Predicting the Probability of Change in Object-Oriented Systems

Nikolaos Tsantalis, Alexander Chatzigeorgiou, *Member*, *IEEE Computer Society*, and George Stephanides, *Member*, *IEEE Computer Society*

Abstract—Of all merits of the object-oriented paradigm, flexibility is probably the most important in a world of constantly changing requirements and the most striking difference compared to previous approaches. However, it is rather difficult to quantify this aspect of quality: This paper describes a probabilistic approach to estimate the change proneness of an object-oriented design by evaluating the probability that each class of the system will be affected when new functionality is added or when existing functionality is modified. It is obvious that when a system exhibits a large sensitivity to changes, the corresponding design quality is questionable. The extracted probabilities of change can be used to assist maintenance and to observe the evolution of stability through successive generations and identify a possible "saturation" level beyond which any attempt to improve the design without major refactoring is impossible. The proposed model has been evaluated on two multiversion open source projects. The process has been fully automated by a Java program, while statistical analysis has proved improved correlation between the extracted probabilities and actual changes in each of the classes in comparison to a prediction model that relies simply on past data.

Index Terms—Object-oriented programming, product metrics, object-oriented design methods, quality analysis and evaluation.

1 Introduction

According to Parnas [30], software engineering deals with "the construction of multiversion software," that is, software that will undergo a number of revisions either to enhance the functionality or to fix bugs. The ability to perform corrective, adaptive or perfective maintenance [6] in an existing object-oriented design has been associated with many closely related terms such as changeability [2], maintainability [16], robustness to changes [15], extensibility [33], and flexibility [25]. A number of terms have also been used in order to describe the lack of the above features in a design such as rigidity, fragility, inflexibility, limited evolvability, etc. The context, in which all of the above terms are usually used, is that changes are unavoidable in software development and that anticipation of changes by a software designer is of major importance.

The object-oriented paradigm offers a number of features that are meant to facilitate the development of flexible software, if employed correctly. By flexibility it is meant that the principles of encapsulation, information hiding, abstraction, inheritance, and polymorphism should be correctly applied so as to remove any odors of fragility and rigidity [25]. The latter properties characterize a design that is easy to break or difficult to change, respectively. According to [19], most of the aspects related to good code writing hinge on a single underlying quality, namely, flexibility.

• *The authors are with the Department of Applied Informatics, University of Macedonia, 156 Egnatia st., 54006 Thessaloniki, Greece. E-mail: nikos@java.uom.gr, {achat, steph}@uom.gr.*

Manuscript received 12 Oct. 2004; revised 29 Apr. 2005; accepted 5 May 2005; published online 26 July 2005.
Recommended for acceptance by J.-M. Jezequel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0207-1004.

Practically, the addition of new functionality in an object-oriented system should have as limited impact on existing code as possible. If the modification of a class method imposes code changes to a number of existing classes, object-orientation is of limited value. This feature has been successfully captured by the *Open-Closed* Principle [28], which states that "software entities should be open for extension but closed for modification." Flexibility becomes a crucial factor also from an economic point of view since a number of studies conclude that the largest percentage of software development effort is spent on rework and maintenance. There are many design principles [25], heuristics [33], and patterns [15] that help to enforce good programming practices in order to build more stable and flexible systems.

In order to characterize several aspects of a software system a large number of metrics has been proposed [13] and, since the initial work of Chidamber and Kemerer [11], the field of software metrics has been expanding to the object-oriented domain as well. Many of these metrics have been both theoretically justified and empirically validated while others lack a systematic validation on real-life industrial software. However, we believe that most of the existing metrics evaluate the degree of object-orientation or measure static characteristics of the design, which are not always helpful in answering the question whether a specific design is good or not [22]. When trying to answer such a question, an expert would assess the conformance of the design to well established rules of thumb, heuristics, and principles. This work attempts to systematize this process by quantifying the change proneness of each class in a design.

In brief, the goal is to assess the probability that each class will change in a future generation. The goal of the proposed method is not to evaluate the change proneness of

a design examining it in isolation. Rather, it should be applied when several successive versions of an application are available. In order to calculate these probabilities, axes of change, through which a change in one class can affect another class of the design, are identified. Statistical results validate that the proposed analysis offers improved prediction accuracy compared to a model that simply considers information from past generations. An improved correlation coefficient is obtained between the calculated probabilities and actual changes for all classes and for all generations of two open source projects. Moreover logistic regression analysis has shown that the proposed model has an increased goodness of fit and a larger impact than previously used measures.

The evaluation is based on simple probabilistic analysis and can be easily automated: A Java program has been developed that can parse a complete hierarchy of directories containing the classes of a design and automatically extract the probabilities of change. For the calculation of probabilities, the tool reads past data for an evolving design, such as the ones developed in the open source community. An additional piece of information that can be extracted from the model are the classes that have a "bad" history of changes and at the same time affect a large number of other classes. These classes can be a localized source of problems to the design and, thus, should be easily identified.

The rest of the paper is organized as follows: In Section 2, the possible axes of change in an object-oriented design are described. The analysis process for the probability estimation along with the assumptions made is presented in Section 3. One sample application employing a design pattern and two larger software applications are extensively analyzed in Section 4 and statistical analysis results are discussed. In Section 5, the software tool that has been developed in order to automate the proposed methodology is described while threats to the validity and limitations of the analysis are mentioned in Section 6. Previous approaches on relevant subjects are described in Section 7 and open research issues in Section 8. Finally, we conclude in Section 9.

2 Axes of Change

In order to emphasize the interference between the classes of a system, the proposed model defines several axes of change through which a change in a class can affect other classes enforcing them to be modified. By change, we mean that given a change in one of the affecting classes, the affected classes should be updated, in order for the software to operate correctly. To describe the way that a change in one module would necessitate a change in any other module, Haney [18] used the term *ripple effect*. For example, the change in the signature of a method in a class will require the update of all classes that use this method. Each class can change because of its involvement in one or more axes of change. The following observations and terminology focus mainly on systems developed using Java; however, the conclusions can be easily ported to any object-oriented programming language.

In general, axes fall in three main categories. For each case, an example of how change can propagate is briefly mentioned:

Inheritance axis:

- *Interface*. A class implements one or more interfaces (inherits pure abstract classes for C++). For example, the addition of a new method in the interface or a change in the signature of an existing method enforces all classes implementing this interface to modify themselves in order to be compliant with this change.
- *Class Inheritance*. A class inherits another class. In case of an abstract base class, the addition of a new abstract method or a change in the declaration of an existing abstract method enforces all classes that inherit (extend) this abstract class to modify the corresponding implementing methods. For a nonabstract class, if one class employs the constructor of its superclass or explicitly uses a method of the superclass (e.g., via the `super` identifier), any change in the signature of the constructor or method imposes the subclass to be modified.

Reference axis.

- *Direct instance*. A class instantiates an object (employing a `new` operation for example). A change in the signature of the constructor implies that all classes that create instances of this class have to modify the corresponding constructor call.
- *Reference*. A class employs an object as a parameter in its constructor or one of its methods. A change in the declaration of one method (whether static or nonstatic) enforces all classes using that method to modify the corresponding method calls. On the other hand, all changes in the body of a method do not affect classes that employ that method (encapsulation).

Dependency axis. By this axis, it is meant that a change in any class or package name on which a class depends, will enforce changes to the dependent class. A change in the package name will enforce the update of all import statements throughout the program. The renaming of a class or the package which contains it is not treated as a new class but as a change in that particular class.

The above three axes are related to a possible modification of a class' probability of change due to other classes and, therefore, will be called **external axes** of change.

