Investigating the Evolution of Feature Scattering

Theodore Chaikalis and Alexander Chatzigeorgiou
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
Email: {chaikalis, achat}@uom.gr

***Abstract*— The identification of software components that implement a certain feature is vital for understanding and reverse engineering an existing software system. Requirements traceability allows the investigation of scattering and tangling requirements which hinder significantly software maintenance. Even worse than feature scattering in a number of modules, is that this diffusion might deteriorate as software evolves. In this paper we attempt to shed light on the following questions: whether scattering in the requirements implementation increases over time, whether the methods implementing a specific feature are distributed unequally and whether scattering can be justified by the reuse among features at method level. Emphasis is placed on the evolution of the examined measures and phenomena, rather than on static snapshots of the systems, to highlight that design problems in the choice of classes and the allocation of methods might become increasingly apparent with the passage of software versions. We illustrate the applicability of the proposed analyses on case studies concerning several versions of two software systems.**

Keywords- Feature identification; feature scattering; program understanding; requirements traceability; software evolution

I. INTRODUCTION

Software maintenance relies heavily on understanding existing systems and particularly on analyzing how certain features are implemented in the source code [1], [3], [8], [11], [35]. This challenge has motivated a large number of studies in software engineering aiming at linking software requirements with source code components, a task which is non-trivial since the required information is inefficiently documented [1]. This field is usually known as Requirements Traceability which according to Gotel et al. [16] is the ability to follow a requirement from its specification through its deployment in code, in both a forward and backward direction. Various researchers also refer to the corresponding activities as Concern [9], [35] Concept [3], or Feature Location [11] since the goal is to identify the source code elements implementing a certain functional requirement.

According to several studies [6], [10], [29] feature scattering and coupling leads to increased fault proneness. Furthermore, features whose implementation is scattered throughout the source code hinder significantly the maintenance and evolution of software systems. As an illustrative example, Robillard and Murphy [31] 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. It should be borne in mind that the problem of feature scattering in a number of modules might deteriorate as the software evolves due to software ageing [26]. In other words, the "diffusion" of a certain requirement in source code might increase with the passage of software versions. This can certainly be attributed to the enhancement of functionality over time, but in some cases it may become severe with tens of classes and hundreds of methods participating in the implementation of a single feature.

The need to continuously monitor software quality, 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, the distribution of methods implementing a specific feature in the involved classes and the reuse among features at the method level. In the following we adopt the term *feature* as defined by Eisenbarth et al. [11] which refers to a distinct, observable, unit of behavior of a system that can be exercised by the end user.

The data 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 over time?
- Are the methods contributing to the implementation of a feature uniformly distributed among the involved classes?
- Does the distribution of methods become 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?

The latter two questions are 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.

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 two open-source projects. 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.

Most previous studies in the field of requirements traceability focus on establishing a sound and accurate approach for identifying software components related to a feature or concept [42]. In this paper we emphasize the need to study the evolution of the requirements scattering. Moreover, we 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 [15], [39].

The tools and techniques that we employed for the proposed analyses are borrowed from a number of diverse fields: Formal Concept Analysis, which has initially been used by Eisenbarth et al. [11] for the identification of features in source code, is employed to formally investigate and visualize the evolution of feature scattering. The Gini coefficient [14], 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. A measure of similarity proposed in paleontology to illustrate part-whole relations, is employed to study the evolution of the reuse among methods contributing to a feature. Finally, Multi-dimensional scaling, a widely used tool for data visualization, is employed to study the evolution of the similarity between features based on their common methods.

The rest of the paper is organized as follows: In Section II we describe the experimental set-up and the characteristics of the projects that have been analyzed. An investigation of the modules that are involved in the implementation of features, the way that methods are distributed among classes as well as the evolution of this distribution, is presented in Section III. In Section IV we employ a measure that quantifies the distance among features to capture their degree of reuse and a method for visualizing how these distances evolve. Threats to Validity are analyzed in Section V while Related Work is discussed in Section VI. Finally we conclude in Section VII.

II. Context of the Proposed Analyses

For the investigation of classes and methods involved in the implementation of a specific functionality, dynamic analysis employing a java profiler has been performed on two projects, namely JFreeChart and JDeodorant. JFreeChart is an open-source chart library [18] 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 [17]. It has been constantly evolving for more than four years as a project of the Computational Systems and Software Engineering Laboratory at the Department of Applied Informatics, University of Macedonia, Greece. The analysis employed 14 and 10 versions of JFreeChart and JDeodorant, respectively. The evolution of size characteristics (lines of code, number of classes and number of methods) for the examined versions of both projects is shown in Table I.

