Software Qual J
DOI 10.1007/s11219-013-9204-4

Investigating the effect of evolution and refactorings on feature scattering

Theodore Chaikalis · Alexander Chatzigeorgiou · Georgina Examiliotou

Abstract The implementation of a functional requirement is often distributed across several modules posing difficulties to software maintenance. In this paper, we attempt to quantify the extent of feature scattering and study its evolution with the passage of software versions. To this end, we trace the classes and methods involved in the implementation of a feature, apply formal approaches for studying variations across versions, measure whether feature implementation is uniformly distributed and visualize the reuse among features. Moreover, we investigate the impact of refactoring application on feature scattering in order to assess the circumstances under which a refactoring might improve the distribution of methods implementing a feature. The proposed techniques are exemplified for various features on several versions of four open-source projects.

Keywords Feature identification · Feature scattering · Program understanding · Requirements traceability · Software evolution · Refactorings

1 Introduction

One of the major difficulties of software maintenance is the linking of certain functional requirements with the corresponding software modules that implement them, a process known as requirements traceability. This is a crucial part of program understanding and a non-trivial task since the required information is in most cases inefficiently documented (Biggerstaff et al. 1994; Antoniol et al. 2002; Eisenbarth et al. 2003; Trifu 2010). According to Gotel and Finkelstein (1994), requirements traceability is the ability to follow a requirement from its specification through its deployment in code, in a both forward and backward directions. This activity is also defined as Concern (Eaddy et al. 2007; Trifu 2010), Concept (Biggerstaff et al. 1994) or feature location (Eisenbarth et al. 2003) since the goal is to identify the source code elements implementing a certain functional requirement. In the following, we adopt the term *feature* as defined by Eisenbarth et al.

T. Chaikalis (✉) · A. Chatzigeorgiou · G. Examiliotou
University of Macedonia, Thessaloniki, Greece
e-mail: chaikalis@uom.gr

Published online: 11 May 2013

(2003) which refers to a distinct, observable, unit of behavior of a system that can be exercised by the end user.

Several studies have indicated that extensive feature scattering (when the implementation of a feature is scattered throughout a large number of software modules) and feature coupling (i.e., increased inter-dependence between features) are factors that increase error-proneness and instability (Wilde and Scully 1995; Eisenbarth et al. 2003; Garcia et al. 2005; Gibbs et al. 2005; Koschke and Quante 2005; Filho et al. 2006; Greenwood et al. 2007; Robillard and Murphy 2007; Eaddy et al. 2008; Conejero et al. 2009; Revell et al. 2011). As an illustrative example, Robillard and Murphy (2007) stress that in order to modify the "save" feature of JHotDraw, the developer has to follow the implementation of this feature throughout at least 35 classes, which are at the same time involved in other features as well, imposing a significant challenge. The problem of feature scattering might deteriorate as software evolves not only due to the expected enhancement of functionality over time, but also due to poor design decisions. In extreme cases, it can lead to systems where a single feature involves hundreds of classes and over a thousand of methods.

The need to continuously monitor software quality and to facilitate software maintenance calls for an appropriate interpretation of requirements traceability in the context of software evolution. Under this perspective, we propose several means for the analysis and visualization of data concerning the evolution of the scattering in the requirements implementation and the distribution of methods implementing a specific feature in the involved classes. We also propose methods for the analysis and visualization of reuse among features at the method level (i.e., how many methods are shared by two features), a matter that is important since an eventual reuse of classes and methods among features provides a reasonable justification for extended feature scattering over source code, which would otherwise be interpreted as a worrying symptom. Finally, we evaluate the impact of refactoring application, as part of preventive maintenance, on feature scattering in order to investigate whether common refactorings affect the distribution of methods that participate in the implementation of a feature.

The data and the visualizations that can be extracted allow software stakeholders (and particularly maintainers and quality engineers) to shed light on questions such as

- How fast is the number of classes and methods involved in the implementation of a certain feature increasing with the passage of software versions?
- Is the distribution of methods contributing to the implementation of a feature uniform?
- Is this distribution becoming more unbalanced as software evolves?
- Are classes/methods reused in the implementation of different features?
- How similar are features to each other, based on their common implementation, and how is this similarity changing over time?
- Is refactoring application improving the scattering of features in source code or not?

To illustrate that the extracted data can provide insight into the evolution of the examined systems, we have run the proposed analyses for a number of successive versions of four open-source projects and for several of their features. The examined systems should be regarded as a sample to exemplify the use of the proposed analyses. It should be clarified that emphasis is given in the proposed techniques rather than the actual results, and therefore, no attempt to generalize the findings is being made.

The majority of previous studies on requirements traceability focus on creating precise and accurate feature location techniques (Zou et al. 2009) aiming at the analysis of individual software versions. In this work, we emphasize the need to investigate the evolution of feature scattering over software versions, and we also perform a more fine-grained analysis which considers not only the evolution of classes and methods involved in the implementation of a feature but also the common classes among features (Wong et al. 2000; Greevy et al. 2006).

For the purpose of the proposed analyses, we employed tools and techniques that are borrowed from various fields: Formal investigation and visualization of the evolution of feature scattering are performed by using Formal Concept Analysis, a technique that has also been used for the identification of features in source code (Eisenbarth et al. 2003; Poshyvanyk and Marcus 2007). The Gini coefficient (Gini 1921), a measure of statistical dispersion typically used for quantifying the inequality of income distribution, is employed to observe the evolution of the distribution of the methods implementing a certain feature over the involved classes. The evolution of method reuse by different features is studied by using a measure of similarity that has been originally applied in paleontology in order to illustrate part-whole relations. Finally, the evolution of feature similarity based on their common methods is visualized by exploiting multi-dimensional scaling, a widely used tool for data visualization.

The rest of the paper is organized as follows: In Sect. 2, we describe the individual steps of the proposed process, regarding the number of modules that are involved in the implementation of features, the formal representation of scattering by means of concept lattices, the way that methods are distributed among classes as well as the evolution of this distribution and the reuse among features. Results from the application of these steps on four case studies are presented in Sect. 3. The impact of refactoring application on feature scattering is investigated in Sect. 4. Threats to validity are discussed in Sect. 5, while the related work is presented in Sect. 6. Finally, we conclude in Sect. 7.

2 Proposed process

In order to investigate the evolution of feature scattering over several versions of a software system, the classes and methods implementing each feature should be identified for each of the examined versions. This constitutes one of the major challenges in the area of requirements traceability and particularly of feature location (Biggerstaff et al. 1994; Antoniol et al. 2002; Eisenbarth et al. 2003; Trifu 2010). In our case, the extraction of classes and methods involved in the implementation of selected features has been performed by employing dynamic analysis with the use of a Java Profiler (Jprofiler 2011). Dynamic analysis as an approach for feature location has also been adopted in other efforts (Eisenbarth 2003; Koschke and Quante 2005; Poshyvanyk et al. 2007).

In order to capture the creation of class instances and method calls related to a specific feature, we set up a scenario that exercises the feature and executes it in analogy to the approach employed by Wilde and Scully (1995), while the program is running in profiling mode. For example, in the case of the JMol chemical structure viewer that we analyze in Sect. 3, the scenario for profiling the rendering of molecules which are stored in files of type *mol* contains the following steps: 1. Click File Menu, 2. Select Open File, 3. Navigate to Aspirina.mol file, 4. Click OK. The analysis is only restricted to the source code of the system classes of the projects. In other words, methods and classes belonging to external packages and libraries are excluded. No further filtering on the obtained data is performed.

The entire process that we have followed in order to analyze the scattering of features is illustrated in Fig. 1. In the first step, selected features are exercised on the application of interest while being monitored by the profiler. Next, the methods invoked in the executed feature are analyzed to obtain the classes in which they reside and to generate the reports shown on the right-hand side of Fig. 1. Regarding the reports, their interpretation can be performed in the following sequence: An overview of feature scattering evolution is provided by the graphs showing the number of involved classes and methods in each version. A formal representation of feature scattering and its evolution can be obtained by Formal Concept Analysis. Further insight into the problem of feature dispersion can be obtained by studying the distribution of methods among the involved classes. Finally, similarity among features in terms of common methods should be examined, since this could provide a justification for the increased scattering. Each analysis is described separately in the following subsections.

2.1 Classes involved in the implementation of features

A number of studies conclude that extensive scattering of a given feature in numerous classes hinders not only the tracing of requirements in code, but also the comprehensibility of the underlying flow of events and therefore encumbers extensibility (Wilde and Scully 1995; Eisenbarth et al. 2003; Garcia et al. 2005; Gibbs et al. 2005; Greenwood et al. 2007; Koschke and Quante 2005; Filho et al. 2006; Robillard and Murphy 2007; Revell et al. 2011). Furthermore, according to Eaddy et al. (2008), the scattering of feature implementation across the program is statistically connected to the number of defects, and consequently, programs with increased feature scattering would probably exhibit more defects and inferior quality. The first goal of our study is to measure the number of classes that contribute to a specific feature by using the metric Count of Number of classes (CDC) (or methods—CDO) that has been introduced by Filho et al. (2006) and has also been employed in Aspect-oriented programming (Garcia et al. 2005; Marcus and Maletic 2003). However, since our goal is to gain insight into the evolution of feature scattering, apart from measuring the number of classes statically, which is for a given snapshot of the examined systems, we also measure the evolution of CDC over a number of successive software versions.

