Ranking Refactoring Suggestions based on Historical Volatility

Nikolaos Tsantalis and Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
email: nikos@java.uom.gr, achat@uom.gr

***Abstract*— The widespread acceptance of refactorings as a simple yet effective approach to improve the design of object-oriented systems, has stimulated an effort to develop semi-automatic tools for detecting design flaws, with simultaneous suggestions for their removal. However, even in medium-sized projects the number of detected occurrences can be so large that the refactoring process becomes intractable for the designer. It is reasonable to expect that some of the suggested refactorings will have a significant effect on the improvement of maintainability while others might be less important. This implies that the suggested solutions can be ranked according to one or more criteria. In this paper we propose the exploitation of past source code versions in order to rank refactoring suggestions according to the number, proximity and extent of changes related with the corresponding code smells. The underlying philosophy is that code fragments which have been subject to maintenance tasks in the past, are more likely to undergo changes in a future version and thus refactorings involving the corresponding code should have a higher priority. To this end, historical volatility models drawn from the field of forecasting risk in financial markets, are investigated as measures expressing the urgency to resolve a given design problem. The approach has been integrated into an existing smell detection Eclipse plug-in, while the evaluation results focus on the forecast accuracy of the examined models.**

Keywords: refactoring; code smell; historical volatility; software history; software repositories; forecasting models

I. INTRODUCTION

Refactoring as a means of preventive maintenance has gained wide acknowledgement [25], since the corresponding transformations are relatively easy to apply, constitute direct solutions to given design problems [11] and have a significant cumulative effect on design quality [23]. Their popularity is evident by the multitude of software tools providing support for both the identification of refactoring opportunities [10], [19], as well as the automation of their application [24].

However, even in medium-sized projects the identified refactoring opportunities can be numerous [7], imposing an additional effort on the designer for determining the priority according to which the refactorings should be applied. In our previous attempts to rank refactoring opportunities [32], [33] we employed an estimate of their impact on the design quality, i.e., the refactorings with a more extensive impact should be applied first. Nevertheless, the decision whether a refactoring should be applied or not, and in what order, is complex and usually calls for the expertise and intuition of the designer.

To minimize the effect of the subjective factor that determines the refactoring application order, we propose the examination of antecedent versions of a given project, to extract information concerning the urgency of a certain refactoring. The underlying philosophy is that refactorings make more sense when future adaptive or corrective maintenance is facilitated by their application. In other words, we assume that a refactoring would be more constructive when the related code fragment has been subject to a considerable volume of changes through past project generations. Conversely, if a piece of code remains unmodified over a number of generations, it would not be a top priority for the designer to apply a refactoring affecting it. In a similar manner, other approaches aiming to guide reverse engineering/maintenance effort [14] and to improve the detection of design flaws [29] have utilized the analysis of past changes.

In this paper, we propose an alternative ranking approach for refactoring opportunities that exploits information from past source code versions. To this end, we define the change in code smell intensity and use it as a means to capture modifications between successive software versions in code fragments related to a given design problem. Next, we introduce the concept of *code smell volatility*, inspired from forecasting models in financial markets, expressing the fluctuation in past changes of code smell intensity. Since volatility is considered to have a good forecasting power in financial contexts, we employ forecasting models of code smell volatility to rank refactoring suggestions. The higher the value of the predicted smell volatility is, the more urgent the application of the corresponding refactoring is considered.

Within the context of software engineering research the term volatility has been associated with *software volatility*, which refers to the frequency or number of enhancements per unit of application functionality over a specified time frame [2]. According to Barry and Slaughter [4] software volatility is associated with three dimensions, namely *amplitude* which measures the size of software modifications made to a system, *periodicity* which measures time between software modifications and *deviation* which is the variance of periodicity.

The rest of the paper is organized as follows: In Section II we introduce the concept of code smell volatility, while in Section III we propose measures for the quantification of the extent of change for three code

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.7

smells, namely *Long Method*, *Feature Envy* and *State Checking*. Section IV presents some implementation details about the tool supporting the proposed approach. The approach is evaluated in Section V by comparing the accuracy of four forecasting models and the similarity of the refactoring suggestion rankings produced by each model. Section VI presents the threats to the validity of our study. An overview of related work is presented in section VII. Finally, we conclude and discuss future work in Section VIII.

II. VOLATILITY OF CODE SMELLS

The overall goal of the proposed approach is to rank refactoring suggestions according to the urgency to resolve the corresponding design problems. Previous approaches implicitly define and quantify urgency according to the impact of suggested refactorings on certain design properties and metrics. Within the context of this work, the urgency for resolving a design problem is associated with past changes in the related code. The rationale behind this point of view is that refactorings constitute a form of preventive maintenance [6], [31] (i.e., activities aiming at the improvement of design qualities in order to facilitate future maintenance); therefore it is reasonable to prioritize refactorings which target pieces of code that have undergone maintenance in the past. Since prioritizing refactorings involves a kind of forecasting about future changes, we could draw ideas from other fields of research where forecasting based on past changes has been studied.

In financial markets and especially stock exchange markets, volatility has attracted growing attention by academics and practitioners. The reason is that volatility is closely related to risk and the general stability of financial markets and is considered an important factor when making investments [34]. In an economic context, volatility is the relative rate at which the price of a financial instrument moves up and down. Historical volatility is calculated as the annualized standard deviation of daily changes in price. A number of studies consider that volatility has in general a relatively good forecasting power, since trends in volatility are more predictable than trends in prices [8].

Within the context of preventive maintenance, risk lies in the decision to invest effort and resources in order to resolve design problems that will potentially improve the future maintainability of software. In the analysis of financial markets the axis of time refers to trading days, whereas in the study of software evolution the concept of time is represented by successive software versions.

