

[5] (http://sourceforge.net/projects/azureus/)

between the changes. An edge from A to B means that A has caused B. Herzig (2010) and Herzig and Zeller (2011) have used this concept to define a model aiming to capture the long-term impact of changes, i.e., the impact of a change in other ones along the lifetime of the software system. Their model aims to predict code changes to be applied in the software system by analyzing how changes are related one to another historically. They consider the following types of change performed in methods of an object-oriented system: adding method, changing definition of method, removing method, adding new method call, changing method call, removing method call. They carried out an empirical evaluation of their approach, using data from four open-source programs, in which they observed a precision around 70%.

Robillard (2008) has proposed a technique for recommending program elements, such as methods and fields, which the developer should consider when performing a given change in the program. The technique is based on the analysis of the structural dependencies in the program. The algorithm used in the technique has as entry a set of program elements which will be initially changed, and results in another set containing program elements that the developer should consider in the change process. Robillard (2008) has evaluated the proposal in an experiment with five software systems, and has concluded that this kind of analysis can aid developers to identify elements to be changed during the change process.

A simulation-based model was proposed by Li et al (2010) to evaluate change propagation in object-oriented software. In the model, a software system is represented as a weighted graph, in which the nodes correspond to classes and interfaces, and the edges correspond to relationships among them. The weights of the edges are defined according to the type of relationship, such as inheritance, aggregation and dependence. The weight of an edge represents the probability of a change in a module will cause change in the other one. The model proposed by Li et al (2010) considers only atomic modifications, i.e., it considers that a single module will be changed each time and, then, it simulates the change propagation. The algorithm applied in the simulation is the following: a class of the target system is selected randomly; the adjacent classes are, then, affected iteratively; the result of the simulation is the number of affected classes. For the target software system, many simulations are performed. The final result is the mean value of the results given by the simulations. Li et al (2010) used their model to evaluate five versions of Apache Ant. From the assumption that the model can be used to evaluate the structural quality of a software system, they concluded that Apache Ant has good quality. However, they did not describe any study to demonstrate that the proposed model can evaluate structural quality of software systems properly.

The work of Dagenais and Robillard (2011) addresses the specific problem of changing frameworks and the impact on client applications. The approach is based on the analysis of the framework repository in order to identify the changes performed on it and, then, recommend needs of changes in the client application.

Kawrykow and Robillard (2011) analyzed data from four open-source systems, and have concluded that up to 15.5% of changes performed in a software system are non-essential, such as local variable extraction and rename refactoring. Regarding this result, they evaluate that any change-based approach should consider only the relevant changes, because elimination of non-essential changes may improve the change-based approach. Moreover, they consider that their technique may be used with an approach of impact of code changes by detecting changes of low impact in the system.

A very interesting empirical work on the topic of change propagation was carried out by Geipel and Schweitzer (2012). They investigated the relationship between class dependence and change propagation. In their study, data from 35 large open-source Java projects were analyzed. The data were gathered from CVS logs, so that they captured the occurrences of classes that have been changed together, what they called *co-change*. Their results support the idea that direct dependencies are propagators of changes. They also concluded that indirect dependencies are important and have to be considered when modeling change propagation. We used the data reported by Geipel and Schweitzer (2012) as a benchmark to evaluate the results of K3B in the present work.

The approach used to model the process of changing propagation in the present work is innovative. We represent the software change process as a stochastic process. As a result, we defined a generic model named K3B. The main differences between K3B and the previous models are the following:

- K3B is a formula whose parameters are software metrics. This may aid software engineers in the task of identifying the aspects of the target software system that contribute to the high rate of change propagation.
- The model is not limited to single changes. K3B has as input parameter the number of modules which will be initially changed.
- The idea of *change step* is introduced in the model to represent the fact that a module is taken to be changed in a given moment and its changing process must be declared as concluded in a given moment too.
- In a change process, a module may be changed many times, due to cyclic dependence between modules within a system. The proposed model is not limited to the number of classes which are affected in a change process. The result of K3B is the estimated number of *change steps*, which considers that a class may be changed more than once in a change process.
- The K3B model was designed to compare the change propagation effect of software systems, by providing as result a measure that indicates how far a set of changes will propagate throughout them. K3B also shows how the change propagation effect of a program behaves as the number of modules that are initially changed grows.
- The implementation of the K3B is simple.
- The model is generically defined in terms of *modules*, in such a way its definition may be applicable to any software development paradigm. In this work, we focus in the object-orientation, by providing an implementation and an evaluation of K3B in this paradigm.