However, since each class can also change due to modifications to the class itself, we define also an **internal axis** of change that summarizes all possible causes of change: modification to method declarations, addition of new methods/attributes, change of implementation, etc. This axis refers to changes originating from the class itself and not changes that have propagated from other classes: Nevertheless, it has to be taken into account since a class with a "bad" history of changes will contribute to the overall system's probability of change, possibly by affecting other classes as well. It is important to mention that in the extraction of changes related to the internal axis, changes which originate from other classes should not be counted;

otherwise, the same effect will be counted twice in the model, once as an internal change and then as a modification of the probability of propagated changes.

At this point, it should be noted that dependence on library classes (such as STL in C++ or API's in Java) is not considered a source of changes since these classes are not likely to change. Axes involving such classes are not taken into account in the analysis. If these classes were taken into account, the analysis would become more complicated due to a larger number of classes and, more important, the zero or close to zero probabilities of change for these classes, would unavoidably affect the evaluation of the system under study.

3 Probability Estimation

3.1 Analysis

Given a class A in which a change can occur, the aim is to calculate the probability that another class C that can be affected, has to change.

The probability that a class C might change in the next generation of the software will be denoted as $P(C)$. Since the only possible events are that 1) the class changes and that 2) the class does not change (i.e., the sample space is $S = \{"change",\ "no\ change"\}$), the proposed probability on the sample space of the two outcomes satisfies the following properties:

1. Any probability is a number between 0 and 1: $0 \leq P(C) \leq 1$. (The probability is physically measured in a class that undergoes a number of modifications through successive generations, as the ratio of the number of changes over the number of generation upgrades. Since this number can be at least 0 and at maximum 1, it follows that the range of the probability function is [0..1].)
2. The sample space, S, of all possible outcomes has a probability of 1: $P(S) = 1$.
3. Since the two events are disjoint, $P("change" \cup "no\ change") = P("change") + P("no\ change")$ and, thus, P is a valid probability measure [29].

As already mentioned, every class is subject to change due to its involvement in several axes of change. Since even one change will be a reason for editing the code, the probability in which we are interested is given by the *joint probability* of all events (i.e., change in any axis), also known as probability of the OR of two or more events. For example, if a class C can change due to two axes of change, $axisA$ and $axisB$, the probability that C will change is given by [1]:

$$\begin{aligned} P(C) &= P(C : axisA \cup C : axisB) \\ &= P(C : axisA) + P(C : axisB) - P(C : axisA \cap C : axisB) \\ &= P(C : axisA) + P(C : axisB) - P(C : axisA) \cdot P(C : axisB) \end{aligned}$$

assuming that changes originating from two different axes are independent. This probability is always lower or equal to one. $P(C : axisX)$ symbolizes the probability that class C will change due to $axisX$.

However, one change in a class does not always propagate to the associated classes. For example, the modification of a method's body in a class does not cause a change to a client class (excluding the case when pre and postconditions of the method are changed). For this to happen, the signature of a method that is being used by the client class has to change. Therefore, the probability of change for the client class should be calculated as a conditional probability upon the probability of change in the other class. Let us assume that class C is involved in one external axis (e.g., has a reference to class A and employs one of its methods). Then, the probability $P(C : axisA)$ is calculated as:

$$P(C : axisA) = P(C|A) \cdot P(A),$$

where $P(C|A)$ is the conditional probability of a change in class C with respect to a change in class A, read as *the probability* of C given A. The term $P(C|A)$ essentially represents the probability that a change in class A will eventually propagate to class C. In other words, it is a probability associated to the axis involving both classes. Within a broader perspective, it could be measured as the percentage of past changes in class A that have caused changes in class C. The reason for incorporating this conditional probability is to avoid a "worst-case" analysis [34], in which all changes are assumed to propagate to other classes.

Since an internal change will always have an effect on the class to which it is made,

$$P(C : internal) = P(C|C) \cdot P(C) = P(C).$$

In case a global stability measure is sought [6], one could employ the probabilities of all classes in the system to derive a measure that characterizes the whole system according to its probability of change in a future generation (e.g., the mean or median value).

3.2 Assumptions in the Model

In the previous analysis, it has been assumed that the events associated with each axis of change (i.e., a change due to $axisA$ and a change due to $axisB$) are independent, meaning that the outcome of one event does not affect in a direct way the probability of the other. Of course, there could be conceptual dependencies within the rest of the system, but, for the sake of simplicity, such dependencies are ignored. A conceptual dependency implies a connection between two components that cannot be discovered by source code analysis. For example, changes that are due to the same requirement but cannot be identified in the code.

In case of multiple axes of change associating two classes (e.g., inheritance and containment), the previous assumption does not hold (i.e., the changes along the two axes are certainly associated). However, because even one axis is sufficient for propagating the change from one class to another, these multiple axes are considered as one.

The application of the proposed model (presented in Section 4.2) considers the percentage of changes that propagate to other classes (propagation factor). This factor can be specified for each axis separately. For practical reasons, the tool that has been developed allows the user to set a common value for all axes in one generation of the system. For the evaluation of the case studies, we have used a common factor throughout all generations. This common

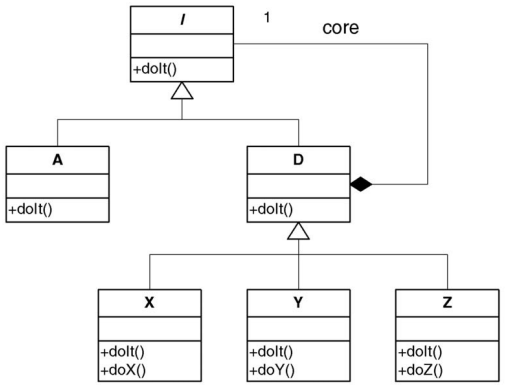


Fig. 1. Sample design using Decorator DP.

factor represents the ratio of the number of propagated changes over the number of all changes.

As already mentioned, changes in a method's body are considered as local, i.e., that they do not propagate. However, this is not true when the pre and postconditions of the method change, causing clients of that method to undergo a change as well. This assumption is made only for practical reasons since code analysis can hardly reveal changes to the pre and postconditions of a method.

4 Application Results—Discussion

4.1 Demonstration of the Methodology

The proposed probabilistic measure does not aim at the evaluation of a given design by examining it in isolation. It rather evaluates the evolution of a design through successive generations and predicts the probability of change for each class. However, in order to demonstrate the application of the proposed model, we have calculated each class' probability of change for a design that employs the *Decorator* Design Pattern [15]. In this example, the applied analysis is partial since past data are not used in order to assess each class' probability of change. The complete application of the methodology will be shown for the real-world examples in the following section.

4.1.1 Decorator Pattern

The system that is being used as an example can add dynamically functionality to a base class. However, this is not performed simply using inheritance and composition. The example design has anticipated that future classes with enhanced functionality might be added in the system and, therefore, employs the Decorator pattern. The corresponding UML class diagram is shown in Fig. 1. Classes X, Y, and Z call in their `doIt()` method the corresponding method of the D class, while D class calls in its `doIt()` method the corresponding method of its contained core object, which can be either a leaf object (such as an object of class A or X) or another composite object.

4.1.2 Analysis

Since there are no data from past generations, to estimate the possibility of change in any part of the system, the model assumes that for classes where changes originate, the probability of change is a constant value, with most reasonable value 0.5 (this value refers to the probability related to the internal axis of change). For the conditional probabilities (i.e., the probability that a change will actually propagate from the originating class to the affected ones), one could either set arbitrarily a constant value such as 0.5, or could perform a worst case analysis by setting this value equal to 1 [34]. For the rest of this example, we set all conditional probabilities equal to 0.5.