In analogy to price changes in the financial context, we define *code smell volatility* based on changes that involve pieces of code affected by the smell. Assuming three successive software versions, *i*-1, *i* and *i*+1 (Figure 1), two changes can be defined, namely a change between versions *i*-1 and *i* ($change_{i-1,i}$) and a change between versions *i* and *i*+1 ($change_{i,i+1}$). A volatility measure for the transition between versions *i* and *i*+1 ($transition_{i+1}$) can be extracted by the standard deviation (σ) of the values quantifying the extent of these two changes.

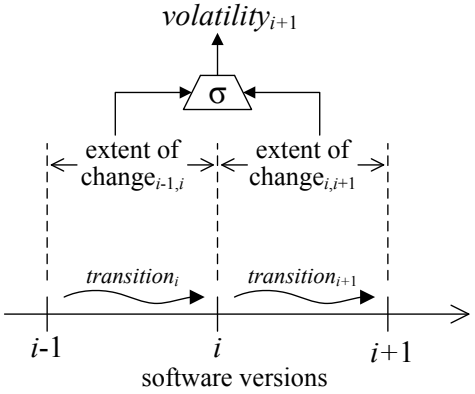

Figure 1. Calculation of code smell volatility.

Returning to our initial assumption, that past code changes related to a code smell constitute a reliable source of information regarding the prioritization of refactoring suggestions, we have employed existing models developed for forecasting future volatility. The models that we have considered in this study are: random walk, historical average, exponential smoothing, and exponentially-weighted moving average [5].

Random Walk (*RW*)

The simplest forecasting model is based on the assumption that the best forecast for the volatility of the immediately subsequent period is the volatility of the current period:

$$\hat{\sigma}_{t+1} = \sigma_t ,$$

where $\hat{\sigma}_{t+1}$ refers to the forecast of the volatility for period *t*+1 *and* σ_t is the actual observed volatility for period *t*.

Historical Average (*HA*)

The historical average model assumes that the conditional mean is constant and that the best forecast of future volatility is given by the average of past volatilities:

$$\hat{\sigma}_{t+1} = \frac{1}{t}\sum_{i=1}^{t}\sigma_i$$

where *t* refers to the number of past periods that are taken into account for the calculation of the average.

Exponential Smoothing (*ES*)

Exponential Smoothing is an adaptive forecasting model where forecasts are adjusted based upon past errors. Volatility is modeled as a weighted average of the previous forecast and actual values of volatility:

$$\hat{\sigma}_{t+1} = (1-\alpha)\hat{\sigma}_t + a\sigma_t$$

where α is the smoothing parameter ($0 < \alpha < 1$).

Exponentially-Weighted Moving Average (*EWMA*)

The EWMA model combines exponential smoothing with a moving average and the forecast is obtained as:

$$\hat{\sigma}_{t+1} = (1-\alpha)\hat{\sigma}_t + a\frac{1}{t}\sum_{i=1}^{t}\sigma_{t+1-j}$$

In our study the smoothing parameter α has been set to 0.5 for both the ES and the EWMA models, in order to assign equal weights to the previous predicted and actual values of volatility.

III. EXTENT OF CODE SMELL CHANGE

As already mentioned the underlying assumption of the proposed approach is that pieces of code which have been the subject of maintenance in the past are more likely to undergo changes in the future, and thus refactoring opportunities affecting them should have a higher priority. In the context of refactoring suggestion ranking, only changes which are relevant to the corresponding code smell intensity should be considered. In the following subsections we define the changes that should be quantified taking into account the particular characteristics of each code smell.

The smells that have been considered are the ones which are currently supported by our refactoring opportunity identification tool, named JDeodorant [16]. These smells are related to important design qualities in object-oriented systems such as method cohesion and complexity, appropriate allocation of methods in classes and the use of polymorphism.

A. *Long Method*

Long Method bad smell [11] manifests itself by means of methods with large size, high complexity and low cohesion among their statements. The cohesion of a procedure is usually estimated based on the degree of the participation of its statements in the computation of output variables. The larger the number of distinct and independent variable computations within the body of a procedure, the less cohesive the procedure is. According to several empirical studies procedures/modules with large size [1], high complexity [13], and low cohesion [22] require significantly more time and effort for comprehension, debugging, testing and maintenance. A solution to this problem can be given by extracting cohesive parts of a method which implement a distinct functionality into new separate methods through the application of appropriate *Extract Method* refactorings [11].

In general, the extent of change in a method's body between two successive software versions can be obtained by the number of edit operations that have to be made in the method of one version to convert it to the method of the other version. A suitable edit distance for this purpose is the Levenshtein distance [18] which is defined as the minimum number of edits needed to transform one sequence of tokens to another. The allowable edit operations are insertion, deletion or substitution of a single token.

Since the smallest piece of code that can be inserted, deleted or substituted within the body of a method and lead to a compilable method is a statement, we consider statements as tokens for the calculation of the Levenshtein distance. For a simple statement the corresponding token is its string representation. For a compound statement (i.e., statements having a body that contains a list of statements such as loops and conditionals) the corresponding token is the kind of the compound statement along with the string representation of its expression(s). Eventually, the body of a method is represented as a sequence of strings corresponding to the statements of the method. The fact that Levenshtein distance takes as input ordered sequences of tokens makes it appropriate for method comparison, where statements are by definition ordered.

To obtain a normalized value for the extent of change for a given method between two successive versions, the Levenshtein edit distance should be divided by the maximum attainable value for this distance [20]. The maximum value corresponds to the length of the largest input sequence. If we assume that there are two sequences *S1* and *S2* of *m* and *n* length ($m > n$), respectively, having no common tokens, then the conversion of *S1* to *S2* (with a minimum number of edits) requires the substitution of the *n* first tokens of *S1* with the *n* tokens of *S2* and the deletion of the remaining *m*-*n* tokens from *S1*. As a result, the total number of edits is $n + (m-n) = m$, which corresponds to the size of the largest sequence.