Since no framework for the integration and synchronization of software artifacts has been used for the development of the examined projects, a formal documentation of requirements mapping to source code is not available. As a result, to perform an investigation of the relationship between functional requirements and software artifacts, the mappings had to be extracted from the executable code. This problem is analogous to the identification of features in source code as discussed by [11], [19]. As already mentioned, features refer to well-defined functionalities which produce a useful and observable output to the end user. For example, a feature of JFreeChart, takes data points to be illustrated as inputs and creates a pie chart in a new window. For the analysis of JFreeChart, seven features have been selected while for JDeodorant the examined features are six. Table II includes a graphical representation for each of the seven features that have been selected from JFreeChart, while Table III briefly outlines the features of JDeodorant. It should be mentioned that the selected features cannot be considered a canonical set (according to Kothari et al. [20] 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). However, since one of the goals is to investigate the reusability of classes, the selection of features should not focus only on distinct functionalities.

TABLE I. 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

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
NOM	513	680	901	990	1004	1074	1122	1201	1358	1466

TABLE II. Examined Features of JFreeChart

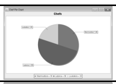
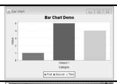
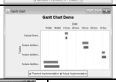
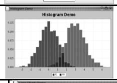
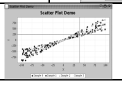
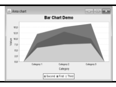
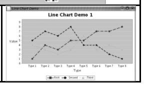
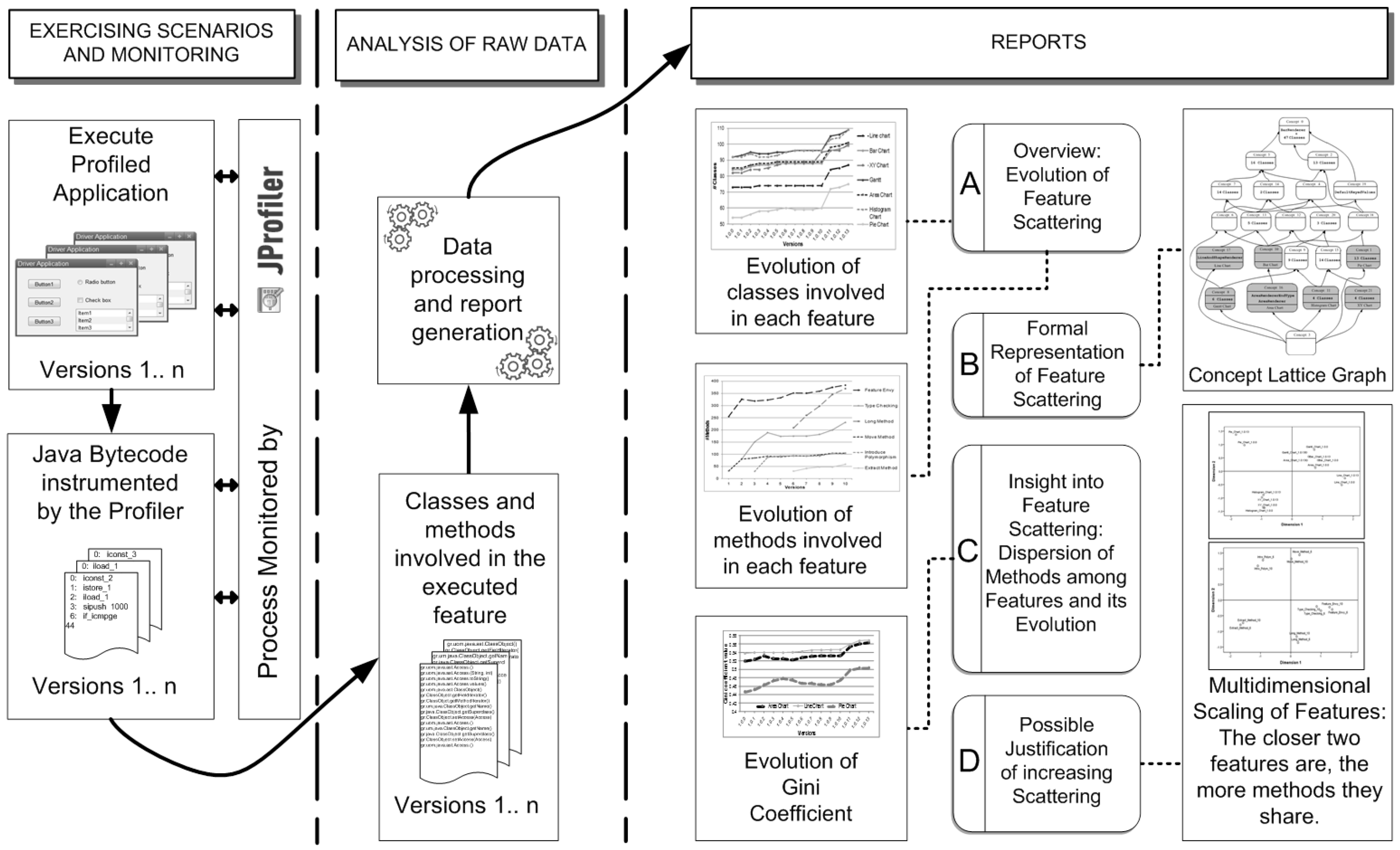
Pie Chart Creation		Bar Chart Creation			
Gantt Chart Creation		Histogram Chart Creation			
XY Chart Creation		Area Chart Creation		Line Chart Creation	

TABLE III. Examined Features of JDeodorant

Feature	Description
Feature Envy	Identification of methods suffering from feature envy code smell
Long Method	Identification of methods which are extremely long, complex and non-cohesive
Type Checking	Identification of conditional statements that select an execution path based on a specific state (lack of polymorphism)
Move Method	Elimination of a selected feature envy code smell through move method refactoring application
Extract Method	Elimination of a selected long method code smell through extract method refactoring application
Introduce Polymorphism	Elimination of a state checking code smell by introducing polymorphism


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.