Abstract classes D and A participate in only one external axis of change with respect to the declaration of method `doIt()` in interface I. (Class D appears to have a second axis due to the containment relationship with class I. However, since we are only interested in the probability that one class is being affected by another, multiple axes of change from one class to another are counted only once). Assuming that a change in interface I will occur with a probability of 0.5 ($P(I) = 0.5$) and that this change will propagate to class D with a probability of 0.5 ($P(D|I) = 0.5$), then:

$$P(D : inheritance\ axis) = P(D|I) \cdot P(I) = 0.5 \cdot 0.5 = 0.25.$$

Consequently, considering also the internal axis:

$$\begin{aligned} P(D) &= P(D : internal\ axis \cup D : inheritance\ axis) \\ &= P(D : internal\ axis) + P(D : inheritance\ axis) \\ &\quad - P(D : internal\ axis) \cdot P(D : inheritance\ axis) \\ &= P(D : internal\ axis) + P(D|I) \cdot P(I) \\ &\quad - P(D : internal\ axis) \cdot P(D|I) \cdot P(I) \\ &= 0.5 + 0.5 \cdot 0.5 - 0.5 \cdot 0.5 \cdot 0.5 = 0.625. \end{aligned}$$

Class A has also one internal axis and one inheritance axis with respect to interface I and, therefore, $P(A)$=$P(D)$=0.625.

Classes X, Y, and Z have only one external axis regarding the inheritance of their superclass D. Any change in the signature of method `doIt()` will cause definite changes to the subclasses since the `doIt()` method of the subclasses calls the `doIt()` method of the superclass. Taking the joint probability due to this axis and that due to internal changes for class X results in:

$$\begin{aligned} P(X) &= P(X : internal \cup X : inheritance) \\ &= P(X : internal) + P(X : inheritance) \\ &\quad - P(X : internal) \cdot P(X : inheritance) \\ &= P(X : internal) + P(X|D) \cdot P(D) \\ &\quad - P(X : internal) \cdot P(X|D) \cdot P(D) \\ &= 0.5 + 0.5 \cdot 0.625 - 0.5 \cdot 0.5 \cdot 0.625 = 0.656. \end{aligned}$$

Classes Y and Z have the same probability of change.

4.2 Analysis of Large-Scale Software

In order to investigate the applicability of the proposed methodology in software of a larger scale, two open-source projects have been examined, namely, JFlex [20] and JMol [21]. These two projects have been selected for analysis because they satisfy the following criteria:

- They have evolved through a number of generations.
- Access to the full source code of each version is possible since they are both open-source projects.
- They contain a relatively large number of classes.

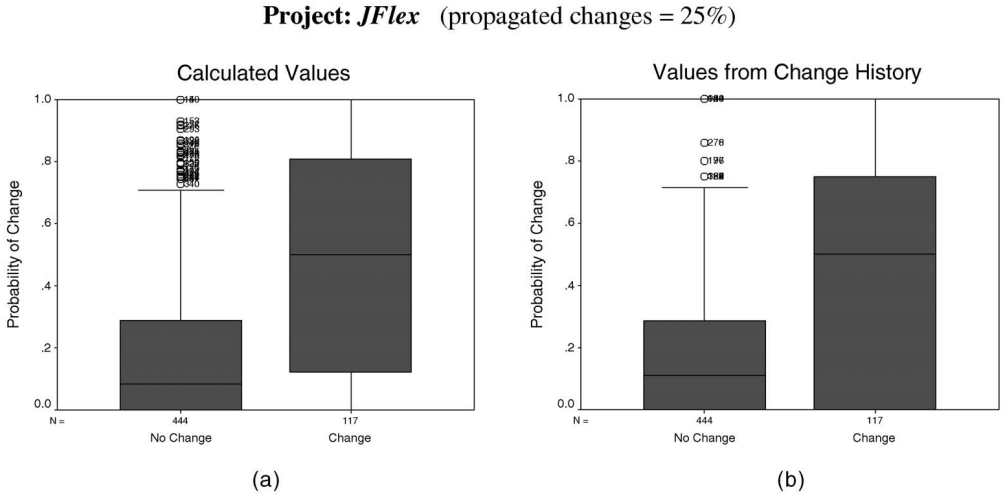


Fig. 2. Boxplots illustrating the correlation between (a) calculated probabilities and changes/no changes in the next generation (correl. = 0.400) and (b) probabilities extracted from change history and changes/no changes in the next generation (corr. = 0.375).

- They have a different percentage of propagated changes. This was important for assessing whether the percentage of propagated changes affects the accuracy of the model.
- Both projects are relatively mature (Registration date on sourceforge.net JFlex: 18 Nov. 2000, JMol: 25 Mar. 2001).
- They are written in Java (Our bytecode parser supports at the moment only Java).

4.2.1 JFlex

JFlex [20] is a "Lexical Analyzer Generator for Java," written in Java, which takes a specially formatted specification file containing the details of a lexical analyzer as input and creates a Java source file for the corresponding lexical analyzer. The source code for JFlex is publicly available, while the latest version that has been examined consists of 58 Java classes. Thirteen subsequent versions have been automatically analyzed using the tool that has been developed for this purpose.

Since past data were now available for each generation (except for the first one), the change history has been analyzed to extract the internal probability of change for each class. For example, in case one class underwent four upgrades and changes took place in three out of the four upgrades, the internal class probability of change at the fifth generation would be $3/4$. However, it is worth mentioning, that only changes that originated in the class itself are considered in this calculation: Changes that have propagated from other classes are not measured within the internal class probability since they are taken into account when the conditional probability related to the originating class is calculated. Otherwise, the model would incorporate the effect of the changes twice. The actual class' probabilities of change have been calculated employing the developed tool that reads all internal probabilities of change (stored in a file), receives as input a factor that represents the percentage of changes that are expected to propagate and analyzes an xml file (or the source code) that contains the static structure of the system. According to the change history for JFlex, the overall percentage of changes that cause a ripple effect on other classes was found to be equal to 25 percent.

In order to check the validity of the extracted probabilities of change by statistical means and to investigate their relation to the actual changes (or no changes) in the classes of JFlex through successive generations, we calculated the correlation between the calculated probability of change for each class and a Boolean variable capturing whether that class was changed in the next generation or not. Since the correlation coefficient has to be calculated between one dichotomous variable and one continuous variable, the point-biserial correlation has been extracted, which is a special case of the Pearson correlation [12].

A boxplot diagram showing the correlation between the Boolean variable in the horizontal axis (change/no change in the next generation) and the probability of change is shown in Fig. 2a. This plot has been extracted from 12 versions of JFlex software (except for the last version for which no validation data were available since there is no further generation) and for all of its classes, namely, from $N = 561$ data pairs. Each data pair contains one value that represents a class' probability of change (that is, class X in generation Y) and one value that represents the actual outcome in the following generation (which was known for all generations but the last one).

To compare the efficiency of the proposed model against a prediction model that is simply based on historic probabilities, in Fig. 2b the boxplot diagram shows the correlation between probability values extracted from change history and changes/no changes in the following generation.