Thus, the extent of change between a method in version *i* (m_i) and the same method in version *i*+1 (m_{i+1}) is calculated based on the following formula:

$$ec_{LongMethod} = \frac{Ld(m_i, m_{i+1})}{\max(length(m_i), length(m_{i+1}))}$$

where:
Ld is the Levenshtein distance, and
length(*m*) is the number of statements of method *m*.

B. *Feature Envy*

Feature Envy bad smell is a sign of violating the design principle of grouping behavior (i.e., methods) with related data (i.e., attributes) and most of the times appears in the form of "a method that is more interested in a class other than the one it actually is in" [11]. The effect of this violation is twofold, since a class suffering from such bad smells has low cohesion (i.e., its methods do not operate on common attributes or with each other) and at the same time it is coupled with other classes of the program (i.e., its methods use attributes or methods from other classes of the program in order to implement their functionality). As a result, the existence of *Feature Envy* smells in a class makes the class more change-prone and error-prone due to propagation of changes and errors from the classes that it depends on. Furthermore, the existence of such smells in a class decreases its understandability and testability due to the need for understanding, debugging and testing the classes that it depends on in order to perform maintenance activities, such as the implementation of new features or bug fixing. A solution to this design problem can be given by moving the misplaced methods to the "envied" classes through the application of appropriate *Move Method* refactorings [11].

The intensity of the Feature Envy smell is obviously related to the number of "envied" members, that is, the number of foreign members that the problematic method accesses from a certain class. Therefore, the extent of change for this smell is determined based on the variation in the number of "envied" members between two successive software versions.

The degree of dependence between the problematic method and the "envied" class is related primarily to the number of distinct member accesses rather than the total number of accesses. In other words, the dependence becomes more intense when an additional member is accessed rather than when an already "envied" member is accessed multiple times.

The extent of change in this case can be normalized by dividing the difference in the number of distinct accesses between methods in two successive versions with the total number of the accessible members of the "envied" class. Since the total number of accessible members is equal to the maximum attainable value for the number of distinct accesses, this normalization guarantees that the extent of change lies in the range [0, 1]. Moreover, there is a possibility that the number of members of the "envied" class changes between two successive versions due to the addition or removal of members. Therefore, normalization should be performed by dividing with the maximum number of elements of the "envied" class between the two versions.

Thus, the extent of change regarding a Feature Envy smell involving method *m* and "envied" class *EC*, between version *i* and version *i*+1 is calculated based on the following formula:

$$ec_{FeatureEnvy} = \frac{|accesses(m_i, EC_i) - accesses(m_{i+1}, EC_{i+1})|}{\max(size(EC_i), size(EC_{i+1}))}$$

where:
accesses(*m*, *EC*) is the number of distinct accesses of method *m* to the members of class *EC*, and
size(*EC*) is the total number of accessible members of class *EC*.

C. *State Checking*

State or Type Checking bad smell [9], [17] constitutes a direct violation of the Open/Closed principle which states that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". It is often employed as an alternative approach to polymorphism in order to simulate *late binding* and *dynamic dispatch* and manifests itself as conditional statements that select an execution path either by comparing the value of a variable representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through RunTime Type Identification (RTTI) mechanisms. The effects of this violation are a) the introduction of additional complexity due to conditional statements consisting of many cases, b) the duplication of code fragments due to conditional statements scattered in many different places of the program that perform state-checking on the same cases for different purposes [11], and c) the modification of already existing code for the introduction of new states in future software versions. As a result, the maintenance of multiple state-checking code fragments operating on common states requires significantly more effort and may introduce consistency errors. A solution to this design problem can be given by replacing the conditional structures with calls to polymorphic methods.

The presence of a State Checking code smell signifies the lack of a State/Strategy design pattern [12]. Within this context, the branches of the problematic conditional structure correspond to the subclasses of the missing State hierarchy. Any change that affects the State Checking code fragment might occur on any of the three following axes:

- Changes in the statements of the conditional branches. In the State pattern equivalent such changes correspond to modifications in the body of the concrete subclass methods.
- Changes in the number of conditional branches. In the State pattern equivalent such changes correspond to the addition/removal of concrete subclasses in the State hierarchy.
- Changes in the number of conditional structures within the entire system, performing state checking on the same set of states. In the State pattern equivalent, such changes imply that the pattern would be further utilized in additional parts of the system (and possibly that new concrete methods would be added to the subclasses of the State hierarchy).

The benefit of employing the State pattern becomes even stronger when there are multiple occurrences of conditional structures performing state checking on the same states. In case of the State pattern any change is performed within the boundary of the State hierarchy, whereas in the state checking alternative, all relevant conditional structures throughout the system have to be located, examined and eventually modified. Consequently, the primary factor which determines whether past changes related to the state checking smell would be facilitated by the removal of the smell is the number of related conditional structures within the entire system.

Therefore, the extent of change is determined by the difference in the number of conditional structures performing state checking on the same set of states, between two successive project versions p_i and p_{i+1}. Since there is no upper bound for the number of conditional structures that might be present in a system, normalization is achieved by dividing with the maximum number between the two versions.

Thus, the extent of change regarding a State Checking smell related to a set of states *S*, between two successive project versions p_i and p_{i+1} is calculated based on the following formula:

$$ec_{StateChecking} = \frac{|cond(p_i, S) - cond(p_{i+1}, S)|}{\max(cond(p_i, S), cond(p_{i+1}, S))}$$

where:
cond(*p*, *S*) is the number of conditional structures in project *p* performing state checking on set of states *S*.