7 Threats to Validity

In this section, we discuss the main threats to validity of this work.

7.1 External validity

We identify four main threats to the external validity of this work. To define the K3B model, a software system is taken as a group of modules connected one to another. The metric that we used to compute ϕ considers only the explicit relationships between the modules, which can be identified by source code analysis. However, a software system can have relationships which cannot be discovered in such a way. For instance, in an object-oriented program, the classes which use a particular file are coupled, even if they do not directly use one another. In this situation, a change in one of those classes may propagate to the other ones. The metrics used for α and ϕ in our implementation of K3B do not consider situations like these. Then, it is not possible to claim that the results of this study can be generalized for programs with such characteristics.

The evaluation of K3B was limited to $1 \leq i \leq 5$ because the simulation for $i > 5$ was not feasible. Hence, is not possible to assure that the achieved conclusions can be generalized for $i > 5$.

An accurate evaluation of K3B in real scenarios of Software Engineering will demand gathering data about the number of change steps resulting of each set of contractual changes performed in a program. It is difficult, or even impossible, to obtain this information from the repositories of open-source software systems, since, in general, they do not maintain detailed information at this level. The data we used to evaluate K3B in real scenarios are not in such level. However, they provide sufficient information to allow a suitable comparison with K3B. The results of such comparison have revealed that K3B, which is a mathematical model, generates results that correspond to the real events that it aims to represent.

The evaluation of K3B was performed using only open-source software systems due to the difficulty of obtaining data from proprietary software. There is no apparent reason to believe that the results would not be applicable to proprietary software. However, from the achieved results of this analysis, we are not able to assure that our conclusions can be generalized to proprietary software.

7.2 Internal validity

We used the sample of open-source Java projects from the study of Geipel and Schweitzer (2012) to evaluate K3B. As we compared the results of K3B with the data reported by Geipel and Schweitzer (2012), it was important ensuring that we are processing the same sample of data. However, they mentioned in their paper only the date of the releases, that is 2008. Due to this, we had problems to identify the proper releases of the programs. To overcome this issue, we considered only the projects for which we could locate the releases with the date reported by Geipel and Schweitzer (2012).

K3B is parametric. One of its parameters is α, a factor that contributes to the high rates of change propagation. In our implementation of K3B, we considered the level of coupling as α. A threat to the validity of our study is that there is no previous work that defines exactly the weight to be considered for each kind of coupling among classes. For this reason, we defined those weights based on calibration of the model, performed in preliminary experiments. Nevertheless, the weights we defined in our implementation are close to the data of co-change found by Geipel and Schweitzer, given in terms of probabilities. Anyway, further empirical studies to identify more precise values for those weight would be appreciable.

7.3 Construct validity

The proposed model do not consider "which modules are inconsistent", but "how many modules are inconsistent". This information is represented by the parameter i in the K3B formulae. K3B is a predicting model and, then, it aims to provide an estimated number of change steps. Nevertheless, knowing "which modules are inconsistent" is still important in maintenance scenarios, and further models should be provided to calculate the exact number of change steps due to the changes in specific modules.

The K3B formulae, given by Equations 4 and 5, were defined from the pattern of the expressions, as described previously. Those expressions were generated for values of n from 4 to 25. Although the K3B formulae are massively applied to all the resultant expressions, there is no formal demonstration that they are applicable for all value of n. Nevertheless, there is no reason to believe that this is not the case. In the studies reported in this paper, the results obtained by means of K3B was systematically compared with data obtained from simulation, as well as with data reported previously in the literature. In both comparative studies, the data reported by K3B is remarkably verified as good predictor of the change propagation effect in object-oriented projects.