The correlation probability for the calculated values is equal to 0.400, while the correlation for the historic values is 0.375. Both values are significant at the 0.01 level. Both plots illustrate that small values of probability are associated with a *No Change* in the next generation, while higher values of probability are associated with *Change*. However, for 444 classes that have not been changed in the next generation, the median of their calculated probability

TABLE 1
Logistic Regression Results for JFlex

Measure	*Overall accuracy*[1] (%)	*Sensitivity*[2] (%)	*False Positive Ratio*[3] (%)	*False Negative Ratio*[4] (%)	*Nagelkerke* R^2	*B (s.e.)*	*Wald* $X^2(1)$
proposed	68.4	56.4	19.6	43.6	0.236	2.849 (0.472)	36.401 , $p < 0.0001$
history	67.5	59.0	23.9	41.0	0.200	2.610 (0.467)	31.255, $p < 0.0001$
CBO	67.5	56.4	21.3	43.6	0.174	0.141 (0.028)	25.519, $p < 0.0001$
NOO	66.7	53.0	19.6	47.0	0.164	0.080 (0.017)	21.849, $p < 0.0001$
WMC	65.0	43.6	13.6	56.4	0.167	0.022 (0.005)	18.085, $p < 0.0001$
DIT	56.4	85.5	72.6	14.5	0.048	-0.311 (0.114)	7.523, $p < 0.01$
RFC	63.2	45.3	18.8	54.7	0.016	0.004 (0.002)	2.599, not significant
NOC	51.7	12.8	9.4	87.2	0.010	0.183 (0.142)	1.672, not significant
LCOM	61.0	33.7	7.6	66.3	0.141	0.008 (0.002)	11.609, $p < 0.01$
CS	66.2	52.1	19.6	47.9	0.246	0.005 (0.001)	16.184, $p < 0.0001$

[1] Overall accuracy: Percentage of correct classifications
[2] Sensitivity: Percentage of occurrences (next generation = "change") correctly predicted
[3] False Positive Ratio: Percentage of incorrect classifications when the event did not occur (i.e. the percentage of cases where a class was predicted to change, when in fact it did not)
[4] False Negative Ratio: Percentage of incorrect classifications when the event did occur (i.e. the percentage of cases where a class was predicted not to change, when in fact it did). It is equal to 1 – *Sensitivity*.

values is 0.0833, while the median for the historic probabilities increases to 0.1111. For the 117 classes that have changed in the next generation, the median of their probability values is 0.5 in both cases. As it can be observed, the calculated probability values are slightly better in predicting changes in the next generation (the box that represents the middle 50 percent for the "Change" is shifted slightly to the top).

To determine whether the proposed probability measure is a useful predictor of the event of "change" or "no change" logistic regression analysis has been performed. As dependent variable the dichotomous value representing "change" or "no change" in the next generation has been used and the calculated probability value for the current generation as the independent variable. To evaluate the proposed model against other approaches, the same analysis has also been performed using as independent variable, a probability value extracted only from past data (history), a coupling measure (CBO) that has been used in the literature for impact analysis [7], [35], the number of methods per class (NOO) used in [35] to check whether it identifies change prone classes, Chidamber and Kemerer's metrics (WMC, DIT, RFC, NOC, LCOM) used to detect faulty classes [5] and class size (CS) used to investigate its relationship to the effort to implement changes [3]. Results are shown in Table 1.

From the interpretation of the results [31], it can be concluded that the proposed method offers, when the propagation factor is relatively low as in the case of JFlex, a very small improvement concerning the correctly predicted outcomes (overall accuracy, with cutoff point set at 0.5), compared to the simple model that relies only on past data. Thus, regression analysis confirms the initial picture obtained by the boxplots. However, the proposed probability measure seems to achieve a better model fitness (R^2) and a larger effect on the logit of the actual outcome in the next version (where the logit of the dependent variable Y is defined as $logit(Y) = \ln\left(\frac{P(Y=1)}{1-P(Y=1)}\right)$ and according to the simple logistic model is equal to $logit(Y) = a + BX$, where B is the regression coefficient).

Although significant, the effect of all other factors (except for class size) on the logit of the dependent variable, is much lower than that of the proposed method and the simple prediction model relying on history. Since the independent variables are measured at different scales, to compare the effect of each measure, standardized coefficients (B^*) have been calculated [26]. The standardized coefficients for all statistically significant variables are shown in order in Table 2. The standardized coefficients indicate how many standard deviations of change in the dependent variable are associated with one standard deviation of change in the explanatory variable [26].

Apart from this ranking, the low value for Nagelkerge R^2 which is a pseudo-R-square analogous to OLS regression, indicates for all most other measures (except for class size) a low explanatory power of the corresponding model. It is worth mentioning that for the DIT (Depth of Inheritance Tree) metric the coefficient B is negative, which indicates that as the depth in which a class is placed in a hierarchy increases, the class has a lower possibility to change. A possible explanation could be the fact that classes inheriting from a large number of ancestors exhibit a large degree of reuse, therefore minimizing their need to undergo changes.

TABLE 2
Standardized Logistic Regression Coefficients for JFlex

Measure	*Standardized coefficient*
CS	0.495
proposed	0.485
history	0.447
CBO	0.416
WMC	0.407
NOO	0.405
LCOM	0.400
DIT	-0.458

TABLE 3
Results of (a) Forward and (b) Backward Logistic Regression Analysis for JFlex

	Variables	Change in −2 Log Likelihood (if variable is removed)	Significance of the Change
Step 1	proposed	45.668	0.000
Step 2	proposed	7.341	0.007
	CS	9.331	0.002

(a)

	Variables	Change in −2 Log Likelihood (if variable is removed)	Significance of the Change
Step 1	proposed	1.571	0.210
	history	0.067	0.796
	CBO	0.167	0.683
	CS	5.755	0.016
Step 2	proposed	6.894	0.009
	CBO	0.142	0.706
	CS	5.862	0.015
Step 3	proposed	7.341	0.007
	CS	9.331	0.002

(b)

To formally demonstrate that the proposed measure can add value over what can be achieved by a simpler prediction model based only on the most important of other metrics (e.g., history, coupling, and class size), forward and backward logistic regression has been performed. In forward inclusion, the four selected independent variables (proposed, CS, history, and CBO) are initially withheld from the model. At subsequent steps, those variables determined to be significant are added to the model, while all others are withheld. The opposite occurs in backward elimination in which the four independent variables are initially included in the model. At subsequent steps, those variables determined insignificant are eliminated from the model until the remaining variables are all deemed important. Selection (or deletion) of variables is based on the "drop-in-deviance" test that checks whether the inclusion (or exclusion) of a variable makes a significant difference in the goodness of fit. The contribution of a given variable is obtained by computing the difference between the -2 Log Likelihood statistic for the reduced model (the one where the variable is eliminated) and the corresponding value of the full model (the one that includes the variable). The more a variable contributes to the model, the larger the change in -2 Log Likelihood.

The results are summarized in Table 3. As it can be observed, both during stepwise inclusion and elimination of variables, the only two variables that at the end are considered important to the model are the proposed measure and the class size, as already observed by the regression coefficients.

One of the advantages of the proposed methodology is that it can be directly applied in order to identify classes in the design ("hotspots") that have a high probability of change and at the same time can affect a large number of other classes, in case of a change. These classes should be easily spotted since they can degenerate the whole system's change proneness even if the rest of the system is relatively stable. This notion is best captured by the product of each class' probability of change times the number of axis of other classes in which it is involved. Both values are automatically extracted by the program that has been implemented and are easily accessible (see Fig. 7). The five worst classes in terms of this product, for the JFlex design, are shown in Fig. 3, which is the reengineered class diagram for the latest version of the system. The intensity of the gray color corresponds to the value of this product (i.e., class LexScan is according to this measure the worst class since it has a very large probability of change and any change in this class can affect at most four other classes).

The reason for which someone cannot use only the probability of change itself or the number of involved axis alone is best explained through an example. In Table 4, the probabilities of change, number of involved axes, and the corresponding product are shown for the worst five classes as well as for two other classes. It can be clearly observed, that although class *GeneratorException* can affect nine other classes, it has a probability of change equal to 0 since no alterations have been made on this class. On the other hand, class *Main* has a reasonably high probability of change, but changes to this class can never propagate to any other class in the system.

4.2.2 JMol

JMol [21] is an open source molecule viewer capable of displaying high quality 3D images of chemical structures reading a large variety of file types. Nine successive generations have been analyzed, while the latest version consists of 169 Java classes. This system underwent a large number of modifications, many of them having a large impact on the system (such as class and package renaming). As a result, it was observed from the change history that almost 50 percent of the changes that have occurred caused a ripple effect to other classes.

Concerning the validity of the proposed model for predicting changes and the efficiency against a simple history-based model, the boxplots in Fig. 4 show the correlation between the probability values and the event of a change/no change in the next generation, for 1,207 data pairs.