IV. Implentation and Tool Support

The proposed approach has been fully automated and integrated into the JDeodorant [16] code smell detection Eclipse plug-in. In order to perform a code smell evolution analysis, the user must load a set of successive versions for a given Java project in the Eclipse workspace and perform an identification of code smells in a single version of the loaded projects. For each selected code smell, the tool automatically discriminates the relevant projects in the workspace and sorts them according to their version number in order to form a sequence of consecutive project pairs. For each project pair in the sequence, the tool computes the extent of change between the two project versions of the pair. This computation may require the analysis of a single method in each project version (e.g., the method corresponding to the selected code smell in the case of Feature Envy and Long Method), or the complete project code in each version (e.g., in the case of State Checking). The results of the analysis are presented in a frame (Figure 2) showing the extent of change for each

project pair and a particular instance of a code smell. For the project pairs where a non-zero extent of change has been computed, the tool presents a diff comparison of the corresponding code fragments in the two project versions, allowing the user to easily locate the exact changes between the two versions of the code smell. If it is not possible to compute the extent of change for a given code smell between two successive software versions, the value for the extent of change is indicated as not applicable (N/A).

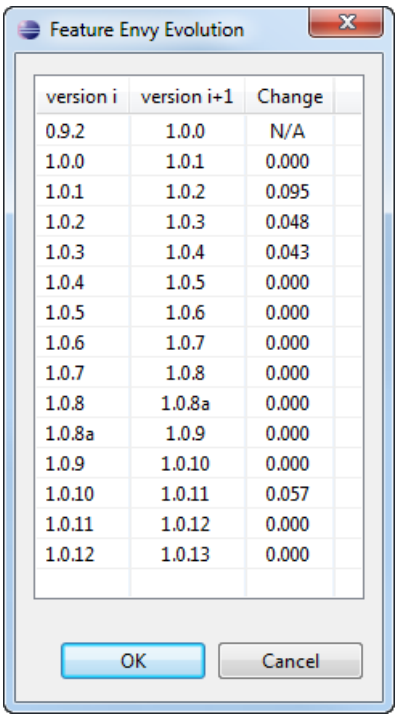

Figure 2. Screenshot showing the extent of change for each project pair.

The code smell evolution analysis provides useful information and insights to the maintainer regarding the following aspects:

a) Code smell discontinuities, which take place when it is not possible to compute the extent of change for a given code smell between two successive software versions. In the case of Feature Envy and Long Method code smells, a discontinuity may occur when the problematic method is not present in at least one of the compared software versions. In the case of State Checking code smell, a discontinuity may occur when there do not exist any state-checking code fragments in both compared versions.

b) Code smell "births" or "eliminations", which take place when the extent of change is maximum (i.e., equal to one) for a given code smell between two successive software versions and designate the introduction or the removal of a design problem. In the case of Feature Envy, birth/elimination occurs when the number of envied members is equal to zero in one of the compared software versions. In the case of State Checking, birth/elimination occurs when the number of state-checking code fragments is equal to zero in one of the compared software versions.

c) Prioritization of preventive maintenance on parts of the software that change more frequently due to the existence of a code smell. This can be achieved by sorting the identified refactoring opportunities according to the future code smell volatility extracted by the employed forecasting models. This alternative ranking mechanism based on historical volatility can be used in combination with structural ranking mechanisms (e.g., sorting based on the impact of the refactoring solutions that resolve the identified code smells on certain design quality characteristics or metrics) to provide a more complete view by taking into account both historical as well as design quality aspects.

V. Evaluation

The proposed evaluation aims at comparing the accuracy of the four examined volatility forecasting models and indirectly at investigating the suitability of such models for ranking refactoring suggestions. The comparison will be performed along two axes: a) a direct comparison of their forecast accuracy in terms of the root mean square error, and b) a comparison of the similarity between the ranking of refactoring suggestions according to the actual volatility and the rankings produced by each forecasting model. The results have been obtained by extracting the refactoring suggestions concerning three code smells and computing the extent of changes between successive software versions in two open-source projects.

The criteria for selecting appropriate projects for the evaluation of the proposed technique are the following:

a) The source code of the projects should be publicly available, since both the identification of code smells as well as the computation of the extent of change between successive project versions requires source code analysis. Furthermore, source code availability will make possible the replication of the proposed study.

b) The projects should present a sufficient number of code smells to enable the extraction of safer conclusions.

c) A sufficient number of stable versions should be available for the examined projects to enable a thorough analysis of historical changes.

The projects which have been selected based on the aforementioned criteria are JMol and JFreechart.

JMol is a free, open source molecule viewer for students, educators, and researchers in chemistry and biochemistry which has been constantly evolving since 2004. The selected project versions range from 11.0 to 11.6. In total, 26 successive project versions are included within this range, leading to 25 version pairs to be examined for changes. Figure 3 shows the evolution in the number of classes and source lines of code throughout the examined JMol versions. The smells that have been examined in project JMol are State Checking and Feature Envy. To highlight the accumulation of changes in particular software transitions, Figure 4 illustrates the average extent of change for all identified smells of the two aforementioned types throughout the evolution of JMol.

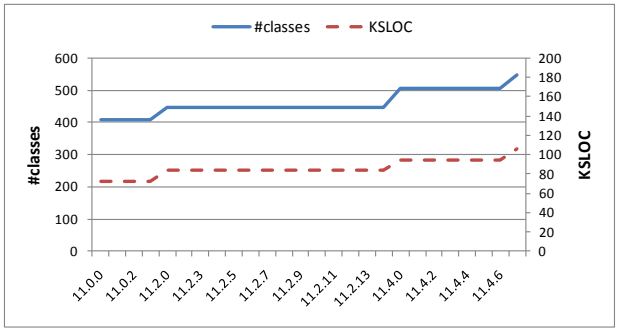

Figure 3. Evolution in the number of classes and source lines of code (KSLOC) for JMol project.

