An analogy-based approach for predicting design stability of Java classes

David Grosser, Houari A. Sahraoui and Petko Valtchev
DIRO, Université de Montréal,
CP 6128, succ Centre-ville,
Montréal QC H3C 3J7, Canada
{grosserd, sahraouh, valtchev}@iro.umontreal.ca

Abstract

Predicting stability in object-oriented (OO) software, i.e., the ease with which a software item evolves while preserving its design, is a key feature for software maintenance. In fact, a well designed OO software must be able to evolve without violating the compatibility among versions, provided that no major requirement reshuffling occurs. Stability, like most quality factors, is a complex phenomenon and its prediction is a real challenge. In this paper, we present an approach which relies on the case-based reasoning (CBR) paradigm and thus overcomes the handicap of insufficient theoretical knowledge on stability. The approach explores structural similarities between classes, expressed as software metrics, to guess their chances of becoming unstable. In addition, our stability model binds its value to the impact of changing requirements, i.e., the degree of class responsibilities increase between versions, quantified as the stress factor. As a result, the prediction mechanism favors the stability values for classes having strong structural analogies with a given test class as well as a similar stress impact. Our predictive model is applied on a testbed made up of the classes from four major version of the Java API.

Keywords: Analogy based estimation, learning, maintainability, predictive model, software quality prediction

1 Introduction

The object-oriented (OO) paradigm has now reached maturity, with a huge amount of software written with OO programming languages currently available. Software maintenance, under evolving requirements and continuous error detection, often leads to the development of several successive version of a system, whereby the preservation of downward compatibility among versions is a highly desirable feature.

Unfortunately, as software products are growing more sophisticated, the writing of newer versions has become complex and time consuming activity. For example, Pressman estimated that 60% of the total effort in software development is spent to maintenance [25], whereby 80% of this amount is spent directly or indirectly on software evolution (adaptive and perfective maintenance) [24]. Despite some clear benefits of object-orientation for maintenance, OO systems are not exempt from this rule. In this respect, it has become important to manage class stability along the version sequence, in particular, by applying tools for stability prediction, e.g., based on the symptomatic detection of potential instabilities during the design phase.

In the context of our investigation, we define stability as the capability of a software system or component to evolve while preserving its design. In the present work, we restrict the part of design considered for preservation to the class interfaces, i.e., set of attributes and methods. This choice was motivated by the observation that any change in class interfaces can trigger a large amount of side effects due to the dependencies (coupling) between classes.

Classical approaches to stability prediction, which is a hard problem, perform some form of inductive inference starting from datasets of classes with known stability levels and looking for typical features that discriminate stable classes from unstable ones. However, most of the effective methods for predictive model construction are based on the implicit hypothesis that the available samples are representative, which is rather strong. In fact, unlike other experimental fields within disciplines such as medicine, sociology, or statistics, where free access to large repositories of consensual data is granted, in software engineering there are no such sources of data. To make the matters worse, the very nature of software and of the related software process makes the constitution of such consensual testbeds unreaistic. Indeed, among the few companies that systematically collect quality-related information, even fewer are those eager to publish such information (usually considered confidential), and, whenever they decide to proceed to publication, only the resulting predictive models are usually pre-

sented, keeping the initial data aside. Consequently, the models that are proposed in the literature are most of the time hard to generalize from and of poor reusability.

In our investigation, we study an alternative approach for building predictive models which we consider as more appropriate to the particular context of software. The approach implements a similarity-based comparison principle: the stability of a given software item is estimated from the recorded stability of a set of other items that have been recognized as the most similar to that item among a larger set of items stored in a database (quality factors whose prediction by means of similarity-based techniques include reusability [10], correctability [2], and reliability [13]). A direct application of the similarity-based approach, also known as case-based reasoning (CBR) has been examined at an earlier step of our study [15], whereby we provided some empirical evidence to illustrate its advantages with respect to a classical inductive learning technique as the decision tree inference.

In this paper, we propose a more sophisticated model which, except for the more flexible learning mechanisms, includes a key factor that has been previously neglected, i.e., the stress a class experiences between two software versions. The stress, usually the result of a major change in the requirements, has been defined as the degree of increase in the responsibilities of the class itself or the classes to which it is related. The role of the stress as a measure is dual to that of stability since the interface of a class in a version $i + 1$ compared to that in the version i is made up of three parts: stable members, unstable members, and newly introduced members reflecting the augmented responsibilities. The paper proposes a formalization of the stress factor as well as a set of strategies for the integration of the stress factor in the stability prediction. An experimental comparison of those strategies is presented as well, involving classes from four major versions of the standard Java API.

The paper is organized as follows. First, existing work on the quality prediction is presented in section 2, with a focus on stability. Then, our approach is described in section 3. Section 4 follows the experimental evaluation of the similarity-based approach and its variants with respect to the utilization of the stress-related information.

2 Stability prediction for Java classes

In the following we present key elements of our approach such as the software quality prediction principles, stability models and the considerations related to the choice of Java.