This time, the observed improvement in the accuracy of the predictions is more intense, which is reasonable since the number of propagated changes doubled from the previous example. The calculated probability values differ substantially from their historic ones since the effect of the axes of changes (conditional probabilities) is larger. This is

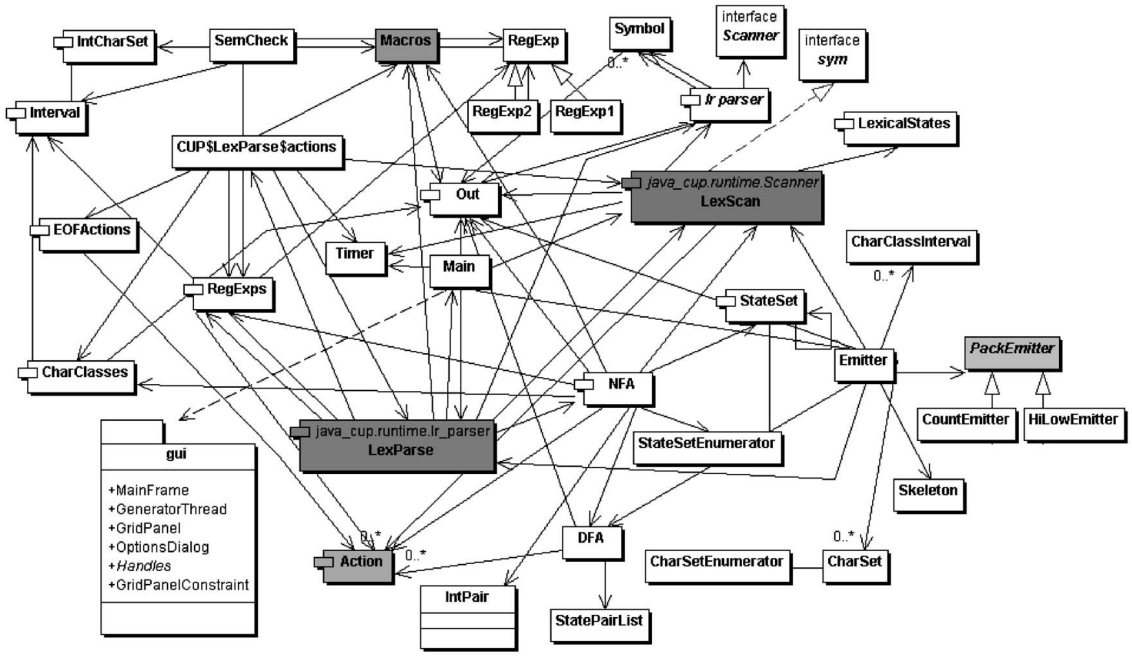


Fig. 3. Class diagram of JFlex (latest version) illustrating classes with a large probability $\times$ axes product.

TABLE 4
Five Worst Classes in JFlex According to the Product of the Calculated Probability Times the Number of Axes of Other Classes in Which They Are Involved

class	probability	#inv. axes	product
JFlex.LexScan	1	4	4
JFlex.LexParse	0.765688	3	2.297064
JFlex.Macros	0.260355	6	1.56213
JFlex.Action	0.307692	5	1.538462
JFlex.PackEmitter	0.5	3	1.5
JFlex.GeneratorException	**0**	**9**	**0**
JFlex.Main	**0.796786**	**0**	**0**

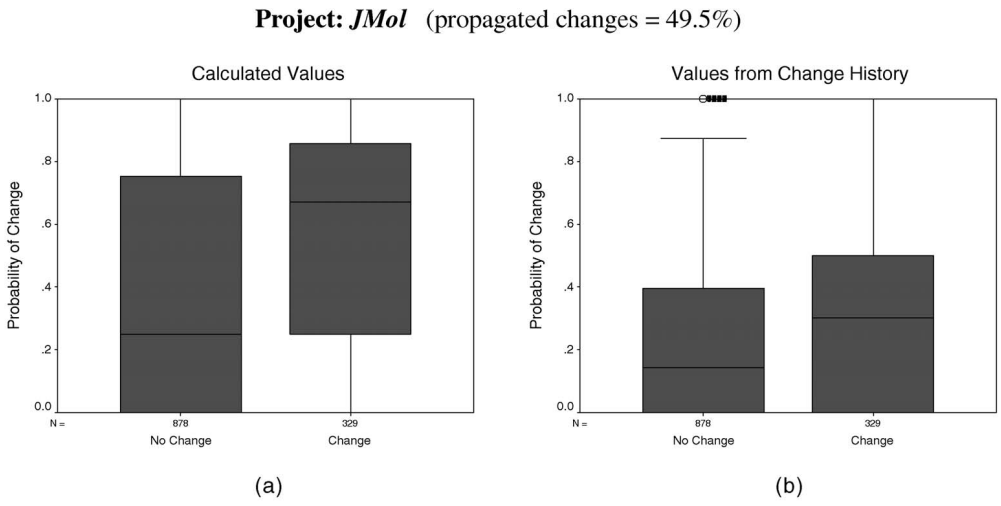


Fig. 4. Boxplots illustrating the correlation between (a) calculated probabilities and changes/no changes in the next generation (correl. = 0.232) and (b) probabilities extracted from change history and changes/no changes in the next generation (corr. = 0.153).

also evident from the shift upward of the 50 percent box for the calculated values associated to a *Change*.

Logistic regression analysis has also been performed for JMol and Table 5 summarizes the results. Although the goodness of fit is now lower, the overall accuracy as well as the correctly predicted changes (sensitivity) for the proposed model is significantly better in comparison to all other possible measures. It is worth mentioning that the history variable presents a sensitivity of 50.8 which is hardly better than that of pure random selection, while its

TABLE 5
Logistic Regression Results for JMol

Measure	*Overall accuracy*[1] (%)	*Sensitivity*[2] (%)	*False Positive Ratio*[3] (%)	*False Negative Ratio*[4] (%)	*Nagelkerke* R^2	*B (s.e.)*	*Wald* $X^2(1)$
proposed	62.9	66.0	40.1	34.0	0.107	1.615 (0.224)	51.847, p < 0.0001
history	57.4	50.8	35.8	49.2	0.057	1.555 (0.300)	26.868, p < 0.0001
CBO	50.0	37.7	37.6	62.3	0.000	0.000 (0.007)	0.000, not significant
NOO	50.8	24.9	23.4	75.1	0.004	0.006 (0.005)	1.535, not significant
WMC	56.4	34.0	21.2	66.0	0.011	0.005 (0.002)	4.609, not significant
DIT	50.3	29.5	28.8	70.5	0.001	0.030 (0.037)	0.672, not significant
RFC	51.7	22.2	18.8	77.8	0.002	0.001 (0.001)	1.197, not significant
NOC	57.6	88.8	73.5	11.2	0.002	-0.025 (0.029)	0.758, not significant
LCOM	51.8	100	74.1	0.0	0.006	0.000 (0.000)	0.842, not significant
CS	53.8	86.0	78.4	14.0	0.057	-0.001 (0.000)	13.817, p < 0.0001

goodness of fit (R^2) is almost half that of the proposed model. All other measures (except for class size) were found to be statistically insignificant predictors, while class size has both a lower R^2 and a lower standardized coefficient as shown in Table 6. Forward and backward logistic regression results, indicating as important variables the proposed measure and the class size are shown in Table 7. (The CBO metric is also considered significant, but the corresponding impact measured by the change in -2 Log Likelihood is significantly lower than that of the other two measures).