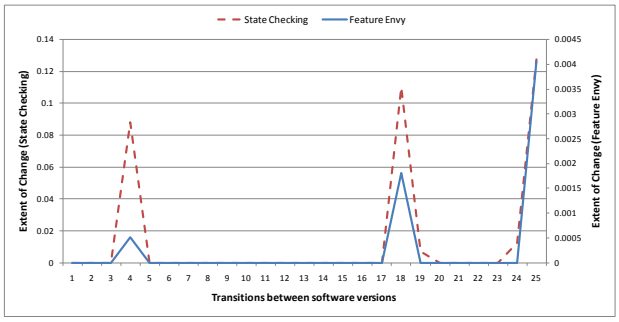

Figure 4. Average extent of change (State Checking and Feature Envy code smells) for the transitions between JMol versions.

JFreechart is a rather mature open-source chart library which has been constantly evolving since 2002. The selected project versions range from 1.0.0 to 1.0.13, which is currently the latest version. In total, 15 successive project versions are included within this range, leading to 14 version pairs to be examined for changes. Figure 5 shows the evolution in the number of classes and source lines of code throughout the examined JFreechart versions. The smell that has been examined in project JFreeChart is Long Method. Figure 6 shows the average extent of change for all identified Long Method smells throughout the evolution of JFreeChart.

The identification of code smells took place in the latest examined version of each project, namely version 11.6 for JMol and 1.0.13 for JFreechart.

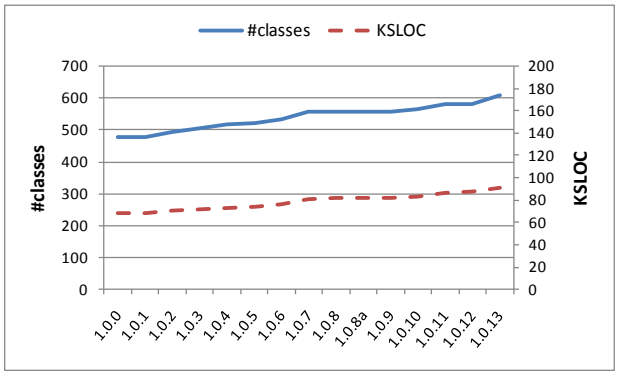

Figure 5. Evolution in the number of classes and source lines of code (KSLOC) for JFreechart project.

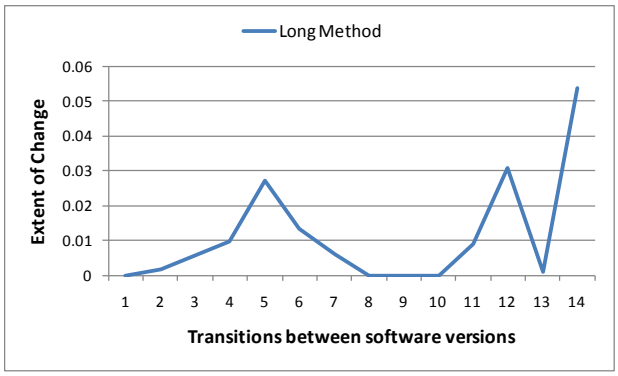

Figure 6. Average extent of change (Long Method code smell) for the transitions between JFreeChart versions.

A. *Comparison of Forecast Accurary*

One of the most popular measures to test the forecasting power of a given model is the root mean square error (RMSE) between the actual volatility values and the predicted ones:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{\sigma}_i - \sigma_i)^2}$$

where N represents the total number of value pairs.

The number of Long Method code smells found in project JFreeChart (in package org.jfree.chart) is equal to 130. The cases presenting code smell discontinuities (i.e., when the problematic method does not exist in one of the two examined versions) have been excluded from the analysis, since it would not be possible to calculate the actual volatility values. The number of remaining cases is equal to 96, from which 14 cases (14.6%) correspond to methods that did not change throughout the examined software versions.

The evolution of RMSE through the successive versions of JFreeChart for all employed forecasting models is shown in Figure 7.

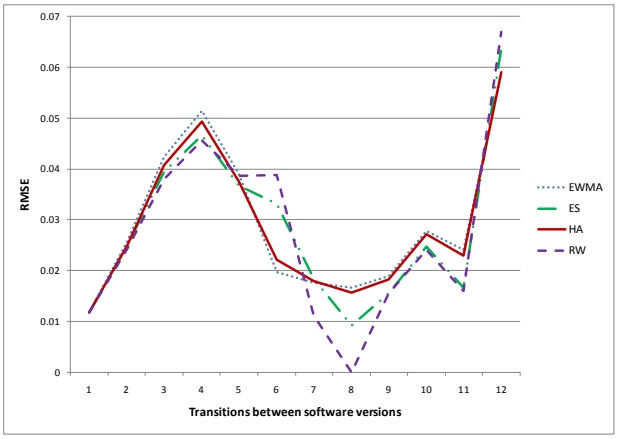

Figure 7. Evolution of RMSE for the examined forecasting models (Long Method code smells found in project JFreeChart)

The number of Feature Envy code smells found in project JMol (in package org.jmol.viewer) is equal to 269. The cases presenting code smell discontinuities have been excluded from the analysis, resulting in 117 remaining

cases. The number of cases for which the envy did not change throughout the examined software versions is equal to 98 (83.8%).

The evolution of RMSE through the successive versions of JMol for all employed forecasting models is shown in Figure 8.

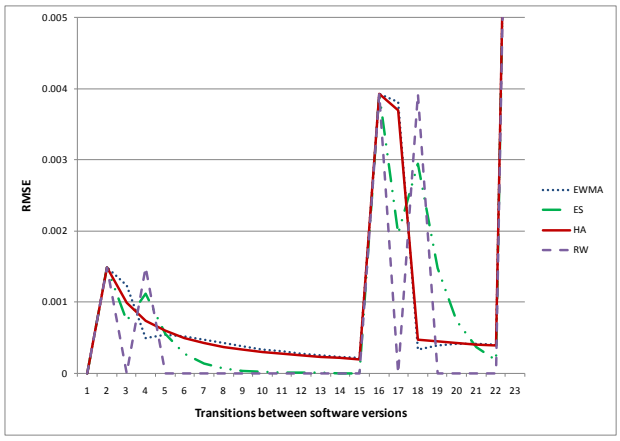

Figure 8. Evolution of RMSE for the examined forecasting models (Feature Envy code smells found in project JMol)

The number of Type Checking code smells found in project JMol is equal to 48. The cases presenting code smell discontinuities (i.e., when there do not exist any state-checking code fragments in both compared versions) have been excluded from the analysis, resulting in 20 remaining cases. The cases for which the number of state checking code fragments did not change throughout the examined software versions are 12 (60%).