*The data for each phase of the depicted process are available from http://java.uom.gr/~chaikalis including the driver programs for exercising scenarios, the extracted methods, the data processing and report generation tool as well as the generated reports.

Figure 1: Data collection and analysis process

The functionality which is the target of the analysis is triggered through a driver program that executes an appropriate scenario in analogy to the test cases employed in [37]. For example, a scenario for profiling the Create Line Chart functionality includes the creation of the appropriate dataset, the input of data, the creation and parameterization of classes that will depict the chart and finally the appearance of the chart. To restrict our analysis on the design of the analyzed systems themselves, invocation of methods which lie outside the system boundary (i.e. methods of library classes) are excluded. No further filtering on the obtained classes is performed.

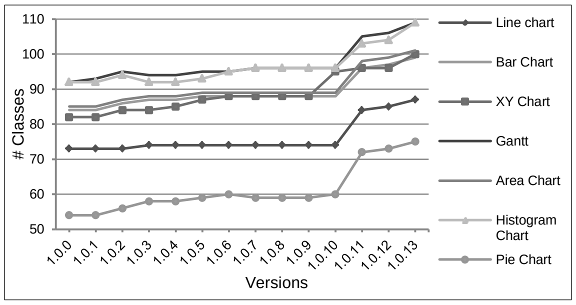
The entire process that we have followed to analyze the scattering of features is illustrated in Figure 1. In the first step, selected features are exercised on the application of interest while being monitored by the profiler. Next, the methods invoked in the executed feature are analyzed to obtain the classes in which they reside and to generate the reports shown on the right hand side of Figure 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. These kinds of analyses are analyzed next.

III. SCATTERING OF FEATURES IN SOURCE CODE

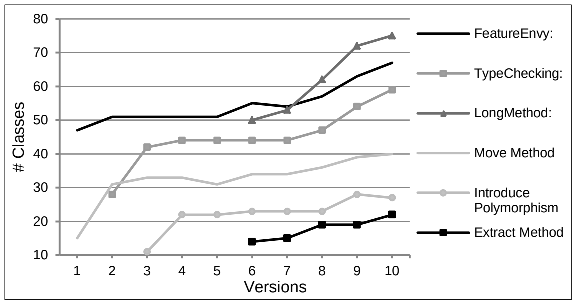
A. *Modules involved in the implementation of features*

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 [11], [19], [29], [31], [37]. Furthermore, according to Eady et al. [10], the scattering of feature implementation across the program leads to more defects in source code, and therefore deteriorates program quality. The first goal of our study is to perform an investigation of the "diffusion" of features into the implementation of a system; in other words to quantify the scattering of a specific functionality into system modules. Apart from measuring the scattering statically, that is for a given snapshot of the examined systems, we aim at studying the evolution of this scattering over a number of successive software versions.

To this end, we illustrate in Figure 2 the number of classes which are involved during the execution of a certain feature, for all versions of JDeodorant and JFreeChart, respectively. This metric has been introduced by Filho et al. [12] as the count of the number of classes (CDC) (or methods - CDO) and has also been employed in the context of Aspect-Oriented programming [13], [23]. To study macroscopically the scattering of features in source code, we have opted for an absolute measure, rather than a measure based upon statistical variance, such as the degree of scattering across classes, proposed by Eady et al. [9]. The reason is, that in the context of software evolution, a degree of scattering which quantifies simultaneously both the number of classes implementing a feature and the localization of the implementation, might yield confusing results. Assume for example, that in one version, four classes contribute equally to the implementation of a feature and that in the next version the number of involved classes increases by one, which however, contributes by a very low degree (low value for the Concentration Metric as defined by Wong et al. [38]). In that case, a decrease in the degree of scattering would be observed, as the implementation is localized mostly in the four initial classes, whereas a first interpretation should highlight that the number of involved classes has increased. Therefore, we study the distribution of the code elements (in our case methods) that contribute to the implementation of a feature, among the involved classes, by means of a separate level of analysis, as shown in subsections B and C.



(a)



(b)

Figure 2. Number of classes involved in the implementation of each feature, for (a) JFreeChart and (b) JDeodorant

The experimental results for both projects 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 the generation of a single chart, even in the early versions of JFreeChart, a number of over 90 classes may be involved. From the reengineering perspective, if a requirement's implementation should be refactored or 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. It should be noted that for 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 introducing a new chart theming mechanism.