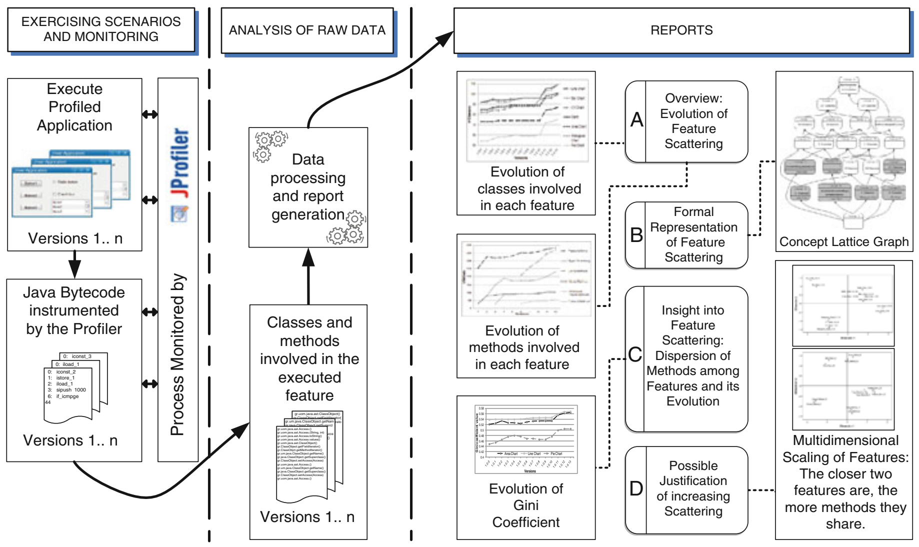

EXERCISING SCENARIOS AND MONITORING
ANALYSIS OF RAW DATA
REPORTS
Execute Profiled Application
Versions 1.. n
Java Bytecode instrumented by the Profiler
Versions 1.. n
Process Monitored by JProfiler
Data processing and report generation
Classes and methods involved in the executed feature
Versions 1.. n
Evolution of classes involved in each feature
Evolution of methods involved in each feature
Evolution of Gini Coefficient
A Overview: Evolution of Feature Scattering
B Formal Representation of Feature Scattering
C Insight into Feature Scattering: Dispersion of Methods among Features and its Evolution
D Possible Justification of increasing Scattering
Concept Lattice Graph
Multidimensional Scaling of Features: The closer two features are, the more methods they share.

Fig. 1 Data collection and analysis process

2.2 Formal analysis of feature scattering evolution

Formal Concept Analysis (FCA) deals with binary relations and uses mathematical lattice theory in order to identify meaningful groups of objects that share common attributes (Ganter and Wille 1996). It has been applied in the field of feature location in order to facilitate the process of tracing specific code units that implement a feature and also to increase the accuracy of the proposed methodologies (Eisenbarth et al. 2003; Poshyvanyk and Marcus 2007). Poshyvanyk and Marcus (2007) used Formal Concept Analysis to model the relation between methods and attributes, while Eisenbarth et al. (2003) exploited FCA to model relationships between concepts and computational units that implement them. Inspired by those approaches, we have applied FCA in order to model and analyze the relations between features and classes that implement each feature. Our goal is to study the evolution of feature scattering by comparing the concept lattices of different versions. The following paragraph briefly describes the theoretical background and provides an example for better understanding of the underlying notions.

Considering the implementation of a feature f (from the set of all examined features F) by a class c (from the set of system classes C) as a relation $r \subseteq F \times C$, the tuple (F, C, r) is a formal context. A formal context is essentially a binary relation table, indicating which of the classes are involved in the implementation of each feature. A tuple (F_i, C_i) is called a concept if and only if all features in the set F_i (extent of the concept) are implemented by all classes in the set C_i (intent of the concept).

We can define a partial ordering relation for the concepts (F_i, C_i) in a formal context by inclusion: if (F_i, C_i) and (F_j, C_j) are concepts, $(F_i, C_i) \leq (F_j, C_j)$ whenever $F_i \subseteq F_j$ or dually whenever $C_i \supseteq C_j$. Based on this partial ordering, a formal context can be graphically represented as a directed acyclic graph (DAG) where nodes represent concepts and edges denote the relations between them. Usually, the sparse form of the concept lattice is employed, where a particular node n is labeled only with each class $c \in C$ and each feature $f \in F$ that is introduced by node n.

Let us consider the example shown in Fig. 2 (Eisenbarth et al. 2003) adapted to illustrate relations between features and classes. Considering features $\{f_1, f_2, f_3\}$ and classes $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$ for a hypothetical system, the set of relations between them can be represented as a two-dimensional matrix also known as Formal Context (Fig. 2a). The concepts that can be derived from this matrix of relations are shown in Fig. 2b. The most general concept (i.e., the classes common to all features) is denoted by T, while the most special concept (i.e., the features containing all classes) is denoted by $\perp$. Figure 3c depicts a graphical representation of the same information known as a concept lattice (sparse form).

From the analysis of a concept lattice, the following two basic pieces of information can be extracted (several other conclusions that can be drawn and which are relevant to feature scattering are presented in Sect. 3.b):

- A feature f involves all classes at and above the node at which the feature appears. For example, feature f_1 (introduced in Concept 4) requires 4 classes, which can be found by traversing upward all paths starting from Concept 4 and ending at the top node, namely c_1, c_4, c_6, and c_7.
- A class c is required for all features at and below the node at which the class appears. For example, class c_4 is involved in the implementation of f_1 and f_2.

The aforementioned analysis can be applied to different software versions in order to investigate the evolution of feature scattering, as it will be shown in the case studies (Sect. 3.b).

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
f_1	x			x		x	x
f_2		x		x	x		x
f_3			x		x	x	x

(a)

$\top$	$\{f_1, f_2, f_3\}$, $\{c_7\}$
Concept 1	$\{f_1, f_2\}$, $\{c_4, c_7\}$
Concept 2	$\{f_1, f_3\}$, $\{c_6, c_7\}$
Concept 3	$\{f_2, f_3\}$, $\{c_5, c_7\}$
Concept 4	$\{f_1\}$, $\{c_1, c_4, c_6, c_7\}$
Concept 5	$\{f_2\}$, $\{c_2, c_4, c_5, c_7\}$
Concept 6	$\{f_3\}$, $\{c_3, c_5, c_6, c_7\}$
$\perp$	$\{\emptyset\}$, $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$

(b)

(c)

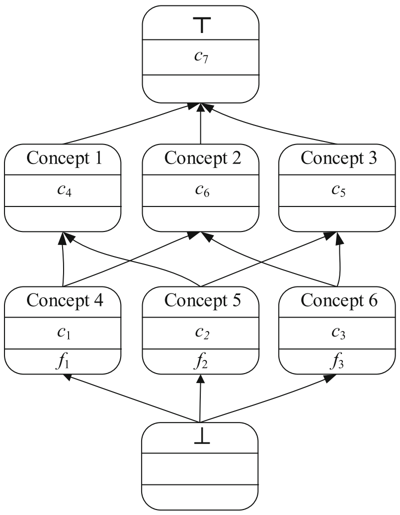


Fig. 2 Example of Formal Concept Analysis. Formal Context (**a**), Concepts of the Formal Context (**b**), and the sparse representation of the corresponding concept lattice (**c**)

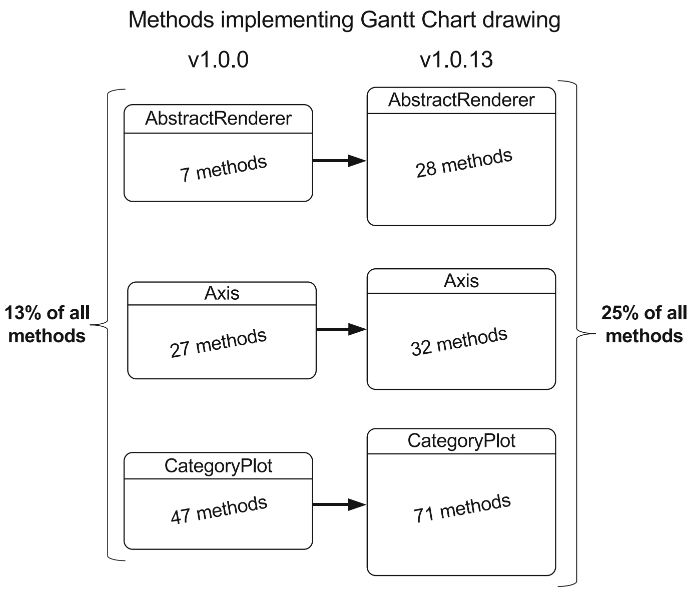


Fig. 3 Evolution of the number of methods for the top three classes in JFreeChart's Gantt Chart functionality

2.3 Distribution of methods among classes

The number of system modules that implement a specific feature might provide a useful insight into the feature's scattering, but it is a rather coarse-grained analysis due to the lack of information about the way in which methods are distributed over the corresponding classes. In this context, we have recorded the number of methods contributing to the implementation of each examined feature for each of the involved classes. Moreover, we studied the evolution of the distribution over a number of generations.

Textbooks that provide practical guidelines for proper object-oriented design and programming (Riel 1996; Sharp 1997) as well as general object-oriented design principles to avoid the creation of "God" or "Blob" classes advise that a systems' functionality should be distributed over the classes of their specific domain. Under this perspective, we suggest that methods implementing a feature should be distributed as uniformly as possible over the involved classes, otherwise classes with a lion's share of feature responsibilities will emerge. The problem often manifests itself in even more worrying form, in the sense that these God classes tend to attract even more functionality over time.

To provide a graphical illustration of this "rich-get-richer" concept, which is frequently and strongly present in technological and social networks (Barabasi 2000), in the evolution of feature scattering, Fig. 3 shows the number of methods for three classes that contribute to the implementation of the Gantt Chart drawing functionality in project JFreeChart, for the first and last examined versions, respectively (Sect. 3). The total number of classes that contribute to this functionality is 63. The three classes shown in Fig. 3 contain 13 % of all methods in the first version and 25 % of all methods in the last version. In other words, 3 out of 63 classes ended up in carrying out one quarter of the Gantt Chart functionality (measured in methods). This can be regarded as a definite sign of unbalanced distribution of methods among classes involved in the implementation of a feature.