2.1 Quality prediction for object-oriented software

Most of the software quality factors, e.g., maintainability, reusability, reliability, etc., admit no direct measurement *a priori*. This fact motivated a wide range of studies aimed at predicting these factors from some measurable software characteristics such as coupling, cohesion, and size [12]. Within this trend, a large number of metrics, in particular, OO metrics, have been proposed in the literature (see [5, 4, 8, 20]).

Early work on effective construction of predictive models for quality relied on classical techniques from statistics such as least squares or robust regression (see [6] for a comparative study). Globally, all these techniques assume particular data distributions or specific dependencies between quality factors and metrics. Unfortunately, both types of hypotheses are difficult to validate in the software domain.

To avoid this pitfall, many researchers, while looking for alternative, adopted techniques inspired by the machine learning field. Thus, in early 90s, predictive models for reliability [18], for size [16] or for development effort [29, 28] have been proposed. However, these models suffered from their "black box" effect which eventually limited their scope. Another trend brought in ideas from the neuro-fuzzy paradigm in machine learning, although they remained of a limited impact on the resolution of the black box problem [17, 19].

Recently, the application of another powerful learning paradigm, the decision trees induction, has been studied [21]. Except for its inherent limitation to classification problems, the technique also requires additional effort in determining the threshold values for variable discretization. Later on, this specific problem has been tackled with methods from the fuzzy logic field [14, 26]. Despite the undeniable progress on the figuring out key difficulties in prediction, there is still no valid solution to the problem of lack of representativity in the available datasets.

Before representing an approach that avoids the representativity trap, we first define what we consider to be the stability of a class, together with a key factor that was previously ignored, the stress of a class.

2.2 Defining the stability

All its life-cycle along, but especially during operation time, a software undergoes various changes, most of the time triggered by error detection or environment changes, but also due to evolution in the requirements. As a result, the behavior of the software is gradually deteriorating along the increase in modifications and this quality slump may go as far as the entire software becoming unpredictable [23]. Consequently, we claim that the software that is intended to

last must be designed in a way that helps them withstand such negative impact, i.e., remain stable in spite of requirements evolution. Unfortunately, as reported in [11], the relative awareness about this important topic within the community, has not yet led to the broad adoption of stability-oriented design methods.

2.2.1 Stability definition from the literature

Much work has been dedicated to the clarification of the stability concept. For instance, in [22], design rules are proposed which ensure the stability of large software systems by rigorous dependency management and abstraction. Similarly, [11] describes a design model that distinguishes between a kernel layer (*Enduring Business Themes and Business Objects*) and peripheral layer (*Industrial Objects*) in the system whereby the former attracts the major part of the stability enforcement effort. Indeed, it is suggested to keep the kernel stable while keeping the peripheral parts open to arbitrary changes. However, neither of the above studies suggests an effective method for stability evaluation, whence their respective impact on stability of the target systems is hard to estimate. Other researchers have put the emphasis on stability evaluation and proposed effective methods for the problem [9, 3], although the scope of these methods remains limited to software frameworks.

Another way to achieve the stability of software systems is to apply some refactoring. From this perspective, the stability enforcement can be seen as a maintenance activity. The main idea is to develop techniques that evaluate/predict the stability with the perspective of deciding whether a major refactoring is needed or not. The goal of the refactoring is then to reduce the implementation cost of the future requirements. Our current work is founded by this belief. It deals with stability in its general form, with no hypothesis about the nature of the target system. In this paper, we study the stability assessment at the class level and between two successive versions.

2.2.2 Basic stability model

The key assumption behind our stability model is that a class is stable whenever its interface remains unchanged between versions. Let c be a class. $I(c_i)$ is the interface of c in version i (public and protected, local and inherited methods). The level of stability c can be measured by comparing $I(c_i)$ to $I(c_{i+1})$ (following version). It represents the percentage of $I(c_i)$ that is included in $I(c_{i+1})$. Formally

$$NS(c_{i\rightarrow i+1}) = \frac{\#\cap(c_i, c_{i+1})}{\#I(c_i)}$$

where

$$\cap(c_i, c_{i+1}) = I(c_i) \cap I(c_{i+1}).$$

Our hypothesis is that the stability of a class interface depends from the design (structure) of the class and the stress induced by the implementation of new requirements between the two versions. The estimation model will take the form of a function f that takes as input a set of structural metrics $(m_1(c_i), m_2(c_i), \ldots, m_n(c_i))$ and an estimation of the stress $St(c_{i\rightarrow i+1})$ and produces as output an estimation of the level of stability $ENS(c_{i\rightarrow i+1})$. Formally

$$ENS(c_{i\rightarrow i+1}) = f(m_1(c_i), \ldots, m_n(c_i), St(c_{i\rightarrow i+1})).$$

In our work we considered 14 structural metrics covering the coupling, cohesion, size/complexity and inheritance (see figure 2).