The evolution of RMSE through the successive versions of JMol for all employed models is shown in Figure 9.

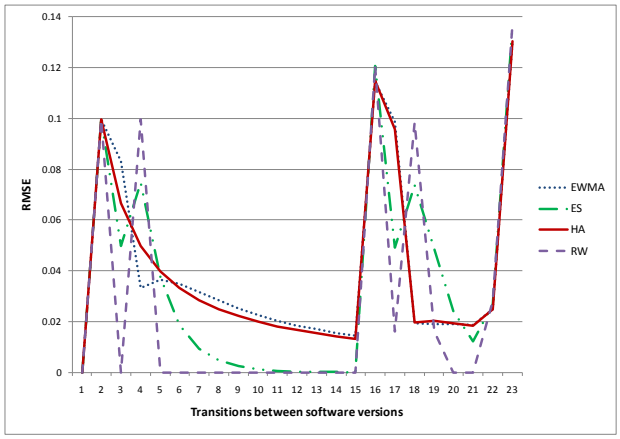

Figure 9. Evolution of RMSE for the examined forecasting models (State Checking code smells found in project JMol)

The following remarks can be made from Figures 7-9:

- The Historical Average and the Exponentially-Weighted Moving Average models present similarities since the EWMA model encompasses the calculation of the average of all previous historical volatility values (which is weighted by the smoothing parameter α, set to 0.5 in our evaluation). In the extreme case where α is set to 1, the two models become identical.
- The Random Walk model is being favored by the existence of successive versions with zero volatility, presenting a zero forecasting error. On the other hand, in the case of extreme changes in volatility, it exhibits tremendous error variation.
- The evolution of RMSE for Feature Envy and State Checking code smells in project JMol follows the same pattern. This can be attributed to the fact that most changes occurred in specific project versions, regardless of the smell type.
- The peaks in the evolution of RMSE usually occur when versions with zero volatility are followed by an abrupt volatility change (or vice versa).

To provide an overview of the accuracy achieved by each forecasting model, the overall root mean square error for each smell is given in Table 1.

Table 1. Overall RMSE for each smell and forecasting model.

	Random Walk	***Historical Average***	***Exponential Smoothing***	***EWMA***
Long Method (*JFreeChart*)	0.032646	0.031972	0.032176	0.032608
Feature Envy (*JMol*)	0.003311	0.003295	0.003309	0.003301
State Checking (*JMol*)	0.052842	0.052967	0.053051	0.053879

parameter α in Exponential Smoothing and EWMA models has been set to 0.5

Since RMSE depends on the units in which the variable under examination is measured, it makes no sense to compare the forecast accuracy among different smell types. On the other hand, RMSE can be used in order to compare the forecast accuracy provided by each model, regarding the same smell type. As it can be observed from Table 1, the Historical Average forecasting model achieves the lowest error for the Long Method and the Feature Envy smells, and the second lower error for the State Checking smell. Furthermore, the more sophisticated models (ES and EWMA) that take proximity into account by assigning higher weights to more recent volatility values do not provide higher accuracy. As a result, due to the simplicity and relatively good forecast accuracy of the Historical Average model, it can be considered as a more appropriate strategy for ranking refactoring suggestions.

B. *Ranking comparison*

Since the forecasting models extract the anticipated smell volatility for future software evolution, the corresponding estimated volatility resulting in the last available transition can be directly employed as ranking criterion for refactoring suggestions. In this section we present the differences among the rankings produced by each of the examined forecasting models and the ranking produced by the actual volatility for the last transition.

The employed measure for comparing the similarity between the alternative rankings is Spearman's footrule [3] which is applied to two rankings of the same set. It should be emphasized that this measure cannot be applied to rankings for non-identical sets of elements (i.e., when the two compared lists contain ranks for non-overlapping elements). For two permutations σ_1 and σ_2 of the same set of element S, the Spearman's footrule is computed as:

$$Fr^{|S|}(\sigma_1,\sigma_2)=\sum_{i=1}^{|S|}|\sigma_1(i)-\sigma_2(i)|$$

where $\sigma(i)$ denotes the rank of the element i and $|S|$ denotes the size of set S.

Spearman's footrule is actually the sum of the ranking differences in the two permutations for each element of S. The measure takes a zero value when the two rank lists are identical, and the maximum obtained value is $½|S|^2$ when $|S|$ is even and $½(|S| + 1)(|S| - 1)$ when $|S|$ is odd. By dividing the result of the measure by its maximum value, the measure is normalized between 0 and 1 and attains the value 0 when the two lists are identically ranked and the value 1 when the lists appear in opposite order. Thus, the *normalized Spearman's footrule*, *NFr* is computed as:

$$NFr^{|S|}=\frac{Fr^{|S|}}{\max Fr^{|S|}}$$

In our study, the set of elements corresponds to the set of identified refactoring opportunities, while the permutations correspond to the ranking of the refactoring opportunities according to each employed model and actual volatility.

The normalized Spearman's footrule between the rankings produced by each model and the ranking corresponding to the actual volatility for the last transition is shown in Tables 2-4 for the Long Method, Feature Envy and State Checking code smells, respectively.

Table 2. Spearman's footrule among ranking of refactoring suggestions based on the actual volatility for the last transition and rankings based on forecasting models (Long Method smells found in project JFreeChart).