8 Conclusions

Software maintenance is responsible for most part of the total cost of a software system. Controlling the software maintainability is important to reduce this cost. Besides, the assessment of maintenance difficulties might help managers and developers in allocating resources to the maintenance tasks.

In the present work we defined a novel predicting model for change propagation in software. The model, called K3B, predicts the number of change steps that a modification process will take. K3B was defined based on probability concepts, especially Markov Chains. K3B is a formula with five variables: n, which is the total number of modules within the system; i, the number of modules which will be initially changed; ϕ, the percentage of connections between modules of the system; α represents a factor which favors change propagation; and β represents a factor which avoids change propagation. In our implementation of K3B, we used a coupling metric for the parameter α, and a cohesion metric for β.

The K3B formulae indicate that the number of change steps in a strongly connected system with high α and low β is explosive. This indicates that changing a software system with such properties is an intractable problem. On the other hand, even for large size programs, when ϕ, α and β are suitable, the change propagation effect might be well controlled.

We evaluated K3B by comparing its results with those given by simulation. In this study, we used 37 releases of 11 open-source software systems. The results of the evaluation showed that there is a strong correlation between the values given by K3B and the values obtained from simulation. The results reported by K3B were also compared with historical data of change propagation. We have also found a positive correlation between K3B and the historical data.

The K3B model provides an automatic and simple way for predicting change impact in software systems. The higher the value of K3B, the greater the number of steps in the modification process. Consequently, a high value of K3B indicates that high amount of time should be spent in the activity and that the cost of the software maintenance would be high.

The K3B model can be used to evaluate software maintainability since it is an indicator of how widely a change or a set of changes would propagate throughout the system. The main application of the model is the comparison of the change propagation effect of software systems, since it indicates how this effect behaves as the number of modules that are initially changed grows.

Based on the results of this work, we identify many directions for further work, such as: new implementations of K3B in order to evaluate other candidate metrics for α, β, and ϕ; empirical studies to accurately identify the weights of the different types of connections between modules; an extension of K3B in order to consider which classes are changed during the maintenance process; development of proper tools based on K3B to detect which classes will be affected during the modification process; evaluation of K3B in representative industrial software systems; and replication of this study with other programming environments.

A The Expressions Obtained

This appendix shows the expressions obtained for number of modules from 4 to 7. The symbol a corresponds to the parameters $\alpha\phi$, whereas b corresponds to β, respectively.

4 modules

$$\begin{bmatrix} 1 + 6\,\frac{a}{b} + 12\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 2 + 10\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 3 + 12\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 4 + 12\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \end{bmatrix}$$

5 modules

$$\begin{bmatrix} 1 + 8\,\frac{a}{b} + 24\,\frac{a^2}{b^2} + 48\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 2 + 14\,\frac{a}{b} + 36\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 3 + 18\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 4 + 20\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 5 + 20\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \end{bmatrix}$$

6 modules

$$\begin{bmatrix}
1+10\,\frac{a}{b}+40\,\frac{a^2}{b^2}+120\,\frac{a^3}{b^3}+240\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}\\
2+18\,\frac{a}{b}+64\,\frac{a^2}{b^2}+168\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}\\
3+24\,\frac{a}{b}+76\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}\\
4+28\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}\\
5+30\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}\\
6+30\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5}
\end{bmatrix}$$

7 modules

$$\begin{bmatrix}
1+12\,\frac{a}{b}+60\,\frac{a^2}{b^2}+240\,\frac{a^3}{b^3}+720\,\frac{a^4}{b^4}+1440\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
2+22\,\frac{a}{b}+100\,\frac{a^2}{b^2}+360\,\frac{a^3}{b^3}+960\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
3+30\,\frac{a}{b}+124\,\frac{a^2}{b^2}+408\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
4+36\,\frac{a}{b}+136\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
5+40\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
6+42\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}\\
7+42\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6}
\end{bmatrix}$$