2.2.3 Stress factors

The stress is hard to estimate since it is difficult to quantify *a priori* at what level a class will be concerned by a set of new requirements. Nevertheless, we have identified the following types of modifications that may have an impact on the stability of a class and therefore will be considered as components of the complex stress factor:

- local modifications of the class c_i, e.g., triggered by the definition of new methods,
- modifications in the ancestor classes in the generalization hierarchy: local modifications of ancestors or in the structure of the hierarchy above the class c_i,
- modifications in the descendant classes in the generalization hierarchy: local modifications of ancestors or in the structure of the hierarchy below the class c_i,
- modifications in the classes depending on c_i or in the classes which c_i depends on,

For practical uses, the stress can be approximated *a posteriori* by the percentage of added methods in the considered class. Formally

$$St(c_{i\rightarrow i+1}) = \frac{\#\Delta(c_{i+1})}{\#I(c_{i+1})}$$

where

$$\Delta(c_{i+1}) = I(c_{i+1}) - I(c_i).$$

In the following section, we describe our predictive model for stability which integrates structural metrics and stress estimations in various prediction strategies derived from a classical case-based reasoner.

2.3 Java API classes

Nowadays, the largest part of OO software is written in Java, whence our choice of this programming language. In the following paragraphs, we recall some basic facts from the philosophy of the language as well as the major steps in the evolution of Java.

2.3.1 Language evolution

Java was initially designed by Sun Microsystems[1], in the early 90s, with its first public release dating back to 1996 (JDK version 1.0). In the first version, only a limited number of standard classes were included (less than 200). Since then, several substantial reorganizations of the language have been undertaken, aimed at two main purposes: on the one hand, correct errors, and on the other hand, extend the functionalities of the language. Basically, most of the changes consisted in improving the development tools, changing standard classes in the API, and adding new packages and classes.

For instance, the graphical library `Swing`, initially included in the releases as an external library, has been included in the standard API in the version 3 of Java (API 1.3). Another large-scale modifications include the introduction of inner and nested classes as well as the profound restructuring of the event-handling mechanisms in the version 2 of Java (JDK 1.2).

Since the very beginning, the portability of the software written in Java has been a major concern and, hence, a key factor in the success of the language. As a matter of fact, Java is the most portable language that is currently used in the software industry. However, other problems, such as lack of downward compatibility between API versions hampers the portability, since for running the same software on two different platforms, one still needs having the same version of Java running on them.

2.3.2 Compatibility between versions

In spite of the significant effort towards version compatibility, far too many systems have to be rewritten, at least partly, to enforce compatibility with the latest versions of the language. To ease the problem of software rewriting for compatibility, a specific mechanism has been implemented in Java. Thus, classes or methods that have been substantially changed are labeled as *deprecated*. At compilation time, the calls to deprecated items trigger warning messages from JDK, so that a developer could be informed about the changes between versions.

2.3.3 Aims of our study

The experimental study presented in section 4 examines the stability of standard Java classes, in systematic way. Four major versions of the language are considered, from the first one(API 1.0.2) to the most recent one that was available at the period of the study (API 1.3.2)[2], thus leading to three transitions. A set of metrics has been measured on the classes of each version, whereby the deprecated classes and methods have been considered as unstable in order to increase the precision of the prediction.

[1] http://java.sun.com

[2] The most recent stable version, API 1.4.1 has been released at later step.

3 Stability prediction using case-based reasoning

CBR emerged as an approach towards problem resolution in domains where little is known about key processes and their interdependencies (also called *weak theory* domains). Thus, instead of relying on a complete domain theory, CBR only explores the expert knowledge encoded in the available documented resolutions of past cases. Technically speaking, the overall reasoning process behind CBR consists in solving new problems by retrieving and adapting the solutions to similar problems that have occurred in the past. These problems are represented and stored as individual cases in a case-base [1]. The choice of the most appropriate case(s) for reuse the solution is driven by analogies in case descriptions. Analogies are detected by a matching mechanism which typically relies on a similarity assessment function.

As there is a lack of knowledge about software evolution, we believe that CBR is a suitable approach to the software stability prediction problem. Thus, we hypothesize that two software items (Java classes in the experimentation) which show same or similar characteristics will also evolve in a similar way.

In the context of CBR, three important issues need to be addressed: case representation, case retrieval and solution reuse.

3.1 Case representation

The representation problem in CBR is primarily the problem of deciding what to store in a case, finding an appropriate structure for describing case contents and deciding how the case base should be organized and indexed for effective retrieval and matching. As described in section 2, here, the relevant characteristics of software items are expressed as a set of class structural metrics completed by a stress factor. For the current study, we regard the stability as a real variable for which range is between 0 (absolute instability of the software component) and 1 (absolute stability). We use a structured representation formalism, of an object-attribute-value type, for classes. Thus, a software class is represented as a structured object in our model, with object attributes representing software metrics and stress. The result for a given metric on a class is stored as the value of the respective attribute in the object representing the class. In addition, the case representation includes an attribute modeling the stability indicator. Thus, each class is represented as a n-tuple $c = (name, m_1, m_2, \ldots, m_k, s, t)$,

where $name$ is the class identity, m_i are metric values, s is the stress value and t is the stability indicator (see figure 1 for symbolic names of attributes).