	Random Walk	***Historical Average***	***Exponential Smoothing***	***EWMA***
Actual	0.6220	0.3255	0.5334	0.3238

Table 3. Spearman's footrule among ranking of refactoring suggestions based on the actual volatility for the last transition and rankings based on forecasting models (Feature Envy smells found in project JMol).

	Random Walk	***Historical Average***	***Exponential Smoothing***	***EWMA***
Actual	0.0096	0.0210	0.0199	0.0213

Table 4. Spearman's footrule among ranking of refactoring suggestions based on the actual volatility for the last transition and rankings based on forecasting models (State Checking smells found in project JMol).

	Random Walk	***Historical Average***	***Exponential Smoothing***	***EWMA***
Actual	0.07	0.13	0.14	0.13

According to the results for the Long Method code smell in project JFreeChart, which exhibits the largest frequency of changes, the Historical Average and the Exponentially-Weighted Moving Average models achieve the most similar rankings with the one corresponding to the actual volatility. On the other hand, for Feature Envy and State Checking code smells in project JMol, which exhibit a significantly lower frequency of changes, the Random Walk model achieves the most similar rankings with the one corresponding to the actual volatility. The reason is that for an evolution with a limited number of changes, the models that have a kind of "memory" (like the Historical Average and EWMA) tend to reflect these past changes on future forecasts, leading to larger dissimilarity in the cases where the volatility in the transition of interest is zero. Additionally, it can be observed that the Historical Average and the EWMA forecasting models produce almost identical rankings for all examined code smells. This similarity is reasonable, since as already explained EWMA encompasses the calculation of the average of all previous volatility values.

Consequently, in projects exhibiting frequent changes, the Historical Average forecasting model should be preferred as a ranking mechanism for the refactoring suggestions, taking also into account the low RMSE error that it achieves.

VI. Threats to Validity

An obvious threat to the conclusion validity of our study is related to the correctness of our initial assumption. In this study emphasis was given on historical information related to code smells; however, developers might choose to rank refactoring opportunities according to other criteria such as the impact of the corresponding refactorings on the design quality, conceptual issues, etc.

The extent of change for the examined smells, as defined in Section III, reflects our perspective on how the intensity of the corresponding problems should be measured. As a result, this poses a threat to the construct validity of our study, since other approaches for the quantification of the extent of change would lead to different experimental results.

Another threat to the construct validity is related to the granularity in analyzing past source code information. In this study we employed stable software versions from software repositories in order to compute the extent of changes between successive instances of a given project. A more fine-grained approach would be to employ successive revisions from version control systems.

A threat to the external validity of our study, which limits the ability to generalize our findings, is the relatively small number of analyzed projects. In other words, the particular characteristics of the examined projects as well as the number of past versions that have been analyzed definitely affect the accuracy of the employed forecasting models.

VII. Related Work

In the field of searching and suggesting sequences of refactoring applications, several research works have focused on the dependencies and interrelationships between relevant refactorings in order to produce feasible sequences.

Mens et al. [21] represented refactorings as graph transformations and used the techniques of critical pair analysis and sequential dependency analysis to detect mutual exclusions (i.e., when two transformations are incompatible with each other and the application of the one prohibits the application of the other), asymmetric conflicts (i.e., when it is possible to apply two transformations in a particular order, but not the other way around) and sequential dependencies (i.e., when the application of a transformation relies on other transformations that should be applied before) between refactorings.

Piveta et al [27] proposed an approach that makes use of Deterministic Finite Automata (DFA) in order to represent refactoring sequences and a set of simplification

rules to reduce the search space. The number of possible refactoring sequences is narrowed by discarding those that semantically do not make sense and avoiding those that lead to the same results.

Qayum and Heckel [28] modeled refactoring steps as graph transformation rules and used the unfolding technique from graph transformation theory to identify dependencies and conflicts between refactoring steps.

Other research works have focused on producing refactoring sequences that optimize certain design quality criteria in a given object-oriented software system.

Seng et al. [30] proposed an approach for improving an aspect of object-oriented design which is related with the cohesion of class modules. To this end, they created a special model that examines a set of pre- and post-conditions in order to simulate the application of Move Method refactorings. Furthermore, they used a genetic algorithm that produces a single sequence of Move Method refactoring applications maximizing the value of a fitness function based on coupling, cohesion, complexity and stability metrics.

Harman and Tratt [15] applied the concept of Pareto optimality to the problem of class module cohesion optimization. An advantage of Pareto optimality is that it allows defining multiple fitness functions and thus helps to avoid the construction of a single complex function which requires the normalization and weighted combination of several metrics. Moreover, it produces multiple optimal refactoring sequences, allowing the designer to select an appropriate sequence based on his preferences and conceptual criteria.

O'Keeffe and Ó Cinnéide [26] proposed an approach for improving an aspect of object-oriented design which is related with the correct utilization of inheritance. The quality evaluation functions used to rank the alternative designs were based on metrics from the QMOOD hierarchical design quality model. The search techniques used to find the optimal solution were three different versions of a local search algorithm, namely first-ascent, steepest-ascent and multiple-restart Hill Climbing, respectively, and a meta-heuristic technique, namely Simulated Annealing.

In contrast to the aforementioned search-based approaches which produce sequences of refactoring transformations that lead to a system with optimized design based on certain criteria, Tsantalis and Chatzigeorgiou [32], [33] proposed a stepwise approach for improving the design quality of object-oriented software systems. In this approach, the identified refactoring opportunities are ranked according to the impact of their application on design quality. The user is allowed to apply a refactoring solution, regardless of its ranking position, by taking also into account conceptual criteria. After the application of the selected refactoring on source code the system is re-examined and a new ranked list of refactoring opportunities that improve the design of the current system is extracted. In this way, the maintainer can form a sequence of refactoring applications that satisfy both conceptual soundness and design quality improvement.