References

Abreu FB, Carapuça R (1994) Object-oriented software engineering: Measuring and controlling the development process. In: Proceedings of 4th Int. Conf. of Software Quality, McLean, VA, USA

Bieman JM, Kang BK (1995) Cohesion and reuse in an object-oriented system. SIGSOFT Softw Eng Notes 20:259–262, DOI http://doi.acm.org/10.1145/223427.211856, URL http://doi.acm.org/10.1145/223427.211856

Brudaru II, Zeller A (2008) What is the long-term impact of changes? In: Proceedings of the 2008 international workshop on Recommendation systems for software engineering, ACM, New York, NY, USA, RSSE '08, pp 30–32, DOI 10.1145/1454247.1454257, URL http://doi.acm.org/10.1145/1454247.1454257

Chaumun MA, Kabaili H, Keller RK, Lustman F (1999) A change impact model for changeability assessment in object-oriented software systems. In: Proceedings of the Third European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, pp 130–139

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493, DOI 10.1109/32.295895, URL http://dx.doi.org/10.1109/32.295895

Dagenais B, Robillard MP (2011) Recommending adaptive changes for framework evolution. ACM Trans Softw Eng Methodol 20(4):19:1–19:35, DOI 10.1145/2000799.2000805, URL http://doi.acm.org/10.1145/2000799.2000805

Ferreira KAM (2011) Um Modelo de Predição da Amplitude de Propagação de Modificações Contratuais em Software Orientado por Objetos. Doctoral Dissertation.Computer Science Department, Federal University of Minas Gerais, Brazil

Ferreira KAM, Bigonha MAS, Bigonha RS (2008) Reestruturação de software dirigida por conectividade para redução de custo de manutenção. Revista de Informática Teórica e Aplicada 15(2):155–179

Ferreira KAM, Bigonha MA, Bigonha RS, Almeida HC, Moreira RCN (2011) Métrica de coesão de responsabilidade - a utilidade de métricas de coesão na identificação de classes com problemas estruturais. In: X Brazilian Simposium on Software Quality - SBQS'2011, Curitiba, Paraná, Brazil, pp 9–23

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Geipel MM, Schweitzer F (2012) The link between dependency and cochange: Empirical evidence. IEEE Trans Softw Eng 38(6):1432–1444, DOI 10.1109/TSE.2011.91, URL http://dx.doi.org/10.1109/TSE.2011.91

German DM, Hassan AE, Robles G (2009) Change impact graphs: Determining the impact of prior codechanges. Inf Softw Technol 51(10):1394–1408, DOI 10.1016/j.infsof.2009.04.018, URL http://dx.doi.org/10.1016/j.infsof.2009.04.018

Grinstead CM, Snell JL (1991) Introduction to Probability. America Mathematical Society

Hassan AE, Holt RC (2004) Predicting change propagation in software systems. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '04, pp 284–293, URL http://dl.acm.org/citation.cfm?id=1018431.1021436

Herzig K, Zeller A (2011) Mining cause-effect-chains from version histories. In: Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering, IEEE Computer Society, Washington, DC, USA, ISSRE '11, pp 60–69, DOI 10.1109/ISSRE.2011.16, URL http://dx.doi.org/10.1109/ISSRE.2011.16

Herzig KS (2010) Capturing the long-term impact of changes. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ACM, New York, NY, USA, ICSE '10, pp 393–396, DOI 10.1145/1810295.1810401, URL http://doi.acm.org/10.1145/1810295.1810401

Hitz M, Montazeri B (1995) Measuring coupling and cohesion in object-oriented systems. In: Proceedings of the 1995 Int. Symposium on Applied Corporate Computing, Int. Symposium on Applied Corporate Computing, Monterrey, Mexico, pp 1–10