2.4 Quantifying the evolution of method distribution

The distribution of methods among the classes that contribute to the implementation of a feature could be investigated accurately if it was presented in a dynamic form, where information about all historical versions of the project will be embedded. For this purpose, we have employed the Gini coefficient (Gini 1921), which is a measure of statistical dispersion. The Gini coefficient, a single numeric value between 0 and 1, has been widely employed in a wide range of diverse fields to study the inequality of a distribution. Most commonly it is used as a measure of inequality of wealth in a country, but recently it has also been employed in the field of Software Engineering. Vasa et al. (2009) employed the Gini coefficient to quantify the distribution of selected metrics over all system modules as an approach that outperforms the explanatory efficiency of the mean value, which is usually employed. Results of the evolutionary analysis of successive releases for numerous projects revealed high Gini values resulting from skewed distributions, while the authors highlighted the increased functionality that few classes must carry, making them oversized and rigid. Goeminne and Mens (2011) used the Gini index for the study of the distribution of developer contribution to open-source projects. Results came out with high Gini values indicating that the majority of development effort is carried out by a small, core group of people, while the rest of the development community contributes only a fraction of work.

A low value for the Gini coefficient implies a uniform distribution of a measure over the elements of a population. In our context, a low value indicates that the methods contributing to the implementation of a certain feature are distributed in a relatively uniform fashion over the involved classes. On the other hand, a high value indicates an uneven distribution and in the extreme case where the Gini coefficient is close to one, a single involved class would contain almost all the required functionality for a feature. Essentially, the Gini coefficient quantifies in the form of a clean and separate metric the localization and distribution of feature implementation.

Usually the deviation from the perfectly even distribution is depicted graphically by means of the Lorenz curve (Lorenz 1905) which, in our context, plots the proportion of the total number of methods (y axis) that are cumulatively contained in the bottom x % of the classes. As an example, let us consider the functionality related to the creation of a XY Chart in version 1.0.13 of JFreeChart. Figure 4 shows the cumulative distribution of methods over the cumulative distribution of classes. A perfectly uniform distribution of the methods contributing to the execution of this feature over the involved classes, would be represented by the 45 degree line, usually referred to as the line of equality (x % of the classes contain x % of the methods). The Gini coefficient can be obtained as the ratio of the area that lies between the line of equality and the Lorenz curve over the total area under the line of equality. Further the Lorenz curve from the 45 degree line lies, the higher the Gini coefficient value is. According to the results, the distribution of methods contributing to the XY Chart feature is highly skewed. As it can be observed, around 90 % of the classes host 50 % of the involved methods, which means that another 10 % of the classes host the rest 50 % of the methods. The corresponding Gini coefficient in this case is 0.581.

Another metric that can quantify the modularity of features is the *Degree of Scattering* (DoS), originally defined by Eaddy et al. (2007). The *Degree of Scattering* quantifies the extent by which the implementation of a feature is scattered among many classes. It builds upon the *Concentration* (CONC) metric that was defined by Wang et al. (2000). The purpose of Concentration is to "*quantitatively reflect how much of a feature is in a component*" by considering the blocks of code that belong to a software component and are executed by a feature. We believe that without loss of generality, we can consider methods as blocks of code and classes as software components. So, in our context, *Concentration* of a class C in a feature F can be defined as

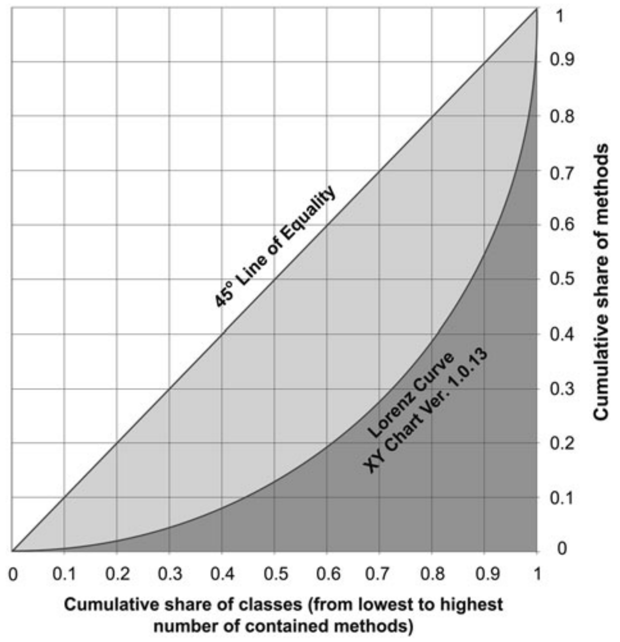


Fig. 4 Graphical representation of Gini coefficient

$$\mathrm{CONC}(F, C) = \frac{\mathrm{MethodsofclassCrelatedtoFeatureF}}{\text{Methods related to Feature F}}$$

Degree of Scattering is a measure of the statistical variance of the concentration of a feature across all program elements in relation to the worst case, where the feature's implementation is uniformly distributed across all program classes (Eaddy et al. 2008).

$$\mathrm{DoS}(F) = 1 - \frac{|\mathrm{S_C}| \sum_{C \in S_c} (\mathrm{CONC}(F, C) - \mathrm{CONC_{WORST}})^2}{|\mathrm{S_C}| - 1}$$

where S_C is the set of classes that contribute to the feature F.$\mathrm{CONC_{WORST}}$ is the concentration of the worst case, where the implementation of F is uniformly distributed across all involved classes and is calculated as $\frac{1}{|S_C|}$.

As an example, let us consider the following system with 2 classes contributing to one feature (left-hand side of Fig. 5). Class A contains 4 methods and class B contains 2 methods participating to the implementation of feature F, as shown in the left-hand side of Fig. 5. The values of the Gini coefficient and the DoS metric are also shown. The Gini coefficient is relatively low, since the entire functionality (6 methods) is spread over two classes in a relatively reasonable way. For the same reason, the Degree of Scattering is relatively high.

Next, we assume that the system evolves to a second version (right-hand side of Fig. 5) and that the change consists in adding four methods to class A, contributing to the implementation of feature F. Clearly, this modification leads to a system where the functionality is spread in a more unbalanced way than in version 1.

As it can be observed from the value for the Gini coefficient and the Degree of Scattering, the two measures are somehow antisymmetric, in the sense that the addition of new methods to class A caused an increase in the value of the Gini coefficient and a decrease in the value of DoS. According to the DoS, the evolution has led to a system where almost all functionality related to feature F is concentrated into class A (lower scattering). On the contrary, the Gini coefficient captures the fact that the functionality related to feature F is distributed in an unbalanced way in the second version, and this is reflected in the increased value of the coefficient.

As it has been mentioned in Sect. 2.a, we have opted for absolute measures (i.e., the number of classes and the number of methods) in order to provide a first overview of feature scattering, rather than a measure based upon statistical variance, such as the Degree of Scattering (DoS) across classes. The reason is that, in the context of software evolution, the Degree of Scattering which quantifies simultaneously both the number of classes implementing a feature and the localization of the implementation in terms of where

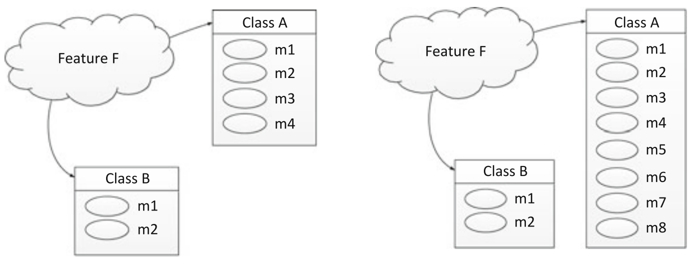


Fig. 5 Evolution of Gini coefficient and Degree of Scattering

methods reside might yield confusing results. Assume, for example, that in version i, a number of classes contribute equally (by 25 %) to the implementation of a feature and that in the next version $i + 1$, the number of involved classes increases by one, which, however, contributes to the implementation of the feature by an extremely small number of methods (e.g., 2 % removed from the other four classes). In that case, a subtle decrease in the Degree of Scattering would be observed (from 1 to 0.95), as the implementation is localized mostly in the initial classes, whereas a first interpretation should highlight that the number of involved classes has increased and scattering deteriorated. Regarding the use of the Gini coefficient, although its variations are opposite to the variations of the DoS measure, it appears to be more sensitive to such kind of changes which might be valuable when studying feature scattering. For the aforementioned example, the change in the Gini coefficient is much more drastic (from 0 to 0.18) highlighting that the feature's implementation in the second version is non-uniformly spread over the involved classes.

2.5 Distance between features

So far, the excessive number of classes and methods involved in the implementation of each feature has been recognized as a factor that possibly increases the required effort to understand and maintain the corresponding requirements (Robillard and Murphy 2007) and even the number of anticipated defects (Eaddy et al. 2008). However, a reasonable question is whether features share classes and methods among their implementations. This would imply that a certain degree of reuse is achieved which reduces development effort and eases maintenance, thus offering a justification for a possibly extended scattering of features in source code. In this section, we present results concerning the commonality between features employing a binary similarity measure.

An abundance of distance and similarity measures can be found in the corresponding literature serving a variety of needs (Choi et al. 2010). The most commonly binary measure used for quantifying the similarity between two sets is the Jaccard similarity which considers the number of elements that are present in both sets and the number of elements which are unique in each set (Naseem et al. 2011). The measure that we employed for evaluating the similarity between two features stems from paleontology (Simpson 1960) and essentially treats two groups as identical if one is a subset of the other. In theory, two features should have a distance equal to zero, if they exactly employ the same set of methods. However, since this might be an unrealistic scenario, we would like to extend the notion of zero distance between two features f_1 and f_2 to the cases where the methods implementing f_1 constitute a subset of the methods implementing f_2.

This measure (Simpson similarity) tends to eliminate the effects of discrepancy in size between two samples and highlights part-whole relations (Simpson 1960). In analogy to natural evolution where part-whole relations between samples might be informative on the evolution of populations, when assessing the evolution of software, we would also like to gain insight into the degree of reuse among features. In other words, let us consider a feature implemented by certain methods. If a second feature is implemented later, on top of the existing code base, by reusing the already implemented methods (and most probably by adding a number of new methods), this feature should be considered as "close" to the initial one, indicating a high degree of reuse. It might be extremely demanding to expect that a new feature employs exactly the same set of methods (and for this reason, we avoided the use of Jaccard similarity) but it would be considered as good practice to reuse all the existing methods, if possible, and introduce additional methods for the new functionality. This aspect of reuse can be accurately captured by the Simpson similarity.