As a result, it could be concluded, that in case of a low estimation by the development team for the number of propagated changes, the accuracy of the change history model could be considered sufficient. However, in case of more "volatile" systems, the use of a model that considers coupling, inheritance, and dependencies as axes of change between classes can offer increased accuracy in the prediction of changes. According to the results for JFlex and JMol, a multivariate regression model with the proposed measure and class size as covariates, would lead to a higher overall accuracy. It should, however, be mentioned that the accuracy should be viewed bearing in mind that models considering only factors within the bounds of the software system will have by default a limited accuracy since it has been recognized [9] that other conceptual axes beyond the software system (such as common requirements) can cause ripple effects.

5 Implementation

One of the goals of analyzing the probabilities of change in a system is to enable the automation of the process by means of an appropriate parser and analyzer. To this end, a Java program has been developed that parses the complete hierarchy of directories that include the project under study in order to reveal the static structure of the object-oriented design. (An XML file that includes tags for annotating each class with the required information concerning associations and inheritance relationships can also be used as input.) Next, the program applies the aforementioned methodology to calculate each class' probability of change. To locate changes in the code between successive generations, any differencing tool can be used. The user should manually classify the changes as internal or ripple effects. This part of the process has not been automated and relies on engineering experience. However, the exact type of change is of no interest; only whether the change is an internal one or a change that propagated from another class.

TABLE 6
Standardized Logistic Regression Coefficients for JMol

Measure	*Standardized coefficient*
proposed	0.327
history	0.239
CS	0.238

The probabilistic evaluation application consists mainly of three parts: 1) a bytecode parser, 2) an XML generator and parser, and 3) a probability calculator.

The bytecode parser is able to analyze the contents of any object-oriented application in Java residing in a hierarchy of directories and provides general class information required for the calculation of the probabilities. For this purpose, the ASM API is used, which is a Java bytecode manipulation framework [4]. More specifically, it provides information relevant to the relationship of each class to the other classes of the system. The extracted information for each class consists of:

- superclass name,
- implemented interfaces,
- field types (global declarations),
- constructors with parameter types,
- methods with return type and parameter types, and
- method/constructor invocations.

Because the dependence on library classes (such as Java libraries or external API's) is not considered as a source of changes, a filtering process follows the source parsing, which removes from the parsed information all classes which are irrelevant to the system and all duplicate class values. Finally, the parameter types of the constructors and methods as well as direct instances are merged in a list, which includes the references of a class to other classes of the system.

TABLE 7
Results of (a) Forward and (b) Backward Logistic Regression Analysis for JMol

	Variables	Change in –2 Log Likelihood (if variable is removed)	Significance of the Change
Step 1	proposed	55.195	0.000
Step 2	proposed	70.204	0.000
	CS	43.502	0.000
Step 3	proposed	67.464	0.000
	CBO	9.269	0.002
	CS	51.902	0.000

(a)

	Variables	Change in –2 Log Likelihood (if variable is removed)	Significance of the Change
Step 1	proposed	23.314	0.000
	history	0.737	0.391
	CBO	9.675	0.002
	CS	52.534	0.000
Step 2	proposed	67.464	0.000
	CBO	9.269	0.002
	CS	51.902	0.000

(b)

The XML generator, after a hierarchy of directories containing Java classes has been parsed, produces an XML file containing class names and their corresponding axes of change, which constitute the required information for the calculation of probabilities. The same kind of XML files can also be used as primary input to the program. The axis information for each class is represented in XML with the following format:

```
<class>
    <name>class-name</name>
    <axis>
          <description>axis-
              description</description>
          <to>class that axis refers to</to>
    </axis>
</class>
```

Each axis of a class is represented in an axis tag, receiving as values the strings "reference axis," "inheritance axis," which can appear multiple times for each class, and "internal axis" which can appear only once. For the "internal axis," the <to> tag has the name of the class itself as value.

The order according to which probabilities are estimated depends on the level of each class in an inheritance tree. For any inheritance relationships, probabilities are calculated starting from the classes higher in the hierarchy since the probability of a superclass is required in order to extract the probability of a subclass.

One issue that requires nontrivial handling concerns mutually dependent classes. For example, consider the hypothetical design in Fig. 5 where classes $C1$ and $C2$ are abstract.

In the above system, the probabilities of classes $C3$ and $C4$ have a mutual or cyclic dependence on each other. The probability for class $C3$ is given by:

$$P(C3) = P(C3 : extension\,axis \cup C3 : reference\,axis \cup C3 : internal\,axis),$$
$$\text{where } P(C3 : reference\ axis) = P(C3|C4) \cdot P(C4).$$

The probability for class $C4$ is extracted in a similar manner. Even if the conditional probabilities have a known constant value k, the above probabilities lead always to a set of first-order equations of the form:

$$P(C3) = a + (1 - a) \cdot k \cdot P(C4),$$
$$P(C4) = b + (1 - b) \cdot k \cdot P(C3)$$

that should be solved in order to calculate $P(C3)$ and $P(C4)$. a and b are coefficients resulting from all other probabilities. For a worst case analysis ($k = 1$), no matter what the values of a and b are the solution to such a system is always equal to 1 ($P(C3) = P(C4) = 1$), as a result of the cyclic dependency between the joint probabilities. For other values of k, the system solution would lead to the actual values.

In order to cope with this problem without adding to the complexity of the software, our implementation initially considers the two or more classes as not associated (their association is temporarily broken) and proceeds to the estimation of probabilities as already described. Once the probabilities are extracted, the association is restored and the probabilities for each class are calculated again as the joint probabilities of their prior value and that of the associated classes.

Concerning the probability calculation, the first step is to identify any cyclic dependencies between the classes. In order to find cycles in the relationships between classes, a tree is built: For each class, the classes involved in each axis are added as children nodes and this process is repeated recursively for the children nodes as well. While this tree is being built, if a leaf is the same class as the root class, a cycle is identified. In order to avoid infinite recursions, if a class node already exists in the tree-path to the root or in a

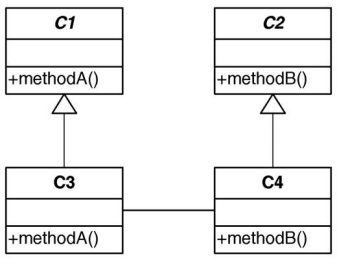


Fig. 5. Cyclic dependencies of probabilities.

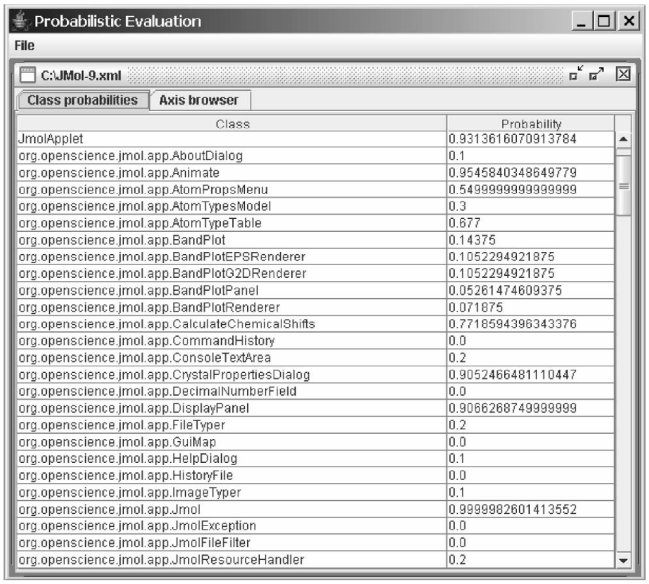


Fig. 6. Sample screenshot for the application showing calculated probability values.