The classes which are involved in the implementation of features can be formally represented, visualized and analyzed by means of Formal Concept Analysis [11], [28]. 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, $(F_i, C_i) \leqslant (F_j, C_j)$ 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. As an example, the concept lattice for the first and last examined version of project JFreeChart is shown in Figure 3. At this point it should be emphasized that one of the major drawbacks of concept lattices is that they do not scale well. In Figure 3 a reduced form of the sparse concept lattice has been employed, i.e. 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 [11]. 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 upwards 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 involved only in the implementation of Bar Chart and Gantt Chart, indicating a relatively low degree of reuse for the class.

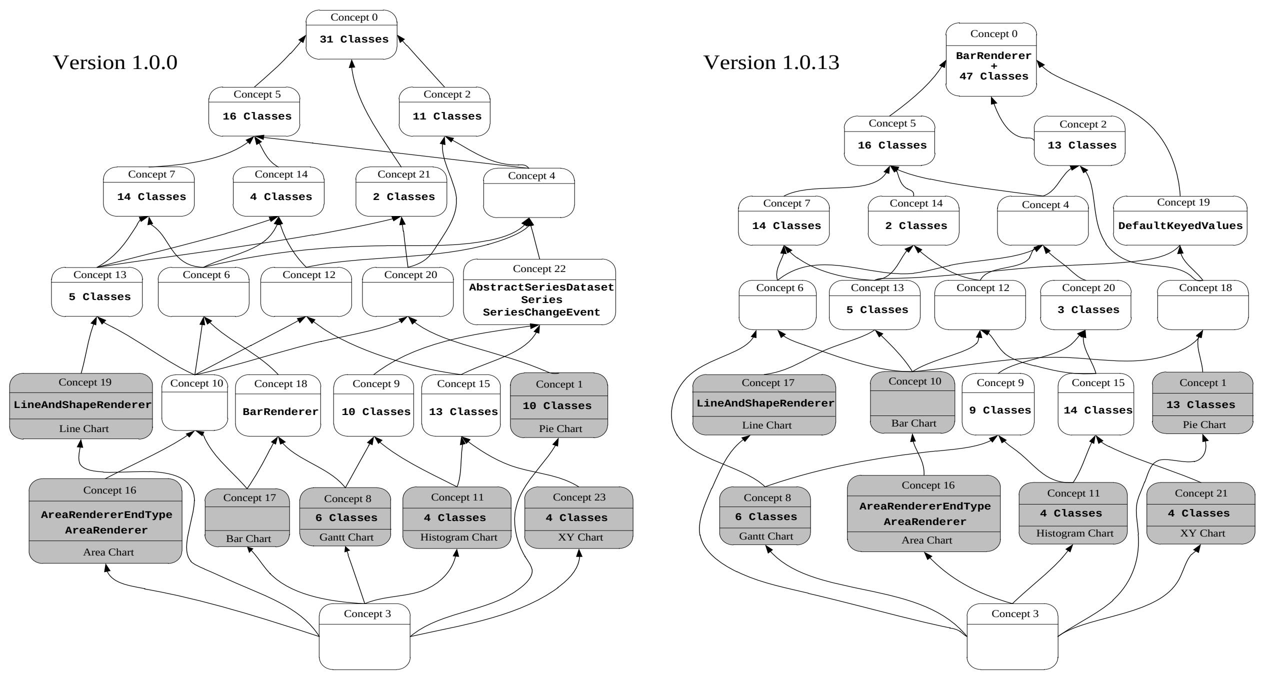


Figure 3. Concept Lattices for the first and last examined versions of JFreeChart

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 involved only 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, …, 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, …, 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.

Beyond the examination of classes involved in each of the examined features, the analysis can be extended to the methods participating in their implementation. The findings 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.

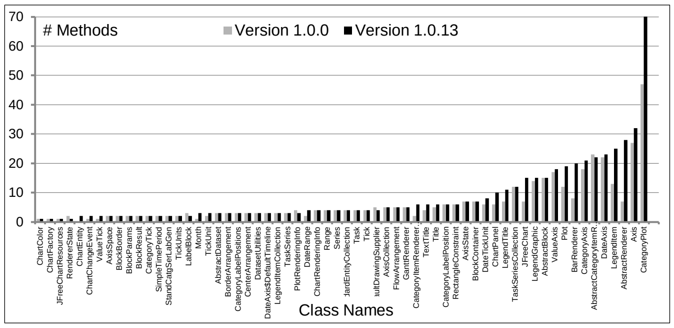
B. *Distribution of Methods Among Classes*

Even if multiple system modules (e.g. classes) are involved in the implementation of a given requirement, the corresponding functionality might not be uniformly distributed among them. To investigate the distribution of responsibilities among system classes implementing the same feature, we have recorded the number of methods contributing to the implementation of that feature for each of the aforementioned classes. Moreover, we studied the evolution of the distribution over a number of generations.