Under this consideration, the distance of two features according to the Simpson similarity can be calculated as

$$\text{distance}(f_1,f_2) = 1 - \text{similarity}(f_1,f_2) = 1 - \frac{|\text{common Methods}(f_1,f_2)|}{\min(|\text{methods}_{f_1}|, |\text{methods}_{f_2}|)}$$

where $|\text{methods}_{f_1}|$ corresponds to the number of methods implementing feature f_1 $|\text{methods}_{f_2}|$ corresponds to the number of methods implementing feature f_2, and,|common Methods $(f_1,f_2)|$ represents the number of common methods between features f_1 and f_2

To obtain a graphical representation of the similarity among features and to provide a tool for assessing whether features are becoming more distant during the evolution, implying reduction in the degree of reuse among them, we propose the use of multidimensional scaling (MDS) for visualizing distances. MDS (Chen et al. 2008) is an approach that allows representing information contained in a set of data by a set of points usually in a two-dimensional Euclidean space. These points are arranged spatially in a way that geometrical distance between points reflects the numerical measure of distance between the examined data items. In other words, what multidimensional scaling is to find a set of vectors in a p-dimensional space (in our case coordinates in a 2-dimensional Euclidean space). As a result, the axes of the extracted plots correspond to the dimensionality of the employed space. The orientation of the axes is arbitrary, and any rotation of the plane will give rise to another valid solution. The interpretation of dimensions is at the discretion of the researcher who attempts to identify what is varying as we move along the two axes (Bartholomew et al. 2008). However, the output of multidimensional scaling may be valuable even if one cannot ascribe meaning to the axes, since the graphical representation can facilitate the comprehension of patterns in the data (i.e., one might be able to identify clusters of closely placed points).

Conventional MDS application would lead to two separate Euclidean distance models, one for each of the examined versions. To understand the nature and extent of association between the examined features, the proximity of points in the derived space needs to be interpreted (Singh 2007). However, the orientation of the axes for each MDS chart can be arbitrary, hindering the comparison between the two versions. Therefore, we adopted a different approach in which all examined features of both versions are fed into a single analysis. Consequently, the resulting diagrams illustrate the distances among all features for two versions, allowing us to investigate the evolution between the similarity of features and consequently the reuse among them.

Multidimensional scaling has been previously used by Fisher and Gall (2003) in order to visualize the proximity between problem report data. The distance between two problem reports was defined as the number of commonly modified files to fix both problems, while groups of feature-related reports have been formed enabling the identification of hidden dependencies between features. The dependencies among features have been visualized by means of MDS for the Mozilla project and for the years 1999–2002. In a more general context, Kuhn et al. (2008) employed MDS to map software artifacts to a two-dimensional space employing the vocabulary of each artifact in order to measure the distance among them.

3 Case studies

In this section, we illustrate the aforementioned techniques and measures in order to study the evolution of feature scattering on four open-source projects, namely JFreeChart, JDeodorant, Jmol and jEdit. JFreeChart is an open-source chart library (JFreeChart 2011) which has been constantly evolving since 2000. JDeodorant is an Eclipse plug-in that automatically identifies design problems, known as “bad smells,” and eliminates them with appropriate refactoring applications (JDeodorant 2011). It has been constantly evolving for more than 5 years as a project of the Computational Systems and Software Engineering Laboratory at the Department of Applied Informatics, University of Macedonia, Greece. Jmol is a Java viewer for chemical structures such as crystals, materials and biomolecules in 3D, which has been evolving since 2002 (JMol 2012). jEdit is a text editor especially built for programmers that can be extended by numerous plug-ins and has been evolving since 1998 (jEdit 2012). The evolution of size characteristics [lines of code (LOC), number of classes (NOC) and number of methods (NOM)] for the examined versions of all projects is shown in Table 1. LOC refers to lines that contain at least one statement, method signature or class definition including lines with comments and blank ones.

Seven features have been selected for the analysis of JFreeChart, six for JDeodorant, six for jEdit and five for Jmol. Table 2 briefly outlines the examined features of the four projects. It should be mentioned that the selected features cannot be considered a canonical set (according to Kothari et al. (2006), a canonical set consists of a small number of features that are as dissimilar as possible to each other, yet are representative of the entire functionality). Since one of the goals is to investigate the reusability of classes, the selection of features should not focus only on distinct functionalities.

The results from the application of the techniques/measures described in Sects. 2.a–2.e on the four case studies are presented in the following subsections in the same order.

3.1 Evolution of involved classes

In Fig. 6, we illustrate the number of classes which are involved during the execution of a certain feature, for all versions of JDeodorant, JFreeChart, Jmol and jEdit.

The experimental results for all projects and for almost all features indicate that the number of classes employed in the implementation of features is monotonically increasing as the projects evolve. A first striking observation is, for example, the fact that for writing and saving, a Java source code file with jEdit, over 300 classes, may be involved in the last examined version. From the reengineering perspective, if a feature’s implementation should be extended, adapted or simply analyzed, the maintainer might have to go through a large number of these classes in order to be able to modify the source code and maintain its external behavior, with profound impact on his productivity. The rate of increase in the involved classes in the implementation of each feature is not constant, and this might be caused by various reasons. For example, in project JFreeChart, an abrupt increase in the number of classes involved in the implementation of the selected features occurred between versions 1.0.10 and 1.0.11. According to the release notes, this might be related to a significant enhancement of functionality by introducing a new chart theming mechanism. The same observation holds for the transition from version 4.2.0 to 4.3.0 in project jEdit. The release notes revealed that in version 4.3.0, a significant number of enhancements, bug fixes and additions of new functionality have taken place.

The findings regarding the evolution of the number of methods that implement a selected feature are similar: the number of methods involved in each feature appears to be very high and increases with the passage of versions. For example, more than 550 methods might be invoked when drawing a Histogram chart in JFreeChart and close to 400 methods are involved in identifying Feature Envy code smells employing the JDeodorant tool. An

Table 1 Size characteristics of the examined versions/projects

JFreeChart

Measures	1.0.0	1.0.1	1.0.2	1.0.3	1.0.4	1.0.5	1.0.6	1.07	1.0.8	1.0.9	1.0.10	1.0.11	1.0.12	1.0.13
kLOC	126	126	130	134	138	142	146	157	157	158	161	168	170	177
NOC	465	466	478	493	502	505	516	540	540	540	546	561	563	587
kNOM	5.4	5.4	5.5	5.7	5.9	6.0	6.1	6.6	6.6	6.6	6.8	7.1	7.1	7.4

Jmol

Measures	11.0.0	11.0.2	11.2.0	11.2.3	11.2.5	11.2.10	11.2.14	11.4.1	11.4.6	11.6.1	11.6.10	11.6.20	11.6.27
kLOC	71	71.1	84	84.1	84	84.2	84.4	96	96.3	108	108.7	108.7	109
NOC	279	280	333	334	333	333	333	387	387	403	403	403	403
kNOM	5.3	5.3	6	6.04	6.04	6.04	6.05	6.5	6.53	7	7	7	7

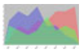
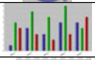
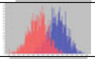
JDeodorant

Measures	1	2	3	4	5	6	7	8	9	10
kLOC	5.1	8.3	14.2	17.2	18.3	18.8	19.8	21.2	24.4	24.4
NOC	53	85	97	104	105	129	134	147	158	170
kNOM	0.51	0.68	0.90	0.990	1.00	1.07	1.12	1.20	1.35	1.46

jEdit

Measures	3.2.2	4.0.1	4.0.10	4.1.0	4.1.1	4.1.5	4.1.9	4.1.15	4.2.0	4.3.0	4.4.1	4.5.0
kLOC	63.9	77.6	81.6	88.2	87.2	93.2	100	101	107	130	123	125
NOC	256	295	295	319	312	330	344	344	367	457	453	469
kNOM	3.4	4.02	4.16	4.4	4.5	4.7	5	5.05	5.3	6.7	6.6	6.67

Table 2 Examined features for each project

Project	Feature Description			Project	Feature Description	
JFreeChart	**Pie Chart**		**Area Chart** 	Jmol	**Open mol file**	Opening of a mol file and rendering of the chemical structure in the screen.
	Bar Chart				**Open cif file**	Opening of cif file and rendering of the chemical structure in the screen.
	Gantt Chart		**Line Chart**		**Change view to Bottom**	Change 3D view side of the object to bottom angle
	Histogram				**Display molecule surface**	Displaying the Connolly surface of the molecule.
	XY Chart				**Export jpeg**	Exporting of the rendered molecule view in jpeg format.
JDeodorant	**Feature Envy**	Identification of methods suffering from feature envy code smell		jEdit	**Open a Java file**	Selection and opening of a java file through a File Chooser.
	Long Method	Identification of methods which are extremely long, complex and non-cohesive			**Select and Replace All**	Selection of a non-reserved word and replacement of all of its occurrences.
	Type Checking	Identification of conditional statements that select an execution path based on a specific state (lack of polymorphism)			**Add 2 markers in Java code**	Insertion of two markers in two inconsecutive lines of code.
	Move Method	Elimination of a selected feature envy code smell through move method refactoring application			**Navigate through markers**	Navigation through the added markers by using the appropriate menu option.
	Extract Method	Elimination of a selected long method code smell through extract method refactoring application			**HyperSearch**	Searching for a specific word. HyperSearch lists all occurrences of the search string in a floating window instead of locating the next match.
	Introduce Polymorphism	Elimination of a state checking code smell by introducing polymorphism			**Write Java code and save file**	Typing of a specific class and saving it as a Java file in order to enable the highlighting of Java reserved words.