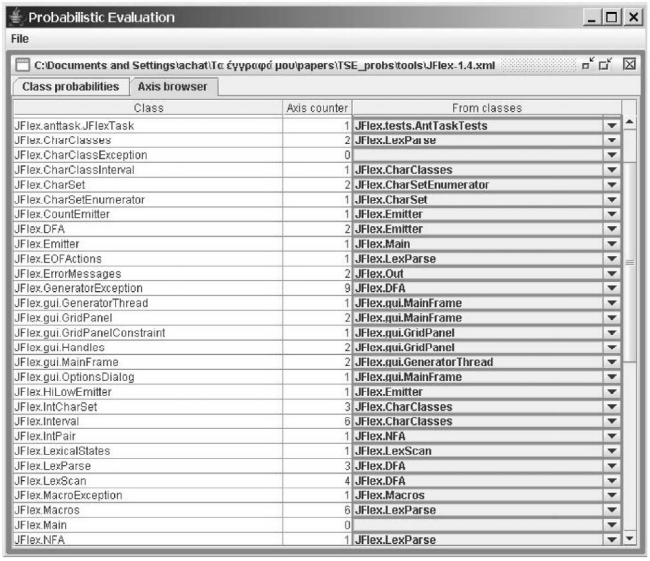


Fig. 7. Sample screenshot for the application showing axes of other classes in which each class is referred.

previously found cycle, then this node is not checked again. This algorithm captures cycles which have one of the following formats: $A \rightarrow B \rightarrow C \rightarrow A$ or $A \leftrightarrow B$.

At the end of this process, all classes are grouped into four categories:

1. Classes that do not participate in cycles.
2. Classes that directly participate in cycles.
3. Classes that do not participate in cycles but have an axis to a class that participates in a cycle.
4. Classes that do not participate in cycles but have an axis to a class of category 3.

First, the probabilities of the class category 1 are calculated, whose values are not dependent on any unknown probabilities. For the class category 2, there are two substeps in the calculation of the probabilities: In the first step, the probabilities of the classes are calculated taking into account only those axes that are not involved in any cycle. In the second step, the calculation of probabilities for the cycle dependent axes proceeds assuming that all classes receive as initial probabilities the values of the first step. Finally, the probabilities of the class category 3 and then of 4 are calculated since all other probabilities are already known.

The user can easily interact with the GUI in order to change the internal probability of change for each class from the default 0.5 to any other value. This task can also be performed by loading a text file including the description of each class and the corresponding internal probability of change, extracted for example from past data. A sample screenshot of the developed software with estimated probabilities is shown in Fig. 6. As already mentioned the tool provides for each class, the number of axes of other classes, in which it is involved, in order to identify modules that can be a source of change (Fig. 7).

The tool as well as all necessary data for the analysis of all versions of JFlex and JMol can be downloaded from [32].

6 Threats to Validity—Limitations

6.1 External and Internal Threats

Although the assumptions in the model have already been mentioned, we list explicitly the most important threats to internal and external validity.

As threats to internal validity we consider those factors that may cause interferences regarding the relationship between the dependent and the independent variable. The proposed prediction model might have omitted other important variables that can serve as predictors. Moreover, the assumptions on the independence between axes of change affect the accuracy of the probabilistic model. These threats are valid, but the goal was to investigate whether the proposed model (which considers both change history and the internal structure of the system) leads to improved accuracy. For this reason, only univariate models have been studied.

As threats to external validity, we consider those factors that may limit the possibility to generalize our findings beyond the two case studies. One threat that is valid and cannot be excluded until extensive empirical results are collected is that the two case studies will reflect the characteristics from their specific domain. However, the analysis does not emphasize on the type of changes, but rather on their number and classification as internal or propagated. Moreover, although the collection of data was performed by analyzing the code and without relying on change logs, poor documentation [10] might affect the analysis significantly: For example, undocumented dead code or conceptual dependencies between classes that are not explicitly listed will affect the number of propagation axes and, therefore, the calculated probabilities.

6.2 Limitations

The main limitation of the approach is that, in any case, predicting changes in a future generation is an ambitious goal since many factors that determine whether a class will be changed or not are not code related. As a result, the

correlation between the calculated probability values and the actual outcome can hardly reach very high values. Moreover, the proposed analysis is a heavyweight approach with regard to the collection of data (previous versions of a system have to be analyzed to acquire internal probability values). This could create scalability problems for large systems. In addition, it cannot be applied at early stages of the development process (e.g., at the design level). Finally, although the identification of changes can be computer-aided, their classification as propagated or not requires human intervention, at least with the current tools.

7 Related Work

Previous attempts to assess the changeability of object-oriented designs include a controlled experiment for comparing responsibility-driven and control-oriented alternatives with regard to required change effort [2]. In [8], a change impact model has been proposed for changeability assessment with primary goal to investigate the relationship between existing design metrics (e.g., Weighted Methods per Class) and the impact of change. However, even if it is useful to know which classes would be impacted from a given change, one has to know the actual changes that occurred in a system, in order to assess the probability of change for a certain class. The relationship between metrics and maintenance effort has also been studied in [24]. In [23], a set of algorithms that determine what classes are affected by a given change is proposed. The methodology represents a systems as a set of data dependency graphs, which is a reasonable and effective approach for object-oriented designs. However, as in any change impact model, reports about the potential impact of a given change can be generated only after the user explicitly specifies the changes.

Briand et al. [7] empirically investigated using a commercial OO system, whether coupling measures are related to ripple effects. The aim is to rank classes according to their probability of containing ripple effects, while the approach proposed in this paper aims at identifying classes that are highly probable to change in a future generation, regardless of whether the change is internal or due to a ripple effect. An advantage of using coupling measures is that they are inherently related to ripple effects since common changes are usually due to relationships between classes. However, ripple effect-prone classes cannot be used for predicting whether they will change in a future release since changes originating in the class itself are neglected. Moreover, in order to perform further empirical evaluation on the use of coupling for impact analysis, one would require change logs listing all classes affected by a given change, something which can hardly be found in practice.

A tool for supporting maintenance in legacy systems has been proposed in [9]. It can assist the maintainer to locate components where changes might propagate by aiding in the identification of data/control flow dependencies. The advantage is that the model also considers conceptual dependencies, something which certainly increases the accuracy; however, components that participate in the same concept must be manually specified by the user. The work of Yau and Collofello [36] and Black [6] on the computation of ripple effects deals essentially with the definition of a logical stability measure. Though, this measure reflects the number of ways by which variable values can propagate to the same or other modules (in procedural programs) and is not a measure of probability. However, the need for a metric system to quantitatively measure the impact of possible changes as well as to estimate the possibility of future changes among with automation tools has not been addressed. In a precursor of this work, we performed a worst case analysis for the evaluation of probabilities [34], where it was assumed that all changes eventually propagate to other classes. According to [27], our approach belongs to predictive analysis, that is, software metrics are used for analysis *before* the evolution has occurred, mainly to assess which parts are likely to be evolved (evolution-prone parts). To this end, release histories of the software can be investigated, as it has been performed, for example, by means of visualization [14]. In that work, a third dimension has been used to visualize the software release history, while color associated to certain module attributes (such as code size or version number) might be helpful in identifying change-prone parts of the system. Recently, Girba et al. [17] proposed an approach to summarize the changes in the history of a system that can offer a solid basis for starting a reverse engineering effort. The methodology consists in identifying the classes that were changed the most in the recent history and at the same time checking whether the same classes are among the most changed ones in the successive versions. However, as change, only the addition or removal of methods is considered. Arisholm et al. [3] investigate the use of dynamic coupling measures as indicators of change proneness. Their approach is based on correlating the number of changes to each class (a continuous variable which represents change proneness) with dynamic coupling measures and other class-level size and static coupling measures. Consequently, it cannot be considered as a prediction model since no attempt is made to correlate the proposed measures with changes/no changes (which is a dichotomous variable) in the next generation. In addition, requirement changes have been factored out since they are not driven by design characteristics.

8 Future Work

As already mentioned, any model considering only the static structure of the system itself ignores other conceptual axes that generate common changes and thus affect any probability measure. Since conceptual dependencies cannot be revealed by static code analysis [9], prior knowledge of connections in the application domain or dependencies due to data flow within library functions (which cannot be recovered due to the unavailability of the source code) could be manually added to the model.