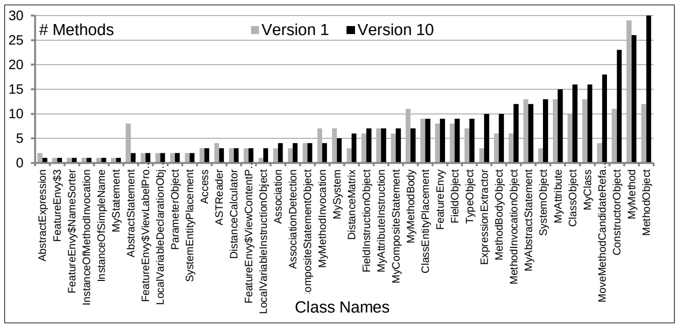
Figure 4 displays the distribution of methods among the classes involved in the generation of a Gantt chart employing the JFreeChart library and the identification of the Feature Envy code smell in JDeodorant, respectively. To provide insight on whether this distribution remains unchanged as the system evolved, the number of methods that are used in the first (light bars) and the last version (dark bars) are shown for each of the involved methods. (The figures display only the classes that exist in both the first and last version of the examined projects).

For JFreeChart, the first observation that can be made concerns the extremely skewed distribution of the methods that implement the functionality of Gantt Chart. 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. Especially the class `CategoryPlot` supports the creation of a Gantt chart by 47 methods. Similar conclusions can be drawn from JDeodorant where however, the distribution is less skewed.

The second remark is that with the passage of software versions, the methods which have been added to the systems have not been uniformly distributed but are rather concentrated in specific classes, which were already contributing a significant amount of the relevant functionality (a kind of rich-get-richer phenomenon). For example, in JFreeChart, 20% percent 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). In JDeodorant this phenomenon is less intense.



(a)



(b)

Figure 4. Distribution of methods over classes for
(a) JFreeChart - Gantt Chart and (b) JDeodorant - Feature Envy

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 is a form of preferential attachment [2], where new methods are mostly attached to the classes that have already a large number of methods contributing to the same feature. To this end, we attempt to study the evolution of these distributions in a more formal manner in the next section.

C. *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 [14], 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 for the interpretation of metrics for evolving software systems [36]. 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 of the required functionality for a feature. Essentially the Gini coefficient quantifies in the form of a clean and separate metric the localization of implementation, which is only partially quantified by the degree of scattering [10].

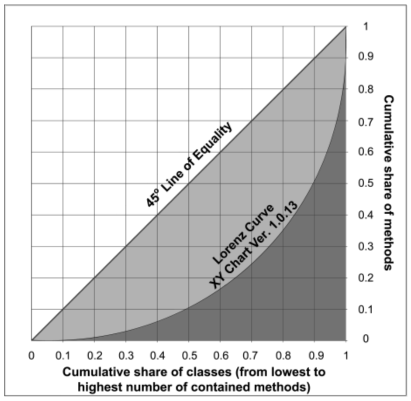


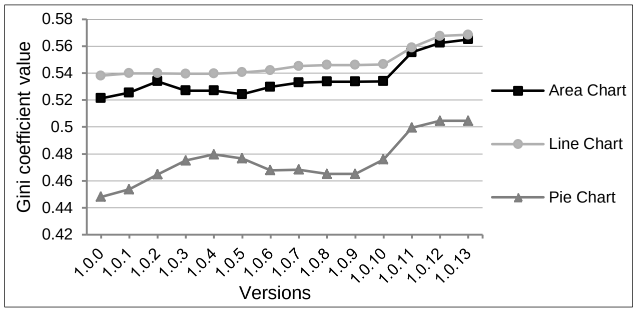
Figure 5. Graphical Representation of Gini coefficient

Usually the deviation from the perfectly even distribution is depicted graphically by means of the Lorenz curve [22] 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 5 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. The 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 observed, around 90% of the classes host 50% of the involved methods, which means that another

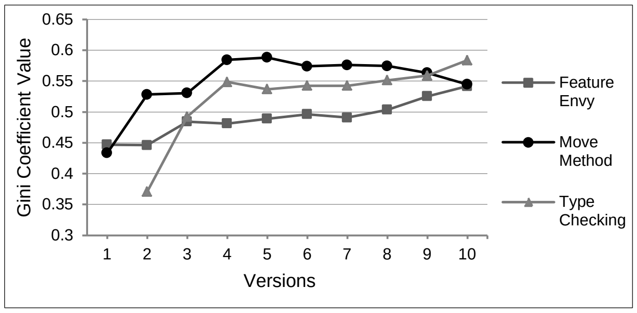
10% of the classes host the rest 50% of the methods. The corresponding Gini coefficient in this case is 0.581.