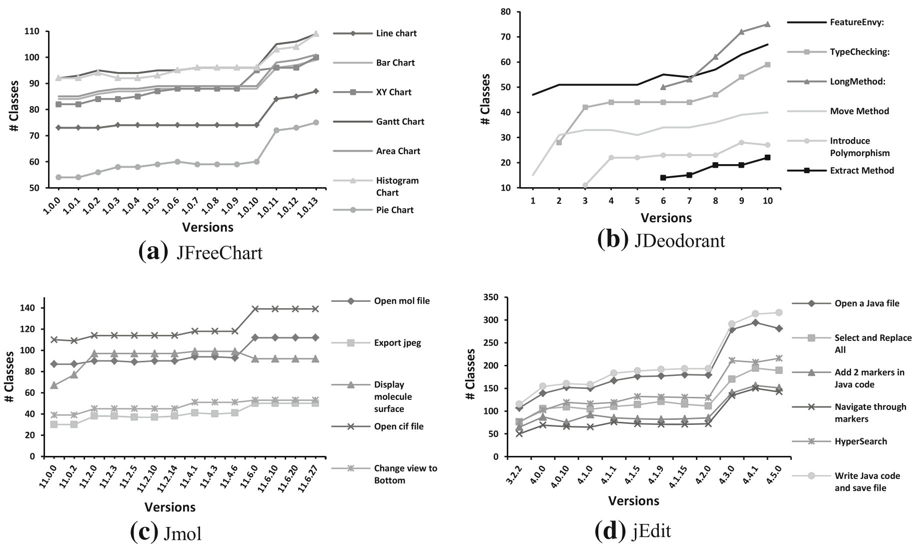


Fig. 6 Number of classes involved in the implementation of each feature, for **a** JFreeChart, **b** JDeodorant, **c** Jmol and **d** jEdit

impressive number of 1,467 methods are invoked in order for a newly typed Java source file to be saved in jEdit, while Jmol needs over 1,000 method invocations to read, render and display a chemical structure that is stored in a ".cif" file.

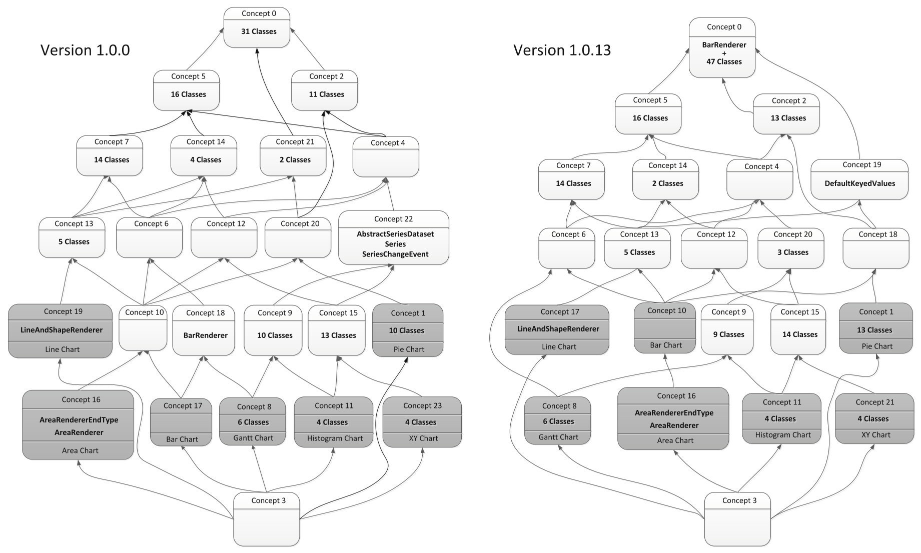


Fig. 7 Concept lattices for the first and last examined versions of JFreeChart

3.2 Concept lattices

The application of Formal Concept Analysis for the first and last version of project JFreeChart yielded the concept lattices shown in Fig. 7. At this point, it should be mentioned that one of the major drawbacks of concept lattices is that they do not scale well; consequently in Fig. 7, a reduced form of the sparse concept lattice has been employed, that is, class names are not shown except for the cases where it is necessary for our discussion. The highlighted nodes are concepts which introduce the examined features and thus can serve as the basis for observing the evolution in the number of classes involved in each feature.

According to the semantics of concept lattices applied in our case, the following pieces of information can be derived from the observation of the graphs (Eisenbarth et al. 2003). Their use can be extended for the interpretation of the evolution in the scattering of features and the reuse of components:

- A feature *f* requires all classes at and above the node at which the feature appears in the sparse lattice representation. For example, feature Line Chart (Concept_19) requires 73 classes in version 1.0.0, which can be found by traversing upward all paths starting from Concept_19 and ending at the top node. In version 1.0.13, the number of classes involved in the implementation of Line Chart increased to 87.
- A class *c* is required for all features at and below the node at which the class appears in the sparse lattice representation. For example, class BarRenderer in version 1.0.0 (Concept_18) is only involved in the implementation of Bar Chart and Gantt Chart, indicating a relatively low degree of reuse for the class. On the other hand, the same class appears in the top node of the concept lattice in version 1.0.13, implying that this class contributes to the implementation of all features, exhibiting a tremendous increase in its reuse.

- A class *c* is specific to exactly one feature *f*, if *f* is the only feature on all paths from the node at which *c* is introduced to the bottom element. For example, in version 1.0.0, the classes which are only involved in the implementation of feature Pie Chart (Concept_1) are 10, while the number of unique classes for the same feature in version 1.0.13 has risen up to 13.
- Classes jointly required for *n* features $f_1, f_2, \ldots, f_n$ are classes belonging to concepts which lie on the intersection of all paths from the node at which features $f_1, f_2, \ldots, f_n$ are introduced, to the top element. For example, features Gantt Chart (Concept_8), Histogram Chart (Concept_11) and XY Chart (Concept_23) in version 1.0.0 share classes AbstractSeriesDataSet, Series, SeriesChangeEvent (lying at Concept_22) as well as all classes at concepts 5, 2 and 0. In total, 61 classes are commonly used in the implementation of these three features in the first version of JFreeChart. From the examination of the concept lattice of the last version, it can be found that the number of common classes increases to 80.
- Classes required for all functionalities lie at the top element (Concept_0). For version 1.0.0, 31 classes are employed in all examined features, while in version 1.0.13, the number of common classes increases to 48.

3.3 Distribution of methods

Figure 8 displays the distribution of methods among the classes involved in the implementation of a feature for the first and last version of all examined projects. More

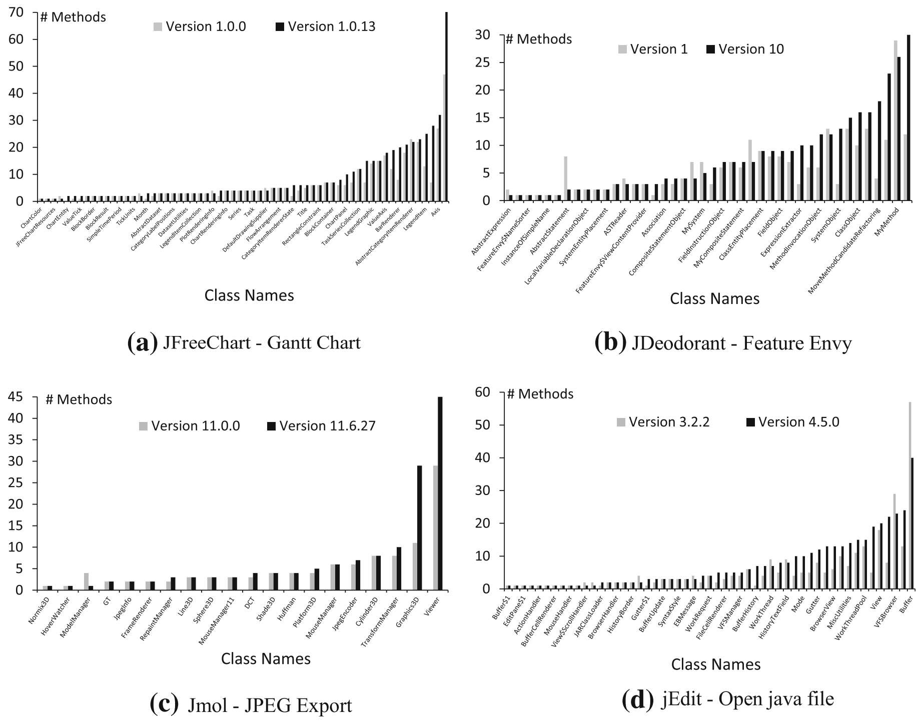


Fig. 8 Distribution of methods over classes for selected features

specifically, Fig. 8a depicts the distribution of methods over classes for a Gantt Chart creation with JFreeChart, 8(b) illustrates the same distribution for Feature Envy functionality of JDeodorant, while 8(c) and 8(d) correspond to the exporting of a jpeg image in Jmol and the opening of a Java file in jEdit, respectively. To understand whether this distribution remains unchanged as the system evolves or not, the number of methods that are used in the first (light bars) and the last version (dark bars) is shown for each of the involved classes. (The figures display only the classes that exist in the both first and last versions of the examined projects).

An observation that can be made for all projects is the skewed nature of the distribution of methods over classes. For example, in 8(a), it can be observed that most of the involved classes host less than 10 methods contributing to the examined functionality, while a relatively small number of classes host over 20 involved methods. The same observation holds for 8(c) and 8(d), while in 8(b), this phenomenon still exists but it is less intense. A characteristic example of the unbalanced distribution of class responsibilities is the class Buffer that supports the opening of a Java file in jEdit 3.2.2 with 57 methods, and the class CategoryPlot from JFreeChart 1.0.0, which supports the creation of a Gantt Chart with 47 methods.