VIII. Conclusions and Future Work

In this paper, we have proposed an approach for ranking identified refactoring opportunities based on past source code changes. More specifically, we defined the volatility of changes related to code smell intensity, since evidence from the study of financial markets has shown that historical volatility has a good forecasting power. The underlying assumption of the proposed ranking strategy is that pieces of code that have been the subject of maintenance in the past are more likely to undergo changes in the future and thus should have a higher priority for refactoring. To this end, we have applied four forecasting models to predict the future code smell volatility that is eventually used as a ranking criterion of refactoring opportunities.

The evaluation results concerning the accuracy of the examined forecasting models indicate that the Historical Average model, which predicts future volatility based on the average of past volatility values, provides the lowest root mean square error. Moreover, in the case of projects with frequent changes, our findings indicate that the Historical Average and Exponentially-Weighted Moving Average models produce the most similar rankings of refactoring suggestions compared to the ranking based on the actual volatility.

Future work could broaden the range of examined smell types and forecasting models in order to validate the suitability of smell volatility as a criterion for ranking refactoring opportunities. Moreover, software history could be analyzed at a more fine-grained level in order to examine changes that occur in shorter time frames. Another interesting research perspective is the combination of different ranking strategies based on historical information, structural properties (such as the impact of refactorings on design characteristics) or dependencies among the suggested refactorings.

Acknowledgments

This work has been funded by the Research Committee of the University of Macedonia, Greece.

References

[1] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81-94, November 1993.

[2] R. D. Banker, and S. A. Slaughter, "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement," *Information Systems Research*, vol. 11, no. 3, pp. 219-240, September 2000.

[3] J. Bar-Ilan, M. Mat-Hassan, and M. Levene, "Methods for comparing rankings of search engine results," *Computer Networks*, vol. 50, no. 10, I. Web Dynamics, pp. 1448-1463, July 2006.

[4] E. Barry, and S. A. Slaughter, "Measuring software volatility: a multi-dimensional approach," *21st International Conference on Information Systems* (ICIS 2000), pp. 412-413, 2000.

[5] C. Brooks, "Predicting stock index volatility: Can market volume help?," *Journal of Forecasting*, vol. 17, pp. 59-80, 1998.

[6] N. Chapin, J. E. Hale, K. Md. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no 1, pp. 3-30, January/February 2001.

[7] A. Chatzigeorgiou, and A. Manakos, "Investigating the Evolution of Bad Smells in Object-oriented Code", *7th International Conference on the Quality of Information and Communications Technology* (QUATIC'2010), pp. 106-115, 2010.

[8] L. A. Connors, *Options, Trading and Volatility Trading*, M. Gordon Publishing Group, Los Angeles, CA, USA, 1999.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufman, 2003.

[10] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells", *23rd IEEE International Conference on Software Maintenance* (ICSM'2007), pp. 519-520, 2007.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, Boston, MA, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[13] G. K. Gill, and C. F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284-1288, December 1991.

[14] T. Gîrba, S. Ducasse, and M. Lanza, "Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes," *20th IEEE International Conference on Software Maintenance* (ICSM'04), pp. 40-49, 2004.

[15] M. Harman, and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," *9th Annual Genetic and Evolutionary Computation Conference* (GECCO 2007), pp. 1106-1113, 2007.

[16] JDeodorant, Available at: http://www.jdeodorant.com, 2010.

[17] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2004.

[18] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707-710, 1966.

[19] R. Marinescu, G. Ganea, and I. Verebi, "inCode: Continuous Quality Assessment and Improvement", *14th European Conference on Software Maintenance and Reengineering* (CSMR 2010), 2010.

[20] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 143-169, March/April 2009.

[21] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software and Systems Modeling*, vol. 6, no. 3, pp. 269-285, 2007.

[22] T. M. Meyers and D. Binkley, "An Empirical Study of Slice-Based Cohesion and Coupling Metrics," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 1, December 2007.

[23] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," *Lecture Notes in Computer Science*, vol. 5082, pp. 252-266, 2008.

[24] E. Murphy-Hill, and A.P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, pp. 38-44, Sept.-Oct. 2008.

[25] E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," *31st International Conference on Software Engineering* (ICSE 2009), pp. 287-297, 2009.

[26] M. O'Keeffe, and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *The Journal of Systems and Software*, vol. 81, no. 4, pp. 502-516, April 2008.

[27] E. Piveta, J. Araujo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price, "Searching for Opportunities of Refactoring Sequences: Reducing the Search Space," *32nd Annual IEEE International Computer Software and Applications Conference* (COMPSAC 2008), pp. 319-326, July 28-August 1, 2008.

[28] F. Qayum, and R. Heckel, "Analysing Refactoring Dependencies using Unfolding of Graph Transformation Systems," *6th International Conference on Frontiers of Information Technology* (FIT 2009), December 16-18, 2009.

[29] D. Rațiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using History Information to Improve Design Flaws Detection, " 8th European Conference on Software Maintenace and Reengineering (CSMR 2004), pp. 223-232, 2004.

[30] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," *8th Annual Conference on Genetic and Evolutionary Computation* (GECCO 2006), pp. 1909-1916, 2006.

[31] E. Stroulia, and R. Kapoor, "Metrics of Refactoring-Based Development: An Experience Report," *7th International Conference on Object-Oriented Information Systems* (OOIS 2001), pp. 113-122, August 27-29, 2001.

[32] N. Tsantalis, and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347-367, May/June 2009.

[33] N. Tsantalis, and A. Chatzigeorgiou, "Identification of Refactoring Opportunities Introducing Polymorphism," *The Journal of Systems and Software*, vol. 83, no. 3, pp. 391-404, March 2010.

[34] J. Yu, "Forecasting volatility in the New Zealand stock market," *Applied Financial Economics,* vol. 12, pp. 193-202, 2002.