Observing the value of the Gini coefficient over the versions of an examined system, can provide insight into the evolution of the distribution of methods among the involved classes as the project matures. Figure 6 illustrates the evolution of the Gini coefficient for all examined versions of JFreeChart (for features Area Chart, Line Chart and Pie Chart) and JDeodorant (for features Feature Envy, Move Method and Type Checking) respectively.

As it can be readily observed, for the majority of the examined features, the distribution of the methods contributing to the implementation over the number of involved classes becomes more uneven as the projects evolve. This tendency, if not interrupted by means of preventive maintenance, could possibly lead to unbalanced designs where, despite the large number of classes involved in the implementation of a given feature, the functionality is mostly located in certain classes, reducing the benefit of distributing functionality among several classes. As a result, the Gini coefficient and its evolution offer an intuitive way to validate the application of a dynamic version of Riel's heuristic [30] according to which classes in a design implementing a common feature, should share the work uniformly as systems evolve.



(a) JFreeChart



(b) JDeodorant

Figure 6. Evolution of the Gini coefficient for selected features

IV. 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 [31] and even the number of anticipated defects [10]. 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 [5]. The most commonly used binary measure for quantifying the similarity between two sets, is the Jaccard similarity which considers the number of elements that are present in both sets as well as the number of elements which are unique in each set [25]. The measure that we employed for evaluating the similarity between two features stems from paleontology [33] 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 employ exactly 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 [33] and highlights part-whole relations. 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.

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

$$distance(f_1, f_2) = 1 - similarity(f_1, f_2)$$
$$= 1 - \frac{|commonMethods(f_1, f_2)|}{\min(|methods_{f_1}|, |methods_{f_2}|)}$$

where:

$|methods_{f_1}|$ corresponds to the number of methods implementing feature f_1

$|methods_{f_2}|$ corresponds to the number of methods implementing feature f_2, and,

$|commonMethods(f_1, f_2)|$ 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 Multi-Dimensional Scaling for visualizing distances. MDS [4] 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

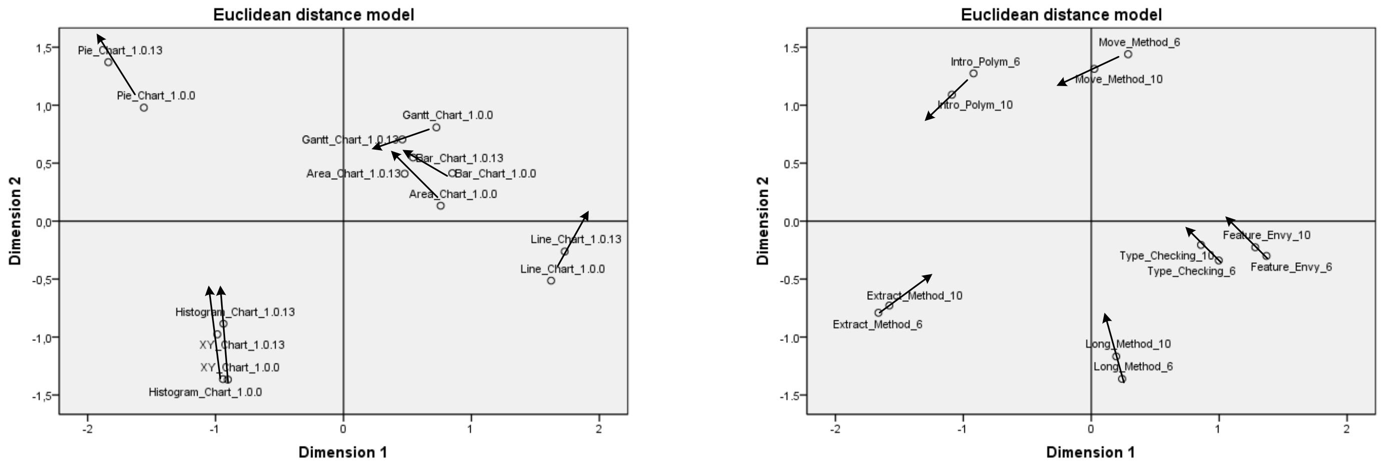


Figure 7. Multidimensional Scaling for features of versions 1.0.0 and 1.0.13 of JFreeChart (left) and versions 6 and 10 of JDeodorant (right)
* the initial version for JDeodorant is v6 since this is the first version in which all of the examined features are present

reflects the numerical measure of distance between the examined data items. Figure 7 illustrates the output of MDS, for two versions (initial and last one) of projects JFreeChart and JDeodorant, respectively, employing as distance the aforementioned Simpson measure.

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 [34]. However, the orientation of the axes 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.

The MDS output for JFreeChart 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 of 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, 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 towards the center of the diagram as the system evolves, implying an increase in the degree of reuse.

V. Threats to Validity

As in any kind of empirical study, the fact that the proposed techniques have been applied on two software projects and some of their features, as well as the fact that a limited number of versions have been analyzed, poses the usual threats to external validity, i.e. limits the possibility to generalize our findings. However, emphasis has not been placed on the conclusions that can be drawn from these case studies. The corresponding systems have been mainly used to exemplify the application of the proposed techniques.