A second remark is related to the methods that are introduced during software evolution. From the figures, it becomes apparent that classes which already hold an increased number of methods act as attractors to the newly inserted methods, a phenomenon similar to the *rich-get-richer* rule underlying preferential attachment (Barabasi et al. 2000). For example, in JFreeChart, 20 % of the total number of additional methods (121 methods, comparing the first and the last examined version) have been added to a single class (class CategoryPlot contributed to the Gantt Chart functionality 47 methods in the first version and 71 methods in the last one). An exception to this phenomenon is class Buffer in jEdit, where despite the fact that it held the majority of methods in the first examined version (57), this number decreased to 40 in the last examined version.

The aforementioned observations imply phenomena which could be rather harmless. For example, the overconcentration of methods in a single class among those implementing a feature might be due to the nature of the involved functionality. On the other hand, highly skewed distributions of methods among the classes involved in the implementation of certain functionalities, which become even more skewed as the systems evolve, could represent inefficiencies of the initial architecture which might go unnoticed by other means, such as metric values or design flaws. In other words, this form of preferential attachment, where new methods are attached to classes that have already a large number of methods contributing to the same feature, might lead to serious maintenance issues. The evolution of these distributions is studied in a more formal manner in the next subsection employing the Gini coefficient.

3.4 Gini coefficient and Degree of Scattering

The evolution of the Gini coefficient over the versions of all examined systems is shown in Fig. 9, for selected features. The values range from 0.37 for the second version of project JDeodorant (feature Type Checking) to 0.64 for version 4.3.0 of project jEdit (feature Open Java File). While an absolute value for the inequality in a distribution might be difficult to interpret, its tendency over time might be informative. As it can be observed the value of the Gini coefficient is generally increasing with the passage of software versions, indicating that the distribution of methods among the classes involved in the corresponding feature becomes more unbalanced over time. As already explained, this means that classes

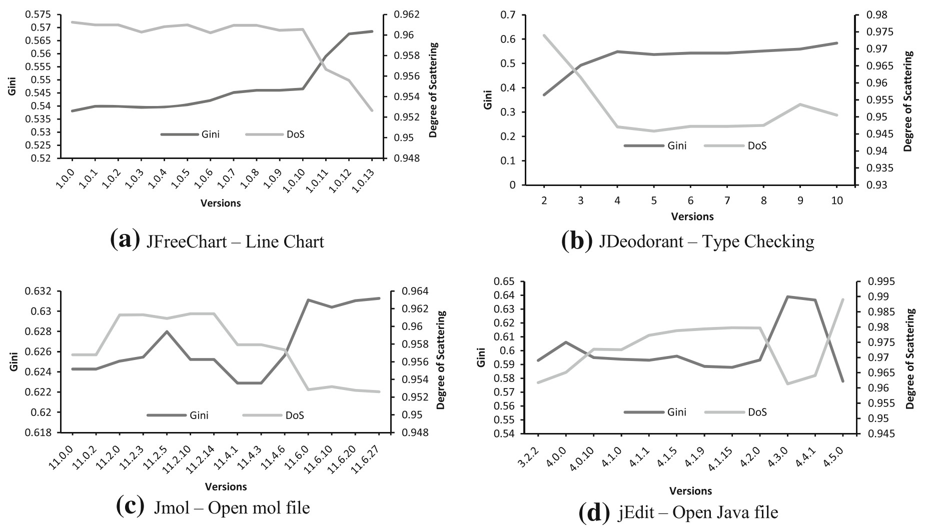


Fig. 9 Co-evolution of the Gini coefficient and Degree of Scattering for selected features

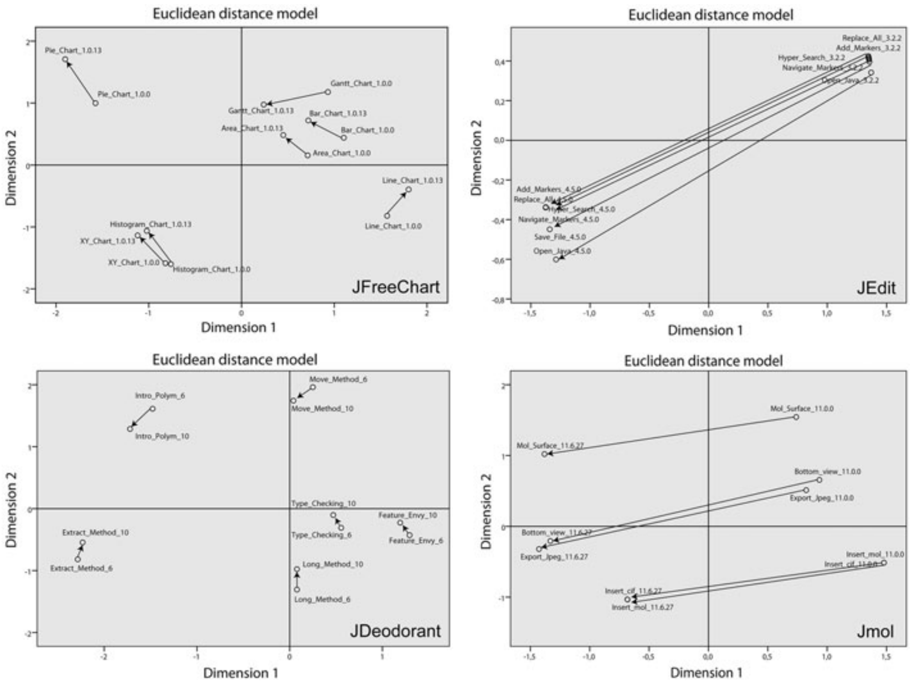


Fig. 10 Multidimensional Scaling. *The initial version for JDeodorant is v6 since this is the first version in which all the examined features are present

with a large share on the total functionality (in terms of methods) attract even more methods as software evolves, becoming a sort of “God” classes in the context of the examined feature.

Figure 9 also displays the evolution of the Degree of Scattering. As it can be readily observed, trends of Gini and DoS are quite opposite, and in most cases, an increase in the value of Gini can be matched to a decrease in DoS and vice versa. Intuitively, this observation makes sense by considering the nature of the two metrics. Gini coefficient is analogous to the inequality of a distribution and increases if this inequality deteriorates (i.e., classes that already hold many methods, attract more new methods than the classes with fewer methods), while Degree of Scattering is analogous to the diffusion of methods across classes, and the more diffused the methods become, the higher the value of DoS is.

3.5 Multidimensional scaling for feature distance visualization

Figure 10 illustrates the output of multidimensional scaling for two versions (initial and last one) for all four examined projects, employing as distance the aforementioned Simpson measure. The axes of the generated two-dimensional maps could be interpreted as follows: As we move along Dimension 1 from the right to the left, in projects jEdit and Jmol, it is clearly evident that any variations are due to the passage of software versions. On the other hand, differences along Dimension 2 can be attributed to variations in functionality. Points that come closer indicate an increase in method reuse, while points that diverge indicate the opposite. Similar observations hold for JFreeChart and JDeodorant but are less striking.

The MDS output for JFreeChart (top left corner) depicts three primary clusters of features located at the upper left, lower left and right areas of the diagram. The clusters of features which can be identified based on their distances are rather reasonable, considering the underlying data structures on top of which they are built. Line Chart, Area Chart, Bar Chart and Gantt Chart functionalities are all dependent on a CategoryDataset class or subtypes of it. Histogram and XY Chart functionalities employ the XYDataSet data structure, while the Pie Chart is rather independent, using the PieDataSet structure.

Concerning the overall evolution of the system, it can be observed that rather small changes occurred in the distances between the features from the first to the last version. A more careful examination can reveal, for example, that the distance between the pair of features Line and Pie Chart, or Histogram and XY Chart, increased with the passage of generations. For example, Histogram and XY Chart are extremely close to each other in version 1.0.0, since they share 365 methods out of 390 methods contained in the XY Chart, which is the “smaller” of the two features. In version 1.0.13, the number of common methods raised to 492, followed by a concurrent increase in the “smaller” feature which remains the XY Chart with 520 methods, leading to a slightly higher distance between the two features. The overall evolution of similarity, as the arrows depict, points that the examined features are becoming less similar by employing fewer common methods.

From the Euclidean distance model for JDeodorant (bottom left corner), the most striking observation concerning clusters that can be identified visually is the cluster containing features Feature Envy, Long Method and Type Checking, at the lower right area of the diagram. These features correspond to code smell identification functionalities which share a number of methods in their implementation and are rather distinct from the other three features corresponding to refactoring application functionalities. Concerning the overall evolution, an improvement in the design properties can be observed, since many of the features appear to converge, in the sense that the corresponding points in the diagram move slightly toward the center of the diagram as the system evolves, implying an increase in the degree of reuse.

The case of jEdit (top right corner) also presents an interesting evolution. It appears from the MDS chart that all features are relatively close to each other, indicating a large degree of reuse among features. This is true for both the first and the last examined versions. As an example, in version 3.2.2, features “Add Markers” and “Navigate Markers” have 50 classes in common out of the 65 and 50 classes of the first and second features, respectively. The same holds for version 4.5.0 where the two features share 139 classes out of 151 and 142 classes, respectively. On the other hand, there is a large displacement between the dots of the first and last version implying limited reuse between the same features as software evolved. For example, feature “Add Markers” in version 3.2.2 and the same feature in the last version share only 30 out of the initial 65 classes, and on top of that, 86 new classes have been added (i.e., 30 out of the 151 classes of the last version).

A similar phenomenon is apparent on the MDS chart for Jmol (bottom right corner). The extent of reuse among features remains relatively stable across versions, whereas the features of the first version share limited classes to the same features of the last version, implying low reuse and the addition of a large number of new classes to each feature.

4 Impact of refactorings on the distribution of feature implementation

The application of preventive maintenance activities such as refactorings is rarely considering the impact on feature scattering, whereas the relocation of methods, the creation of new classes and methods definitely affect the implementation of features. For example, let us consider that a method, contributing to the implementation of a particular feature, is moved from a source class *A* to a target class *B* after applying the Move Method refactoring. In the extreme case where *B* was not involved at all in the implementation of the feature prior to the refactoring, moving the method will increase feature scattering in the sense that a larger number of classes will be involved. On the other hand, if both classes are part of the feature implementation and class *A* contains a larger number of involved methods, moving the method will lead to a more balanced distribution of the functionality across the classes, reflected on a decrease in the Gini coefficient.