Another possible contribution has to do with the fact that the internal probability values that have been extracted from the history of each class have been treated as data points that present no internal structure. An alternative to building such a prediction model would be to perform time series analysis although there is no distribution into equally

spaced time intervals. Such a fitting of time series models could possibly offer improved forecasting capabilities.

One of the implementation issues that remains open for further improvement of the accuracy of the results is the algebraic solution of the probabilities associated with classes that have a cyclic dependency. Such a solution, although possible by means of a matrix manipulation package, has not been implemented so far since the incurred complexity was beyond the scope of this research.

Finally, we believe that by further analyzing existing large software projects, much knowledge could be gained by investigating the relationship of change-prone classes to other structural characteristics or even factors such as the number of different authors, age of each piece of code, experience and so on.

9 Conclusions

Acknowledging the importance of change handling in the software development process, a methodology for quantifying the change proneness of an object-oriented design has been proposed. The rationale behind this approach is that in a well-designed software system, feature enhancement or corrective maintenance should affect a limited amount of existing code. The goal is to quantify this aspect of quality by assessing the probability that each class will change in a future generation. Apart from the probability that a change occurs in a class itself, changes can propagate through so-called axes of change, affecting the overall probability value.

Statistical analysis has shown a correlation between the extracted probabilities and the actual changes in a system. The proposed approach improves the prediction accuracy over a simple model that relies only on past data, while most other structural metrics (except for class size) have been proved to be statistically insignificant predictors. The advantage lies in that the probability values are extracted considering both the change history of a design as well as its structural characteristics.

The proposed methodology has been automated and can be applied to any object-oriented software system in order to evaluate the evolution of a design through successive generations and to identify "bad" classes that can cause changes to the rest of the system.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable contributions to the research methodology presented in this paper.

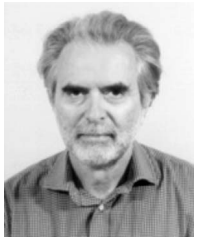
References

[1] A.V. Aho and J.D. Ullman, *Foundations of Computer Science,* third ed. CS Press, 1995.
[2] E. Arisholm, D.I. Sjøberg, and M. Jorgensen, "Assessing the Changeability of Two Object-Oriented Design Alternatives—A Controlled Experiment," *Empirical Software Eng.,* vol. 6, pp. 231-277, 2001.
[3] E. Arisholm, L.C. Briand, and A. Føyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Trans. Software Eng.,* vol 30, no. 8, pp. 491-506, Aug. 2004.
[4] ASM Home Page, http://asm.objectweb.org/, 2004.
[5] V.R. Basili, L.C. Briand, and W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.,* vol. 22, no. 10, pp. 751-761, Oct. 1996.
[6] S. Black, "Computing Ripple Effect for Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice,* vol. 13, pp. 263-279, 2001.
[7] L.C. Briand, J. Wüst, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM),* pp. 475-482, Aug./Sept. 1999.
[8] M.A. Chaumun, H. Kabaili, R.K. Keller, and F. Lustman, "A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems," *Science of Computer Programming,* vol. 45, nos. 2-3, pp. 155-174, Nov. 2002.
[9] K. Chen and V. Rajlich, "RIPPLES: Tool for Change in Legacy Software," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM),* pp. 230-239, Nov. 2001.
[10] K. Chen, S.R. Schach, L. Yu, J. Offutt, and G.Z. Heller, "Open-Source Change Logs," *Empirical Software Eng.,* vol. 9, pp. 197-210, 2004.
[11] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.
[12] A.L. Edwards, *An Introduction to Linear Regression and Correlation.* San Francisco, Calif.: W.H. Freeman, pp. 72-76, 1976.
[13] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach.* Boston, Mass.: Int'l Thompson Publishing, 1997.
[14] H. Gall, M. Jazayeri, and C. Riva, "Visualizing Software Release Histories: The Use of Color and Third Dimension," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM),* pp. 99-108, Aug./Sept. 1999.
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[16] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering,* second ed. Prentice Hall, 2003.
[17] T. Gîrba, S. Ducasse, and M. Lanza, "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes," *Proc. Int'l Conf. Software Maintenance (ICSM '04),* Sept. 2004.
[18] F.M. Haney, "Module Connection Analysis—A Tool for Scheduling of Software Debugging Activities," *Proc. AFIPS Fall Joint Computer Conf.,* pp. 173-179, 1972.
[19] A. Hunt and D. Thomas, "OO in One Sentence: Keep it DRY, Shy and Tell the Other Guy," *IEEE Software,* vol. 21, no. 3, pp. 101-103, May/June 2004.
[20] JFlex—The Fast Scanner Generator for Java, http://jflex.de/, 2004.
[21] JMol Open Molecule Viewer, http://jmol.sourceforge.net/, 2004.
[22] C. Kirsopp, M. Shepperd, and S. Webster, "An Empirical Study into the Use of Measurement to Support OO Design Evaluation," *Proc. Sixth IEEE Int'l Symp. Software Metrics,* pp. 230-241, Nov. 1999.
[23] L. Li and A.J. Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM),* Nov. 1996.
[24] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems and Software,* vol. 23, no. 2, pp. 111-122, Nov. 1993.
[25] R.C. Martin, *Agile Software Development: Principles, Patterns and Practices.* Prentice Hall, 2003.
[26] S. Menard, *Applied Logistic Regression Analysis.* Thousand Oaks, Calif.: Sage Publications, 2001.
[27] T. Mens and S. Demeyer, "Future Trends in Software Evolution Metrics," *Proc. Fourth Int'l Workshop Principles of Software Evolution (IWPSE),* pp. 83-86, 2001.
[28] B. Meyer, *Object-Oriented Software Construction,* second ed. Prentice Hall, 1997.
[29] A. Papoulis, *Probability, Random Variables, and Stochastic Processes,* second ed. McGraw-Hill, 1984.
[30] D.L. Parnas, "Some Software Engineering Principles," *Structured Analysis and Design,* pp. 237-247, 1978.
[31] C.-Y.J. Peng, K.L. Lee, and G.M. Ingersoll, "An Introduction to Logistic Regression Analysis and Reporting," *The J. Educational Research,* vol. 96, no. 1, pp. 3-14, Sept. 2002.
[32] Probabilistic Evaluation Framework, http://java.uom.gr/nikos/probabilistic-evaluation.html, 2004.
[33] A.J. Riel, *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[34] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and I. Deligiannis, "Probabilistic Evaluation of Object-Oriented Systems," *Proc. 10th Int'l Symp. Software Metrics (METRICS 2004)*, Sept. 2004.
[35] F.G. Wilkie and B.A. Kitchenham, "Coupling Measures and Change Ripples in C++ Application Software," *J. Systems and Software*, vol. 52, pp. 157-164, June 2000.
[36] S.S. Yau, J.S. Collofello, and T.M. McGregor, "Ripple Effect Analysis of Software Maintenance," *Proc. Computer Software and Applications Conf. (COMPSAC '78)*, pp. 60-65, 1978.

Nikolaos Tsantalis received the BS degree in applied informatics from the University of Macedonia in 2004. He is a masters student in the Department of Applied Informatics at the University of Macedonia, Greece. His research interests are in the areas of design patterns and object-oriented quality metrics.

Alexander Chatzigeorgiou received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. He is a lecturer in software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. From 1997 to 1999, he was with Intracom S.A. Greece as a telecommunications software designer. His research interests are in object-oriented design metrics, pattern detection, and low-power hardware/software design. He is a member of the IEEE Computer Society.



George Stephanides received the PhD degree in applied mathematics from the University of Macedonia. He is an assistant professor in the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. His current research and development activities are in the applications of mathematical programming, security and cryptography, and application specific software. He is a member of the IEEE Computer Society.