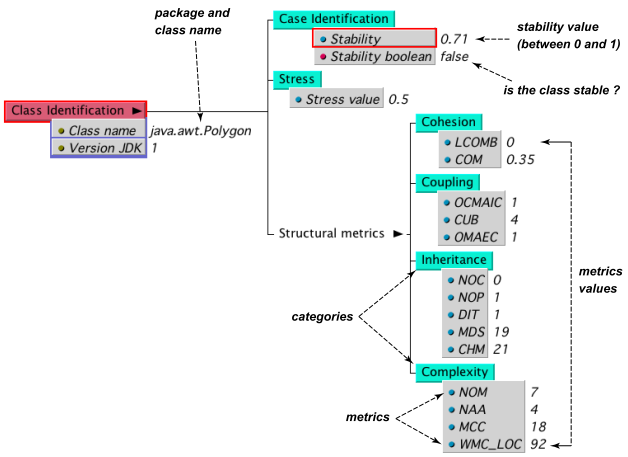

Figure 1. Example of structured Case Representation

In the above example, the class *Polygon* from the package *java.awt* (graphics library) has 50% of its interface in the version 1.1 newly introduced with respect to the version 1.0.2 (attribute *Stress value*). Similarly, 70% of its public interface changed between version 1.0.2 to version 1.1 (attribute *Stability*). The value of *Stability* in a class is the target of the prediction process, i.e., given a previously unseen class c with unknown stability indicator, a prediction value will be extracted from the cases in the case base.

3.2 Case retrieval

The aim is to find the set of known cases that match the new case at best, i.e., the *BestMatch* set. In our case, this amounts to look for most similar classes (in terms of our similarity assessment function). These classes, further called the *nearest neighbors*, lay within a particular neighborhood of the new case in the description space. Thus, the retrieve task takes a (possibly partial) problem description, and ends with a complete *BestMatch* set.

The retrieval algorithm performs a complete search through the case base whereby each case in the base is compared to the new case. Depending on the similarity value, the current known case may be inserted into the set of current best matches. In order to increase the chances of a correct prediction, the size of set *BestMatch* may be extended. The exact value of |*BestMatch*| is a parameter of the algorithm, the approach being known as the *k-nearest neighbors* learning (*k-NN*). For our study, we experimented various size of the best-match set.

3.3 Similarity issues

With the above representation, a component may be seen as a point in the n-dimensional Euclidean space where coordinates correspond to software metrics m_j and to stress. Thus, classical distance measures can be applied to the assessment of inter-component similarity, for example the metrics of the Minkowski family (Euclidean, Manhattan distance, etc.). Our own measure, derived from the Manhattan distance, is basically a linear combination of the pointwise differences (absolute values) between the vectors representing a pair of components.

The measure is defined on two levels: local level, and component or global level.

Local similarity factor For each individual factor, either structural metric m_j or stress, the local disimilarity between two components c and c' with respect ot this factor, is defined as the normalized absolute arithmetic difference of the respective values. Thus, for a metric m_j, the function considers the values $m_j(c)$ and $m_j(c')$, furhter denoted $dis(m_j(c), m_j(c'))$. Formally, the function is computed as follows:

$$dis(m_j(c), m_j(c')) = \frac{|m_j(c) - m_j(c')|}{|\mathrm{dom}(m_j)|} \qquad (1)$$

where $|\mathrm{dom}(m_j)|$ stands for the maximal difference of two values v_1, v_2 in $\mathrm{dom}(m_j)$. Clearly, for any pair of classes c and c', the value $dis(m_j(c), m_j(c'))$ is in [0 1], with 0 meaning perfect match, i.e., identical values on each metric, and 1 meaning complete mismatch. The reasoning with the stress estimation value is similar.

Global similarity measure The contributions of each metric m_j and of the stress estimation on the local level, are combined into a unique value reflecting the overall disimilarity of both components. For this purpose, we use a normalized linear combination. Thus, given a pair of classes from version i of the system, c_i and c'_i, their global resemblances are evaluated as follows:

$$Dis(c, c') = \sum_{j=1}^{j=p} \beta_j \, dis(m_j(c_i), m_j(c'_i)) + \qquad (2)$$

$$+\beta^s \, dis(St(c_{i\to i+1}), St(c'_{i\to i+1})) \qquad (3)$$

where $\beta_j \geq 0$ is the weight of the software metrics m_j and $\beta^s \geq 0$ is the weight of the stress factor. For the experimental study in section 4, we always take equal weights for structural metrics, i.e.,

$$\forall j \in \{0, \ldots, p\}, \beta_j = \frac{1 - \beta^s}{p}$$

which yields a normalized similarity function that depends only on the choice of β^s. It is noteworthy that the ratio between β^s and $1 - \beta^s$ translates the mutual importance of both high-level factors, design and stress, in the disimilarity computation and therefore by varying the value of β^s one reflects specific hypotheses about their respective role in the prediction. For this study we did not have strong reasons for distinguishing among structural metrics, so we kept their weights equal. This choice reflects the initial lack of knowledge about the relative importance of each metric, for the prediction goal. However, finer weights can be extracted from previous experiences by accounting for the contribution of an attribute to the correct/incorrect prediction for each case in a training dataset [27] (see section 4).