To evaluate the impact that selected refactorings have on feature scattering, we conducted an experiment by applying consecutive refactorings of the same type for a specific feature of a given system. In particular, we selected one version of each of the examined open-source systems and employed JDeodorant (2011) in order to identify refactoring opportunities for Extract Class, Extract Method and Move Method refactorings (Fowler 1999) that affect classes which are involved in the feature of interest. The Gini coefficient and the DoS have been measured before and after the application of each refactoring, allowing us to assess whether the refactoring improved scattering or not. Figure 11 illustrates the successive values of the Gini coefficient and DoS, resulting from the application of all identified refactorings for a selected feature and for all four examined systems.

The effect of the Extract Class refactoring depends on the number of methods extracted as a new class as well as on the number of methods involved in the feature under study in the original class. For example, the application of Extract Class refactorings in projects JDeodorant and jEdit appears to have a positive impact on the Gini coefficient by relieving in most cases heavily loaded classes (i.e., classes with a large share on the total number of involved methods) from several methods which are moved to the extracted class. On the other hand, in project Jmol, all Extract Class refactorings moved a very small number of

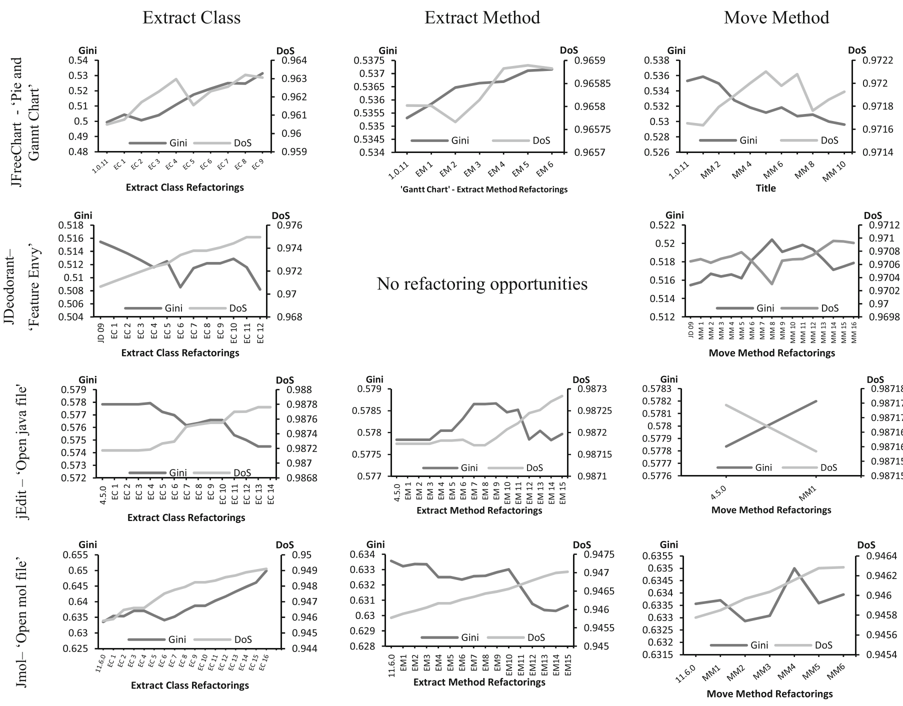


Fig. 11 Impact of refactorings on Gini coefficient and Degree of Scattering

methods to the new class (usually one), leading to the creation of an additional "poor" class, that is, a class with very small participation in the implementation of a feature and thus deteriorating even more the distribution of functionality. According to these observations, if the goal is to achieve a balanced distribution of methods, Extract Class refactorings should be applied only if a substantial amount of functionality is to be moved to the new class.

Extracting a method from an existing one (which is involved in the implementation of a feature) will affect the distribution of methods and as a result the Gini coefficient, depending on whether the class hosting the original method resides in a class that has a large or small share on the total number of methods implementing a feature. The addition of the extracted method to a class that had a relatively small number of involved methods reduced the Gini coefficient in all cases, while the upward trends are due to the extraction of a new method in a class that was already "rich" in terms of the number of involved methods. As a guideline, from the perspective of feature functionality distribution, one could suggest to avoid performing the Extract Method refactoring for classes that already have a large share of the total number of involved methods.

Moving a method to another class will improve (deteriorate) the distribution of methods involved in a certain feature and consequently decrease (increase) the Gini coefficient, in case the target class to which the method is moved is a "poor" ("rich") class in terms of method concentration. An exception is the move of a method to a class that was not involved in the implementation of the examined feature prior to the refactoring. In this case, the addition of another "poor" class deteriorates the distribution of methods. This interpretation explains all variations in the charts of Fig. 11 for Move Method refactorings.

The aforementioned guidelines do not aim to substitute the already existing criteria or heuristics for assessing the impact of a refactoring. For example, the application of an Extract Method refactoring might be valuable, in the case of a large, complex and non-cohesive method, regardless of the effect on the Gini coefficient. However, these rules might be considered in parallel since in most cases, the suggestions are in line with common sense for achieving better design quality. For example, one would rarely perform an Extract Class refactoring if the concept of the extracted class is too limited (i.e., if the number of methods in the new class is very small), and this is in absolute agreement to the observation made earlier regarding the distribution of methods implementing a feature.

5 Threats to validity

In this paper, we have introduced a set of techniques and measures in order to investigate the evolution of feature scattering and then presented results for four case studies. The following threats to construct and internal validity (Wohlin et al. 2000) can be identified. Since the application on the four case studies has been performed as an illustration of how the proposed techniques/measures can be employed, threats to external validity are not present.

The entire process is based on the assumption that the number of methods involved in the implementation of a given feature constitutes a valid measure for the quantification of feature scattering. This could potentially impose a threat to construct validity which deals with how well the selected measures or tests can stand in for the concepts of interest. According to the taxonomy by Dit et al. (2011) regarding feature location techniques, a feature's implementation can be traced down to the following measures: (1) files/classes, (2) methods/functions, (3) statements and (4) non-source code artifacts. It appears that methods as an output of feature identification are used much more frequently than any other measure, and in particular, it has been used in 39 out of 45 feature location approaches. As a result, the selection of methods appears to be the most reliable and sound choice.

Regarding the internal validity of the study (i.e., the factors that might affect the phenomena that we are trying to investigate), an identified threat is related to the presence of other features which might have not been included in our analysis. However, this threat is only valid for the investigation of reusability among features and its evolution by means of multidimensional scaling. The reason is that other ignored features might be interleaved with the features that have been the focus of our study. For example, a feature that exhibits relatively low reuse with other selected features might share a large number of classes and methods with a feature that has been omitted. To mitigate this threat, anyone who aims at analyzing the degree of reuse among features should be mindful to select all possible features which are conceptually or functionally similar. On the other hand, for the techniques and measures presented in Sects. 2.a–2.d, this threat is not present since the employed measures are not affected by the existence of other features.

6 Related work

The primary challenge in the field of feature to source code mapping is the correct identification of software components implementing a certain feature. Feature identification approaches can be categorized as static, dynamic and hybrid, depending on the nature of the processed information.

Static techniques are mainly based on various Information Retrieval (IR) methods that involve textual matching of terms in the project's requirement documentation that describe a feature, to source code identifiers on the premise that they have meaningful names (Antoniol et al. 2002; Conejero et al. 2009). IR models that are usually employed are Vector Space Model (VSM), Latent Semantic Indexing (LSI) and Probabilistic Network (PN) (Zou et al. 2009). The first steps on automated static feature location were made by Biggerstaff et al. (1994) who have built a tool that locates identifiers in source code and clusters them in order to facilitate feature location. Antoniol et al. (2002) proposed a method that employs both Probabilistic Network and Vector Space Model, in order to analyze the mnemonics that serve as identifiers in source code and use them to associate high-level concepts with program concepts. Marcus et al. (2003, 2004) employed Latent Semantic Indexing in order to locate concepts in source code, while, for the same purpose, Shepherd et al. (2007) have made use of Natural Language Processing, a method that originates from Artificial Intelligence. In some approaches, IR methods are assisted by different techniques, as in the work of Poshyvanyk et al. (2007), who used Formal Concept Analysis in order to refine the mapping tables that resulted from Latent Semantic Indexing, or Zhao et al. (2003, 2004), who presented a non-interactive method that also employs a structural analysis process named branch-reserving call graph, which is a call graph with the addition of branch information.

Dynamic approaches entail the execution of a number of test cases that exercise the desired feature in order to enable the capturing of the execution trace and to determine the software modules that are involved in the feature's implementation. Wilde and Scully (1995) presented "Software Reconnaissance," a method which discovers software modules that implement a particular feature. Dynamic execution tracing (slicing) has also been adopted by Wong et al. (1999) who identify source code elements that implement a specific feature or group of features.

The combination of static and dynamic methods has been recognized as an approach that considerably improves the effectiveness of feature identification. In the hybrid approach of Eisenbarth et al. (2003) and Koschke et al. (2005), features are invoked based on execution scenarios, in order to collect dynamic information. Aided by Formal Concept Analysis, the proposed methodologies create concept lattices whose interpretation combined by a static dependency graph lead to a mapping between features and computational units. Poshyvanyk et al. (2007) and Liu et al. (2007) propose methods that locate features by exploiting the advantages of two distinct methods, namely Latent Semantic Indexing and Probabilistic Ranking of entities that came of scenario executions.

A study on the evolution of features and their implementation has been performed by Greevy et al. (2006), who categorize software entities according to the level of participation in features. Furthermore, they investigate the changes in categorization during the evolution of software.