Regarding the internal validity of the study (i.e. the factors that might affect the phenomena that we are trying to investigate), the most serious threat is related to the presence of other features which might have not been included in our analysis. These features might be interleaved with the features that were the focus of our study, especially in the investigation of reusability among classes. This threat is particularly valid in JFreeChart where other types of charts are also available, while in JDeodorant all of the available features have been studied.

VI. 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 [1], [6]. IR models that are usually employed are Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Probabilistic Network (PN) [42]. The first steps on automated static Feature Location were made by Biggerstaff et al. [3] who have built a tool that locates identifiers in source code and clusters them in order to facilitate Feature Location. Antoniol et al. [1] 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. [23], [24] employed Latent Semantic Indexing in order to locate concepts in source code, while, for the same purpose, Shepherd et al. [32] 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. [27], who used Formal Concept Analysis in order to refine the mapping tables that resulted from Latent Semantic Indexing, or Zhao et al. [40], [41], 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 [37] presented "Software Reconnaissance", a method which discovers software modules that implement a particular feature. Dynamic execution tracing (slicing) has been also adopted by Wong et al. [39] 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. [11] and Koschke et al. [19], 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. [27] and Liu et al. [21] 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. [15], 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. [38], who quantified the closeness between a feature and a software component involved in its implementation. Eady et al. evolved and extended Wong's metrics in [10] where they investigated 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. [6] have also employed crosscutting metrics in order to predict possible software instability in early development artifacts such as requirement descriptions.

VII. Conclusions and Future Work

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, as well as an appropriate similarity measure along with multi-dimensional scaling to study the evolution of the reuse among methods contributing to a feature.

The application of the proposed analyses on two case studies revealed the following information: feature scattering increases monotonically as the systems evolve and the distribution of methods contributing to the implementation of the examined features tends to become more unbalanced over time. In one of the systems, this deterioration of design properties can be partially justified by an increase in the reuse of methods and classes among features, while in the other system the distance between features increased over time, signifying less reuse among their implementations. Although the goal was not to perform an extensive empirical study regarding software evolution, 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.

Beyond the normal adaptive and corrective maintenance performed on an evolving software system, a line of future research could be the study of the impact of intentional design modifications such as refactorings, on the aforementioned trends. In other words, it would be interesting to investigate whether code restructuring is able to invert an existing trend in the evolution of feature scattering. Moreover, the automation of the entire process, i.e. the linking of source code elements to selected features and the application of the proposed analyses without human intervention, will enable large scale empirical studies on the evolution of feature scattering.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, ″Recovering traceability links between code and Documentation,″ IEEE Transactions on Software Engineering, vol. 28, no. 10, pp. 970-983, 2002.

[2] L Barabasi, R. Albert, H. Jeong, and G. Bianconi, ″Power-law distribution of the World Wide Web,″ Science, vol. 287, pp. 2115 2000.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, ″Program Understanding and the Concept Assignment Problem,″ Communications of the ACM, vol. 37, no. 5, pp. 72-82, 1994.

[4] C. C. Chen, W. Hardle, A. Unwin, M. Cox, T. F. Cox, Handbook of Data Visualization. Springer Berlin Heidelberg, 2008.

[5] S.-S. Choi, S.-H. Cha, and C. C. Tappert, ″A survey of Binary similarity and distance measures,″ Journal of Systemics, Cybernetics and Informatics, vol. 8, no. 1, pp. 43 – 48, 2010.

[6] J. M. Conejero, E. Figueiredo, A Garcia, J. Hernández and E. Jurado, ″Early Crosscutting Metrics as Predictors of Software Instability,″ Objects, Components, Models and Patterns. Lecture Notes in Business Information Processing, Vol. 33, Part 3, pp.136-156, 2009

[7] A. De Lucia, M. Di Penta, and R. Oliveto, ″Improving Source Code Lexicon via Traceability and Information Retrieval,″ IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 205-227, 2011.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "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.

[9] M. Eaddy, A. Aho, and G.C. Murphy, ″Identifying, Assigning, and Quantifying Crosscutting Concerns,″ Proc. Workshop Assessment of Contemporary Modularization Techniques, 2007.

[10] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, ″Do Crosscutting Concerns Cause Defects?,″ IEEE Transactions on Software Engineering, vol. 34. No. 4, pp. 497-515, 2008.

[11] T. Eisenbarth, R. Koschke, and D. Simon, ″Locating Features in Source Code,″ IEEE Transactions on Software Engineering, vol. 29, no. 3, pp. 210-224, 2003.

[12] F.C. Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, and C.M.F. Rubira, ″Exceptions and Aspects: The Devil Is in the Details,″ Foundations of Software Engineering, pp. 152-162, 2006.

[13] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A.v. Staa, ″Modularizing Design Patterns with Aspects: A Quantitative Study,″ Proc. Int'l Conf. Aspect-Oriented Software Development, 2005.