In contrast, various hypotheses were formulated regarding the role of the stress factor and the weights followed them. In a first setting, we decided to consider the stress as an ordinary characteristic rather than as a higher-level factor in the stability. Therefore, we assigned equal weights to all quantities describing a class, metrics and stress, i.e., $\beta^s = 1/(p+1)$. The second setting reflected our belief that the contribution of structural properties of a class to its stability is at best as important as the stress the class is experiencing. This has been translated as 50% weight assigned to the stress and the remaining 50% were equally distributed among the metrics ($0.5/p$). Finally, we adopted a radically different view on stress regarding it no more as quantity to contribute to an overall similarity between classes, but rather as an independent criterion used to refine the initial similarity-based comparison at a further step.

3.4 Solution adaptation

The aim of the adaptation step is to design a solution of the current problem from the solutions of the cases in the *BestMatch* set. For this purpose, a combination of the solutions in *BestMatch* is defined that represents a reasonable trade-off of several factors such as frequency of particular class in the set, ranks of the best matches, etc. For example, the adaptation may simply choose the solution of the first nearest neighbor or rather combine the solutions of several ones. For Boolean variables, the majority choice is usually used, whereas for continuous measures, linear combinations are preferred.

We used various strategies in the adaptation. On the one hand, the classical *1-NN* variant has been compared to *k-NN* of larger values for k. On the other hand, we have been following different hypotheses about the role of both factors in the stability prediction, i.e., structural similarity versus similarity in the applied stress. Thus, in the case of *k-NN* with $k > 1$, two adaptations have been compared. First, a linear combination of all stability values is considered, whereby each case stability is weighted by the respective distance. In this way, the nearer a case, the stronger the contribution of its own stability value. Second, we examined a hypothesis that stress factor may play the selector in the adaptation process. Technically speaking, stress values are no longer included in the similarity computation inputs ($\beta^s = 0$), but they are rather used to chose a single nearest neighbor, among the k retrieved classes, whose stability is then copied as a solution.

The following table 1 provides an illustration of the three strategies. It shows the five nearest neighbors of the class `java.net.Socket` (version 1.1) computed with a 0 weight for the stress.

Class name	Stability	Stress	Distance
`java.net.DatagramSocket`	.46	.45	.77%
`java.lang.System`	.94	.44	1.5%
`java.net.ServerSocket`	1.	.31	1.51%
`java.awt.Point`	.78	.5	1.75%
`java.awt.MenuComponent`	1.	.46	1.78%

Table 1. The five NN for `java.net.Socket` with respective similarity, stability and stress values.

The class `java.net.Socket` itself has a stability factor of $.62$ while the stress amounts to $.47$. Table 2 illustrates the results of the adaptation step, i.e., the predicted value for `java.net.Socket` according to each of the three strategies.

Strategy	Predicted stability
1-NN	.46
5-NN adaptation on stress	.78
5-NN weighted average	.88

Table 2. The three values for `java.net.Socket` stability.

4 Evaluation

In order to evaluate our CBR-powered stability prediction method, we applied it to dataset made up of the classes found in four major releases of JDK from which 14 structural metrics have been extracted (see figure 2).

To feed the remaining parameters of our model, i.e., the stress factor and the stability level, we further considered the three transitions between versions and compared the subsequent variants for each class. Thus, three test datasets have been constituted, one per transition, completed by a fourth one, made up of all classes in the first three sets.

Our experimental study followed a test protocol based on a "leave-one-out" validation strategy. In the sequel, relevant aspects of the global evaluation are discussed.

4.1 Choice of the variables

We choose 14 structural software metrics among the set of metrics used in different theoretical models [8, 7, 20, 9, 30]. The metrics belong to one of the four categories of *coupling*, *cohesion*, *inheritance* and *complexity* (see figure 2).

Metric	Description
Cohesion	
LCOM	Lack of cohesion in methods
COM	Cohesion metric
Coupling	
OCMAIC	Other class method attribute import coupling
OCMAEC	Other class method attribute export coupling
CUB	Number of used classes
Inheritance	
NOC	Number of children
NOP	Number of parents (including interfaces)
DIT	Depth of inheritance tree
MDS	Message domain size
CHM	Class hierarchy metric
Complexity	
NOM	Number of methods
NAA	Number of public attributes
MCC	McCabe's complexity weighted methods/class
WMC_LOC	LOC weighted methods per class

Figure 2. Metrics description

The choice of the metrics in figure 2 has been carefully studied. Actually, the set of 14 metrics emerged from a previous study on a similar but simpler experimental framework [15]. In that study, we examined the predictability of class stability for Java classes coming from several applications, based on a set of 22 metrics. One of the results suggested a limited subset of the entire metrics collection were well correlated with the stability variable (binary in that case, i.e., stable vs unstable). Based on this experience, we hypothesize that the same metrics are suitable for the prediction on datasets stemming from standard Java APIs.