A set of useful metrics for the analysis of how features are implemented in source code has been proposed by Wong et al. (Wong et al. 2000), who quantified the closeness between a feature and a software component involved in its implementation. Eaddy et al. (2007, 2008) evolved and extended Wong's metrics by investigating the consequences of scattered and tangled concern implementation (crosscutting concerns) in the quality of programs, in terms of defects. They examined the correlation between the number of bugs and metrics that quantify the scattering of concerns in code (e.g., Degree of Scattering, DoS) at class and method level. Their results indicate a relatively strong correlation between DoS and number of defects. Finally, Conejero et al. (2009) have also employed crosscutting metrics in order to predict possible software instability in early development artifacts such as requirement descriptions.

7 Conclusions

Significant effort in the field of requirements traceability has been devoted to the identification of features in source code. Locating where features are implemented is important for understanding an existing software system, and moreover, it can reveal possible problems, such as extended scattering in the implementation of a feature. An even more important issue that deteriorates system maintainability arises when feature scattering increases as software systems evolve. In this paper, we have proposed a set of techniques for the analysis of the evolution in feature scattering, based on the classes and methods involved in the implementation of high-level, distinct and observable pieces of functionality. In particular, we employed Formal Concept Analysis to investigate the evolution of feature implementation, the Gini coefficient as a measure of the distribution of methods over the involved classes, an appropriate similarity measure along with multidimensional scaling to study the evolution of the reuse among methods contributing to a feature and the impact of selected refactorings on feature scattering.

The proposed analyses have been applied on several versions of four open-source projects. Based on the results, the applied techniques appear to be promising since they allow software stakeholders to assess visually the evolution in feature scattering and gain insight into the associated implications. In particular, the obtained visualizations facilitate the study of feature spreading in terms of the number of classes and methods. A more in-depth analysis can be performed by examining the distribution of methods contributing to the implementation of the examined features and the use of the Gini coefficient to determine whether this distribution tends to become more unbalanced over time. Since similar features usually rely on common methods, the investigation of feature scattering should consider the corresponding degree of method reuse among features, as we have shown by means of MDS charts. Finally, we have studied the impact of three widely used refactorings on feature scattering and concluded that generic rules can be employed to assess the circumstances under which a refactoring improves or deteriorates the distribution of feature functionality.

Acknowledgments This research has been co-financed by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis—Athens University of Economics and Business—SOFTWARE ENGINEERING RESEARCH PLATFORM.

References

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering, 28*(10), 970–983.

Barabasi, A. L., Albert, R., Jeong, H., & Bianconi, G. (2000). Power-law distribution of the World Wide Web. *Science, 287*, 2115.

Bartholomew, D.J., Steele, F., Moustaki, I., and Galbraith, J. (2008). *The Analysis and Interpretation of Multivariate Data for Social Scientists*, Chapman and Hall/CRC.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. E. (1994). Program understanding and the concept assignment problem. *Communications of the ACM, 37*(5), 72–82.

Chen, C. C., Hardle, W., Unwin, A., Cox, M., & Cox, T. F. (2008). *Handbook of data visualization*. Heidelberg: Springer Berlin Heidelberg.

Choi, S–. S., Cha, S.-H., & Tappert, C. C. (2010). A survey of Binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics, 8*(1), 43–48.

Conejero, J. M., Figueiredo, E., Garcia, A., Hernández, J., & Jurado, E. (2009). Early crosscutting metrics as predictors of software instability. *Objects, Components, Models and Patterns. Lecture Notes in Business Information Processing, 33*(3), 136–156.

Dit, B., Revelle, M., Gethers, M., & Poshyvanyk, D. (2011). Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software Maintenance and Evolution: Research and Practice*, published online: 28 November 2011, Early Access.

Eaddy, M., Aho, A. V., & Murphy, G.C. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Proceedings of the Workshop Assessment of Contemporary Modularization Techniques*.

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., et al. (2008). Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering, 34*(4), 497–515.

Eisenbarth, T., Koschke, R., & Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering, 29*(3), 210–224.

Filho, F.C., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., & Rubira, C. M. F. (2006). Exceptions and aspects: The devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 152–162.

Fisher, M., & Gall, H. (2003). MDS-Views: Visualizing problem report data of large scale software using multidimensional scaling. In *Proceedingsof the Large-scale Industrial Software Evolution Workshop (ICSM 2013)*, 110–122.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring improving the design of existing code*. MA: Addison-Wesley.

Ganter, B., & Wille, R. (1996). *Formal concept analysis*. Berlin: Springer-Verlag.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., & von Staa, A. (2005). Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, 3–14.

Gibbs, C., Robin Liu, C., & Coady, Y. (2005). Sustainable system infrastructure and big band evolution: Can aspects keep pace?. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 241–261.

Gini, C. (1921). Measurement of inequality of incomes. *The Economic Journal, 31*(121), 124–126.

Goeminne, M., & Mens, T. (2011). Evidence for the pareto principle in open source software activity. In the *Joint Porceedings of the 1st International workshop on Model Driven Software Maintenance and 5th International Workshop on Software Quality and Maintainability*, 74–82.

Gotel, O. C. Z., & Finkelstein, C. W. (1994). An Analysis of the requirements traceability problem. In *Proceedings of the 1st International Conference on Requirements Engineering*, 94–101.

Greenwood P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., & Rashid, A. (2007). On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, 176–200.

Greevy, O., Ducasse, S., & Girba, T. (2006). Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice, 18*(6), 425–456.

JProfiler: An all-purpose Java profiling suite. http://www.ej-technologies.com/products/jprofiler/overview.html, October 2011.

JDeodorant, http://www.jdeodorant.com, October 2011.

jEdit—programmer's text editor, http://www.jedit.org, August 2012.

JFreeChart, http://www.jfree.org/jfreechart, October 2011.

Jmol: An open-source Java viewer for chemical structures in 3D. http://www.jmol.org, August 2012.

Koschke, R., & Quante, J. (2005). On dynamic feature location. In *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, 86–95.

Kothari, J., Denton, T., Mancoridis, S., & Shokoufandeh, A. (2006). On computing the canonical features of software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering*, 93–102.

Kuhn, A., Loretan, P., Nierstrasz, O. (2008). Consistent Layout for Thematic Software Maps. In *Proceedings of the 15th Working Conference on Reverse Engineering*, 209–218.

Liu, D., Marcus, A., Poshyvanyk, D., & Rajlich, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd International Conference on Automated Software Engineering,* 234–243.

Lorenz, M. O. (1905). Methods of measuring the concentration of wealth. *Publications of the American Statistical Association, 9*(70), 209–219.

Marcus, A., & Maletic, J. I. (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, 125–136.

Marcus, A., Sergeyev, A., Rajlich, V., & Maletic, J. I. (2004). An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, 214–223.

Naseem, R., Maqbool, O., & Muhammad, D. (2011). Improved similarity measures for software clustering. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 45–54.

Poshyvanyk, D., & Marcus, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, 37–48.

Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G., & Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering, 33*(6), 420–432.

Revell, M., Gethers, M., & Poshyvanyk, D. (2011). Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering, 16*(6), 773–811.

Riel, A. J. (1996). *Object-oriented design heuristics*. MA: Addison-Wesley.

Robillard, M. P., & Murphy, G. C. (2007). Representing Concerns in Source Code. *ACM Transactions on Software Engineering and Methodology*, *16*(1), 1–38.

Sharp, A. (1997). *Smalltalk by Example: The developers Guide*. Mcgraw-Hill.

Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., & Vijay-Shanker, K. (2007). Using natural language program analysis to locate and understand action-oriented concerns, In *Proceedings of the 6th International Conference on Aspect-Oriented software development*, 212–224.

Simpson, G. G. (1960). Notes on the measurement of faunal resemblance. *American Journal of Science, 258*(A), 300–311.

Singh, K. (2007). *Quantitative social research methods*. California: Sage Publications.

Trifu, M. (2010). *Tool-supported identification of functional concerns in object-oriented code.* PhD thesis, Karlsruhe Institute of Technology.

Vasa, R., Lumpe, M., Branch, P., & Nierstrasz, O. (2009). Comparative Analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th International Conference on Software Maintenance*, 179–188.

Wilde, N., & Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice, 7*, 49–62.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2000). *Experimentation in software engineering. An introduction*. The Netherlands: Kluwer Academic Publishers.

Wong, W. E., Gokhale, S. S., Horgan, J. R., & Trivedi, K. S. (1999). Locating program features using execution slices. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*, 194–203.

Wong, W. E., Gokhale, S. S., & Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software—Special Issue on Software Maintenance, 54*(2), 87–98.

Zhao,W., Zhang, L.,Liu, Y., Luo, J., & Sun, J. (2003). Understanding How the Requirements Are Implemented in Source Code. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, 68–77.

Zhao, W., Zhang, L.,Liu, Y., Sun, J., & Yang, F.(2004). SNIAFL: Towards a Static Non-Interactive Approach to Feature Location. In *Proceedings of the 26th International Conference on Software Engineering*, 293–303.

Zou, X., Settimi, R., & Cleland-Huang, J. (2009). Improving automated requirements trace retrieval: A study of term-based enhancement methods. *Empirical Software Engineering, 15*, 119–146.

Author Biographies

Theodore Chaikalis Received the B.Sc. and M.Sc. degrees in Applied Informatics from the University of Macedonia, in 2007 and 2009, respectively. He is currently working toward the Ph.D. degree in the Department of Applied Informatics at the University of Macedonia under the supervision of Dr. Alexander Chatzigeorgiou. His research interests include object-oriented design and quality metrics, exploitation of graph theory in software engineering and software evolution simulation. He is a student member of the IEEE .

Alexander Chatzigeorgiou Is an associate professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. His research interests include object-oriented design, software maintenance and evolution. He is a member of the IEEE .

Georgina Examiliotou Graduated from the department of Applied Informatics, University of Macedonia, Thessaloniki, Greece in 2009. She is currently working toward her master thesis for the M.Sc. in Computer Systems at the same department. From 2010 to 2011, she worked at the General Hospital of Komotini, Greece, for the analysis and design of IT applications as well as a server administrator. Her research interests include object-oriented design, software maintenance and feature scattering evolution .