Kawrykow D, Robillard MP (2011) Non-essential changes in version histories. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 351–360, DOI 10.1145/1985793.1985842, URL http://doi.acm.org/10.1145/1985793.1985842

Li B, Sun X, Leung H, Sai Z (2012) A survey of code-based change impact analysis techniques. Softw Test Verif Reliab pp 1–34, DOI 10.1002/stvr.1475

Li L, Qian G, Zhang L (2009) Evaluation of software change propagation using simulation. In: Proceedings of the 2009 WRI World Congress on Software Engineering - Volume 04, IEEE Computer Society, Washington, DC, USA, WCSE '09, pp 28–33, DOI 10.1109/WCSE.2009.22, URL http://dx.doi.org/10.1109/WCSE.2009.22

Li L, Zhang L, Lu L, Fan Z (2010) Assessing object-oriented software systems based on change impact simulation. International Conference on Computer and Information Technology pp 1364–1369

Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall International Series, USA

Mirarab S, Hassouna A, Tahvildari L (2007) Using bayesian belief networks to predict change propagation in software systems. International Conference on Program Comprehension pp 177–188

Myers GJ (1975) Reliable software through composite design, 2nd edn. Petrocelli/Charter, New York

Petrov V, Mordecki E (2003) Teoría de Probabilidades. Matematnka, Editorial URSS, Moscow

Pressman RS (2009) Software Engineering: A Practitioner's Approach, 7th edn. McGraw Hill

Rajlich V (1997) A model for change propagation based on graph rewriting. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '97, pp 84–91, URL http://dl.acm.org/citation.cfm?id=645545.656039

Robillard MP (2008) Topology analysis of software dependencies. ACM Trans Softw Eng Methodol 17(4):18:1–18:36, DOI 10.1145/13487689.13487691, URL http://doi.acm.org/10.1145/13487689.13487691

Sommerville I (2011) Software Engineering 9. Pearson Education

Zimmermann T, Weissgerber P, Diehl S, Zeller A (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445, DOI 10.1109/TSE.2005.72, URL http://dx.doi.org/10.1109/TSE.2005.72

Fig. 8 Kecia Ferreira received her Ph.D. in Computer Science from Federal University of Minas Gerais (Brazil) in 2011. She is a professor of Computer Engineering at Federal Center for Technological Education of Minas Gerais. Her main research interest is software measurement and its applications in software development and maintenance. Her most recent research has focused on software metrics, software evolution and software maintenance.

Fig. 9 Mariza Bigonha received her Ph.D. in Computer Science from Pontifical Catholic University Rio de Janeiro (Brazil) in 1994. She is a professor of Computer Science at Federal University of Minas Gerais (Brazil) since 1994. Her main research interests are Programming Languages, Compilers and Code Optimization.

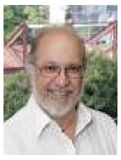


Fig. 10 Roberto Bigonha received his Ph.D. in Computer Science from University of California, Los Angeles (1981). He is a professor of Computer Science at Federal University of Minas Gerais (Brazil) since 1974, and he is also a member of the Brazilian Computer Society. His main research interests are Programming Languages, Compilers and Formal Semantics.

Fig. 11 Bernardo N. B. de Lima received his Ph.D. in Mathematics from National Institute for Pure and Applied Mathematics (IMPA), Rio de Janeiro-Brazil in 2003. He is a professor of Mathematics at Federal University of Minas Gerais (Brazil) since 1999. His main research interest are Probability Theory, Statistical Mechanics and Discrete Mathematics with emphasis in the study of percolative models.

Fig. 12 Luiz Felipe O. Mendes is a Bachelor in Computer Science by Universidade Federal de Minas Gerais. He is a data scientist and co-founder of Hekima, a Brazilian company of Big Data and Artificial Intelligence. His main interest are Machine Learning and Data Analytics.

Fig. 13 Bárbara M. Gomes is a Computer Engineering by Federal Center for Technological Education of Minas Gerais. Her main interests are Software Engineering and Optimization.