4.2 Data collection

The selected software metrics were extracted from each Java class using the ACCESS tool of the Discover© environment[3]. For this purpose, we considered the major versions of Java, from its first stable version (1.0.2) to its most widely used version at present, 1.3.1. The latest API, version 1.4.1, emerged only recently and could not be included in our study.

The following table 3 illustrates the evolution in the size of the datasets along the version increase. The figures also indicate the number of classes that underwent some, even minor, changes between versions (the unstable classes) compared to classes that kept their public interface unchanged from previous version (the stable classes). It is noteworthy that the major shift in the composition of the Java API which has been operated between the versions 1.1 (Java 1) and 1.2 (Java 2) is also reflected by the table. In fact, several novel features have been incorporated into the language during this transition: inner and nested classes, enhanced event mode, etc., thus leading not only to the introduction of many new classes but also to a large number of modifications to already existing classes (37% of all classes). Moreover, the jump in the number of classes in the version 1.3 with respect to 1.2 is the result of the integration into the standard API of two previously independant libraries, Swing and CORBA.

JDK version	Nb of classes	Nb of stable classes	% of stable classes	Nb of unstable classes	% of unstable classes
JDK1.0.2 to JDK1.1	185	144	77,84%	41	22,16%
JDK1.1 to JDK1.2	580	363	62,59%	217	37,41%
JDK1.2 to JDK1.3	2158	1978	91,66%	180	8,34%
All versions	2923	2486	85,05%	437	14,95%

Figure 3. Data

The last line of the table represents the global dataset, i.e., the one resulting from the merge of the previous three. Thus, in this dataset, consecutive versions of the same class may be observed.

4.3 Experimental parameters

Several major parameters have been used in the experiments, their number reflecting the complexity of the stability prediction problem.

First, we used variable weights in the computation of the similarity measure (parameters β^s and β_j). There was no particular reason for distinguishing among structural metrics - although this would be an interesting track to follow - therefore we kept their weights equal in all experimental settings. In contrast, we made vary the relative weight of the stress variable with respect to the remaining 14 metrics, as we were intrigued about its specific role in the stability. Our hypothesis was that the stress is a major factor and should be assigned greater importance in the computation.

To test this hypothesis, we chose three different experimental settings. In the first setting, we assigned equal weights to all 15 quantities describing a class, i.e., metrics

[3] available at *http://www.mks.com/products/discover/developer.shtml*.

and stress, feflecting a straightforward hypothesis that the stress is an ordinary metrics rather than as a higher-level factor. The second setting reflected our belief that the stress should be seen as a factor of the same importance as the entire set of design-related properties. Thus, 50% weight was assigned to the stress alone and the remaining 50% were equally distributed among the 14 other metrics. Finally, we adopted a radically different view on stress regarding it no more as quantity to contribute to an overall similarity between classes, but rather as an independent criterion used to refine the initial similarity-based comparison at a further step.

Second, we used various mechanisms for the adaptation of the results from the similarity-based retrieval of cases to the resolution of the test case. On the one hand, the parameter k indicating the number of nearest neighbors participating in the adaptation procedure has been tuned. Two values for k, 1 and 5, have been studied, the former one leading to what is known in the CBR community as "null adaptation" as the stability of the unique nearest neighbor is simply copied. With a set of five neighbors, there are further possibilities, whereby a straightforward one would be to take the average of the five stability values. We preferred a bit finer version in which the weighted sum of these quantities is used with weights being the normalized similarity values. On the other hand, we played with the contribution of the stress factor to the CBR mechanisms and moved it from the retrieval step to the adaptation process. Thus, we used its value to select among the five nearest neighbors, the one whose stress value is the closest to that of the test case. This setting reflects our hypothesis that among a set of classes of similar structural properties, the one which has the best chances of nearing the stability for a given case, is the one whose stress is the closest to the stress experienced by that case.

Further parameters were used in the validation procedure as described in the next paragraph.

4.4 Error computation and global validation

We used particular settings to tune the computation of the error in the prediction. In fact, as the value to predict is a ratio of two integer quantities, which ranges between 0 and 1, exact predictions are simply unrealistic. Therefore, some degree of deviation in the prediction should be admitted. Among the different possibilities, we have chosen a simple model in which a tolerance threshold is used to separate sufficiently well predicted cases from the rest. Thus, a percentage q is specified as a parameter of the error computation mechanism and its role is as follows: given an effective stability value v_e and a prediction value v_p, if v_p fells within the interval of $[(100 - q) * v_e, (100 + q) * v_e]$ it is considered as valid, otherwise it is invalid. Intuitively, the greater the threshold, the bigger the number of the valid predictions, whereby a value of 100% for q leads invariably to no false predictions. For our experiments, a value of 20% has been used, which is fairly precise for stability prediction.

To evaluate the global rate of accuracy in the prediction, we used the correctness factor that accounts for the number of well predicted cases with respect to the total number of cases. Separately, we have computed the rates of the valid predictions for stable and unstable classes.