[14] C. Gini, ″Measurement of Inequality of Incomes,″ The Economic Journal, vol. 31, no. 121, pp. 124-126, 1921.

[15] O. Greevy, S. Ducasse, and T. Girba, ″Analyzing Software Evolution Through Feature Views,″ Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, no. 06, pp. 425-456, 2006.

[16] O. C. Z. Gotel and C. W. Finkelstein, ″An Analysis of the Requirements Traceability Problem,″ Proc. 1st Int'l Conference on Requirements Engineering, pp. 94-101, Colorado Springs, 1994.

[17] JDeodorant, http://www.jdeodorant.com, October 2011.

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

[19] R. Koschke and J. Quante, ″On Dynamic Feature Location,″ Proc. 20th IEEE/ACM International Conference on Automated software engineering, pp.86-95, California, 2005.

[20] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh, ″On Conputing the Canonical Features of Software Systems,″ Proc. 13th Working Conf. on Reverse Eng, Benevento, Italy, October 2006.

[21] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, ″Feature location via information retrieval based filtering of a single scenario execution trace,″ Proc. 22nd International Conference on Automated Software Engineering, Atlanta, 2007.

[22] M. O. Lorenz, ″Methods of measuring the concentration of wealth,″ Publications of the American Statistical Association, vol. 9, no. 70, pp. 209 – 219, 1905.

[23] A. Marcus and J. I. Maletic, ″Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing,″ Proc. 25th Int'l Conference on Software Engineering, pp. 125-136, Portland, 2003.

[24] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, ″An information retrieval approach to concept location in source code,″ Proc. 11th Working Conference on Reverse Engineering, pp. 214- 223, 2004.

[25] R. Naseem, O. Maqbool, and S. Muhammad, ″Improved Similarity Measures For Software Clustering,″ Proc. 15th European Conference on Software Maintenance and Reengineering, Oldenburg, 2011.

[26] L. Parnas, ″Software aging″, Proc. 16th Int. Conf. on Software Engineering, pp. 279–287, Italy, 1994.

[27] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, ″Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval,″ IEEE Transactions on Software Engineering, vol. 33, no. 6, pp. 420-432, June, 2007.

[28] D. Poshyvanyk and A. Marcus, ″Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code,″ Proc. 15th IEEE Int'l Conf. Program Comprehension, pp. 37-48, June 2007.

[29] M. Revell, M. Gethers, and D. Poshyvanyk, ″Using structural and textual information to capture feature coupling in object-oriented software,″ Empir. Software Eng. vol. 16. no. 6, March 2011.

[30] A.J. Riel, Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[31] M. P. Robillard and G. C. Murphy, ″Representing Concerns in Source Code,″ ACM Trans. on Software Engineering and Methodology, vol. 16, no. 1, 2007.

[32] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, ″Using natural language program analysis to locate and understand action-oriented concerns,″ Proc. 6th International Conference on Aspect-Oriented software development, Vancouver, 2007.

[33] G. G. Simpson, ″Notes on the measurement of faunal resemblance,″ American Journal of Science, 258-A, pp. 300-311, 1960.

[34] K. Singh, Quantitative Social Research Methods. Sage Publications, 2007.

[35] M. Trifu, ″Tool-Supported Identification of Functional Concerns in Object-Oriented Code,″ PhD thesis, Karlsruhe Institute of Technology, 2010.

[36] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, ″Comparative Analysis if Evolving Software Systems Using the Gini Coefficient,″ Proc. 25th Int'l Conference on Software Maintenance, pp. 179-188, Edmonton, 2009.

[37] N. Wilde and M.C. Scully, ″Software Reconnaissance: Mapping Program Features to Code,″ Software Maintenance: Research and Practice, vol. 7, pp. 49-62, 1995.

[38] W. E. Wong, S. S. Gokhale, and J. R. Horgan, ″Quantifying the closeness between program components and features,″ Journal of Systems and Software – Special Issue on Software Maintenance, vol. 54, no. 2, pp. 87-98, 2000.

[39] W. E. Wong, S. S. Gokhale, J. R. Horgan and K. S. Trivedi, ″Locating Program Features using Execution Slices,″ Proc. IEEE Symposium on Application-Specific Systems and Software Engineering & Technology, pp. 194–203, Texas, 1999.

[40] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, ″SNIAFL: Towards a Static Non-Interactive Approach to Feature Location,″ Proc. 26th International Conference on Software Engineering, pp. 293-303, Scotland, 2004.

[41] W. Zhao, L. Zhang, Y. Liu, J. Luo, and J. Sun, ″Understanding How the Requirements Are Implemented in Source Code,″ Proc. 10th Asia-Pacific Software Engineering Conference, pp. 68-77, Chiang Mai, 2003.

[42] X. Zou, R. Settimi, and J. Cleland-Huang, ″Improving automated requirements trace retrieval: A study of term-based enhancement methods,″ Empirical Software Engineering, vol. 15, pp. 119-146, July 2009.