Finally, to make our results more representative, we used a validation strategy that computes average rates on several runs. Our current results have been obtained with a "leave-one-out" technique, which means that there have been as many runs as the size of the corresponding dataset. This technique has been preferred to the classical 10-fold cross-validation for two main reasons: on the one hand, the small number of unstable classes in the global dataset may easily lead to unbalanced distributions of those classes between the training set and the test set, while on the other hand, having a single class to compare to a large number of already examined and labeled classes is a situation close to the settings of the real problem of stability prediction.

4.5 Results

We have carried out a series of experimental tests over the four datasets with different parameters as explained in the previous paragraphs. Figure 4 presents a summary of the results for the major experiments for which the "leave-one-out" validation has been used.

Each line refers to a particular experimental setting with columns corresponding to parameters and datasets. Thus, for any combination of weight assignment and adaptation method, on the one hand, and dataset, on the other hand, we provide the respective value of the global correctness.

The above results are to compare to those obtained with a *10-fold cross-validation*, which is usually preferred to the *leave-one-out*, since more robust in general (see figure 5). However, this technique is more delicate to use on highly unbalanced dataset, as the global Java API dataset considered here, since it is hard to assure a good trade off between

Adaptation	Variable Weights	All cases
1-NN	equal weight	86,61%
Stress	equal weights	**87,47%**
5-NN	equal weights	86,40%

Figure 5. Prediction accuracy with CBR on all classes, 10-fold cross-validation.

Adaptation	Variable Weights	v1	v2	v3	All
1-NN	equal, metrics and stress	90,32%	83,10%	94,07%	91,52%
1-NN	metrics 50%, stress 50%	89,25%	82,07%	94,12%	91,38%
5-NN	equal, metrics and stress	**93,01%**	**87,59%**	94,07%	91,93%
5-NN	metrics 50%, stress 50%	91,94%	86,72%	92,73%	91,76%
Stress	equal metrics, no stress	91,94%	84,14%	**94,81%**	**92,13%**

Figure 4. Prediction accuracy with CBR for Java API classes with "leave-one-out" validation.

the random choice of the 10% samples and the minimum ration between stable (Stability = 1) and unstable (Stability < 1) cases to keep results sensible. In other terms, completely random choice of the samples reduces bias, but, as there are few unstable cases, may easily increase the disbalance in the remaining 90% of cases in the learning set. We chose to favor randomness in the data partition, hence, the results of this experiment are bare indicators of the possible drop in prediction accuracy due to disbalance in the data.

4.6 Interpretation

In the light of the above remarks about balance in the Java API dataset, the results from the first study may be interpreted as follows. First, globally, the larger set of nearest neighbors (five versus one), favors more precise predictions as the risk of wrongly selecting a value is reduced by the greater number of values considered.

In the first two transitions, where the percentage of unstable cases in the data is relatively high (between the fifth and the third of all cases) the most successful technique is a *5-NN* that considers both structural characteristics and stress as similarity factors. Thus, every individual measure is assigned the same weight in the similarity computation, whereby in the computation of the predicted stability value, a weighted average of the five stabilities in the *BestMatch* set is used. The good relative performance of the technique is due to the fact that stability and stress are similarly distributed over the classes and favoring one of them leads to a decrease in precision. To support this view, a correlation level of $-.5$ between stress and stability has been observed in the first version dataset ($v1$), but this vanishes in the data from later version. On the other hand, this fact offers a hint about the sharp drop in prediction correctness between $v1$ and $v2$: it reflects the impact of adding the version 1.2 of JDK which includes the biggest ratio of unstable cases. In the third transition dataset, the overwhelming number of classes (six in every seven) are stable, which makes the prediction of instability a hard task. In fact, since the probability of retrieving stable nearest neighbors is high, even for *5-NN* the computed stability tends to be an overestimation of the real value. Therefore, the most successful technique is the one that excludes the stress from similarity computation but uses it as a selector for the unique nearest neighbor, among the five retrieved, whose stability will be used as a prediction. The results of the 10-fold cross-validation, despite the sharp drop in the accuracy, indicate a similar ranking between techniques, with the adaptation on stress showing the best score.

To sum up, we may globally conjuncture that the appropriate use of the stress in stability prediction is as follows. At a first step, the most similar classes are retrieved on design-based criteria, i.e., unbiased by stress consideration, and at a second step, the stability is adaptated from the case solutions with respect to the stress levels in the test class and the retrieved cases. Furthermore, the adaptation and the retrieval may vary in shape, e.g., using *k-NN* for retrieval and *p-NN* for adaptation, with $p \leq k$.

5 Conclusion

In this paper we propose a case-based reasoning approach for predicting the stability of software items from relevant metric data and results about the stress. The technique presented here considers each item as a point in a multi-dimensional space, one dimension per metric, in which a distance function is defined. The stability of each new item is computed with respect to a fixed number of nearest cases in the case base whose stability and stress are known. Various techniques for adaptation of the stability values from nearest neighbors to the test case have been experimented whereby the most promising approach seems to be the use of the stress factor as adaptation selector rather than as basic similarity factor.

The resulting predictive model is a refinement of our previous work on stability prediction which had a scope limited to pure design considerations. In a very general sens, the new model can be seen as a mapping between stress levels for a class and its respective (expected) stability values. This sort of models fit better realistic situations in which the available data is neither of sufficient size nor representative enough to develop universally valid models by abstraction. Indeed, our similarity-based prediction avoids the pitfalls of over-generalization of logical classification models: the

preliminary results show that a very straightforward CBR classifier (*5-NN*, no tuning of weights, no domain theory) can perform significantly well when the stress estimation is properly fed in.

Our study is a first attempt to construct a realistic prediction models for stability and as such we consider it successful since the stability has been proved to be predictable. Moreover, the experiments we carried out confirmed the the utility of CBR for tasks where few theoretical knowledge is yet available. Thus, it could be applied to other maintenance tasks and quality factor prediction, in particular to the prediction of the stress from the results of some rough analysis of the requirements evolution.

To our view, exploring analogy through CBR does not mean that domain theories are excluded: we see both approaches as complementary and a natural integration between both, e.g., as a theory-powered case-based stability predictor, seems even more promising. In this respect, our future research is directed at the design of enabling techniques for a reasonable amount of domain knowledge, i.e., information about the structure of the manipulated items and about possible relationships among items, to be fed into the classification process. Another research track leads to the improvement of the feature-selection capabilities of our method to help reduce the number of the software metrics used in the prediction and fine-tune their respective weights.

References

[1] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 1(7):39–59, 1994.

[2] M. D. Almeida, H. Lounis, and W. Melo. An investigation on the use of machine learned models for estimating software correctability. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):565–593, 1999.

[3] J. Bansiya. Evaluating framework architecture structural stability. *ACM Computing Surveys (CSUR)*, 32, 2000.

[4] J. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *ACM Symp. Software Reusability (SSR'94)*, 1995.

[5] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *19th International Conference on Software Engineering*, 1997.

[6] L. Briand and J. Wüst. *Advances in Computers*, chapter Empirical studies of quality models in object-oriented systems. Academic Press, 2002.

[7] L. Briand, J. Wust, J. W. Daly, and V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51:245–273, 2000.

[8] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] S. Demeyer and S. Ducasse. Metrics, do they really help ? In *Langages et modèles à objets (LMO'99), 1999*, 1999.

[10] J. Esteva and R. Reynolds. Identifying reusable software components by induction. *International Journal of Software Engineering and Knowledge Engineering*, 1(3):271–292, 1991.

[11] M. Fayad. Accomplishing software stability. *Communications of the ACM*, 45, 2002.

[12] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *To appear in IEEE Transactions on Software engineering*, 2000.

[13] K. Ganesan, T. Khoshgoftaar, and E. Allen. Case-Based Software Quality Prediction. *International Journal of Software Engineering and Knowledge Engineering*, 10(2):139–152, 2000.

[14] M. Genero, L. Jimenez, and M. Piattini. Measuring the quality of entity relationship diagrams. In *In Proc. of 19th International Conference on Conceptual Modeling*, 2000.

[15] D. Grosser, H. A. Sahraoui, and P. Valtchev. Predicting Software Stability Using Case-Based Reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 295–298, 2002.

[16] J. Hakkarainen, P. Laamanen, and R. Rask. Neural networks in specification level software size estimation. In *26th Hawaii Int. Conf. System Sciences*, pages 626–634, 1993.

[17] S. Horikawa, T. Furnuhashi, and Y. Ucikawa. On fuzzy modelling using fuzzyneural networks with the backpropagation algorithm. *IEEE Trans. Neural Networks*, 3:801–806, 1992.

[18] N. Karunanithi, D. Whitley, and Y. Malaiya. Prediction of software reliability using connectionist models. *IEEE Trans. Soft. Eng.*, 18:563–574, 1992.

[19] S. Kumar, B. Krishna, and P. Satsangi. Fuzzy systems and neural networks in software engineering project management. *Journal of Applied Intelligence*, 4:31–52, 1994.

[20] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[21] Y. Mao, H. A. Sahraoui, and H. Lounis. Reusability hypothesis verification using machine learning techniques: A case study. In *IEEE Automated Software Engineering Conference*, 1998.

[22] R. Martin. Stability. *C++ Report*, 9(2), 1997.

[23] D. Parnas. Software aging. In *In Proc. 16th Int. Conference on Software Engineering (ICSE'94)*, 1994.

[24] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1997.

[25] R. S. Pressman. *Software Engineering, A Practical Approach*. fourth edition, McGraw-Hill, 1997.

[26] H. Sahraoui, M. Boukadoum, and H. Lounis. Building quality estimation models with fuzzy threshold values. *l'Objet*, 17(4), 2001.

[27] D. Wilson and T. Martinez. Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, 6, 1997.

[28] G. Wittig. Estimating software development effort with connectionist models. *Working Paper Series 33/95*, 1995.

[29] G. Wittig and G. Finnie. Using artificial neural networks and function points to estimate 4gl software development effort. *Australian Journal of Information Systems*, 1994.

[30] H. Zuse. *A Framework of software Measurement*. Walter de Gruyter, 1998.