Fundamenta Informaticae 91 (2009) 507–522
DOI 10.3233/FI-2009-0054
IOS Press

Analysis of Components for Generalization using Multidimensional Scaling

Robertas Damaševičius[*]
Software Engineering Department, Kaunas University of Technology
Studentu 50-415, LT-51368, Kaunas, Lithuania
robertas.damasevicius@ktu.lt

Abstract. To achieve better software quality, to shorten software development time and to lower development costs, software engineers are adopting generative reuse as a software design process. The usage of generic components allows increasing reuse and design productivity in software engineering. Generic component design requires systematic domain analysis to identify similar components as candidates for generalization. However, component feature analysis and identification of components for generalization usually is done *ad hoc*. In this paper, we propose to apply a data visualization method, called Multidimensional Scaling (MDS), to analyze software components in the multidimensional feature space. Multidimensional data that represent syntactical and semantic features of source code components are mapped to 2D space. The results of MDS are used to partition an initial set of components into groups of similar source code components that can be further used as candidates for generalization. STRESS value is used to estimate the generalizability of a given set of components. Case studies for Java Buffer and Geom class libraries are presented.

Keywords: component based software engineering, feature engineering, domain analysis, software similarity, generalization, multidimensional scaling

1. Introduction

Component-based software engineering (CBSE) aims at decreasing software development costs and time-to-market by building software systems from prefabricated building blocks (components) [1]. CBSE focuses on software reuse, i.e., the use of existing artifacts for the construction of software. Reuse cannot be achieved without some form of generalization [2].

[*]Address for correspondence: Software Engineering Department, Kaunas University of Technology, Studentu 50-415, LT-51368, Kaunas, Lithuania

Generalization is a form of knowledge representation that means a transition from narrow and specific principles and concepts to the wider and more general ones. A higher, more generalized, level of domain knowledge encapsulates an understanding of the general properties and behavior possessed by a subset of its domain entities. Introduction of generalization usually means transition to the higher level of abstraction, where domain knowledge can be represented and explained more comprehensibly and effectively. Thus, generalization allows introducing more simplicity into the domain. In computer science, generalization is usually understood as a technique of widening of an object (component, system) in order to encompass a larger domain of objects (systems, applications) of the same or different type. Generalization is mainly used for developing reusable software components and reuse libraries [3, 4, 5].

A prerequisite for generalization is to identify similarity (other terms used in the literature are commonality, resemblance, proximity) among a set of domain entities. The commonality may refer to essential features of a software component such as attributes or behavior, or may concern only similarity in description, code repetition, duplication, cloning or redundancy. Commonalties may be introduced into programs for a variety of reasons such as rapid software development using copy-and-paste technique, reuse of widely used code fragments for reliability, poor design practices or *ad hoc* maintenance. Thus, separation and identification of common and variable concerns in the domain is a step towards achieving generalization.

Program similarity and its detection is a core concept in software evolution and maintenance [6, 7], code clone and redundancy detection [8, 9], program compression [10], malware analysis [11], plagiarism and copyright infringement detection [12, 13], and legacy system reengineering [14, 15]. Program similarity is also important for solving library scaling problem [16]. Discovering and understanding program similarity allows for efficient development of new component architectures and systems [17] and for well-organized maintenance of existing systems [18].

Exploiting similarity and managing differences (variability) in software a key success factor in software product lines [18]. A key aspect of software similarity management is the discovery, explicit representation and modeling of similarity. Similarity reveals itself on different stages of software development process such as requirement formulation (e.g., feature models), automated application development (e.g., program generators, metaprograms [19]), system architectures (e.g., design patterns), implementation constructs (e.g., templates, parameters). However, there is still lack of understanding among software developers that all these aspects of program similarity are related and must be managed explicitly.

Program similarity can be defined in terms of the form, properties or characteristics of program representation. Here program representation is understood as a sequence of source code characters forming a more complex text structure. We distinguish between textual, characteristics-based, feature-based and information-based similarity:

- *Textual similarity* can be evaluated using string comparison measures such as Levenshtein distance or longest common sequence (for a review, see [9]).
- *Characteristics-based similarity* evaluates abstract characteristics of source code such as line count, McCabe's cyclomatic complexity, or Halstead metric [20, 21].
- *Feature-based similarity* evaluates the amount of correspondence between specific aspects of programs such as the list of identifiers [22] or business elements [23].
- *Information-based similarity* evaluates the amount of information shared between two programs using information-theoretic metrics such as Kolmogorov complexity [24] or conditional entropy.

Another taxonomy of program similarity types can be found in [25]. Similarity also can be explored along different ”levels” of abstraction, which correspond to the levels of abstraction in programs. For example, we could examine similarity at statement, block, class, unit, or architectural levels.

A family of similar components can be generalized into a generic component. Members of such family share the same commonalities and have the related variability. The usage of generic components increases reuse and design productivity in software engineering, because specific components can be derived at any time by specialization or instantiation. The success of generalization and generic component design largely depends upon domain analysis.

Several domain analysis methods, in one way or another, consider component generalization such as Multi-Dimensional Separation of Concerns [26], FODA [27] or FAST [28] (for a review, see [29]). The common aim of domain analysis methods is to construct or improve a reusable (generic) component, or to populate reusable component libraries. All these domain analysis methods perform analysis for generalization *ad hoc*. The designer analyzes available components or/and design space, builds feature/concern tables, separates commonalities and variabilities, and uses the results of analysis for developing generic components. The analysis process is heuristic. Given the same set of components, other designers can select other components and features for generalization. No evaluation is given on the quality of selected partition of a component set and the extent of generalization.

The novelty of this paper is as follows. We propose to use a mathematical method, called Multidimensional Scaling (MDS), for visualizing multidimensional software component feature space and identifying clusters of similar components as candidates for generalization.

The remaining parts of the paper are as follows. Section 2 formulates the main problems of component analysis for generalization. Section 3 presents a brief description of the MDS method. Section 4 describes application of MDS for component analysis. Section 5 presents the experimental validation of the approach for Java Buffer and Geom libraries. Finally, Section 6 presents the evaluation of results and conclusions.

2. Main problems of component analysis for generalization

We formulate the main problems of component analysis for generalization as follows:

1. *How to identify similar components amongst the available set of components?* Generic components capture commonalities in the domain. The more there are similarities between the generalized components, the better generalization can be achieved, which ultimately allows for better component reuse, library scaling and maintenance.

2. *How many generic components we should design?* We can design one large generic component, which generalizes all available components for generalization and has many different parameters. However, such generic component may be difficult to comprehend and to handle, and it may be over-generalized. Alternatively, we may design several smaller generic components, which better capture commonalties in a domain, and are more scalable and dependable.

3. *How to partition a set of components into subsets, each for every generic component?* Different partitioning of a component set may lead to different quantity of generic components developed and may increase or decrease the designer’s effort for generalization. Unsuccessful partitioning may lead to unsuccessful generalization and un-usable (un-reusable) generic components.

4. *How to determine a set of parameters for generalization of components?* Different software components may have distinct features. The generic component must reflect all features of the components it generalizes. Smaller number of parameters means better comprehensibility and easier maintenance, and leads to higher reusability.

5. *How to separate dependable and undependable parameters of generic components?* The dependency relationships between parameters may be a problem for the designer. They can be difficult to identify and the available metalanguage (i.e., a language used for describing generic components) may not support specification of dependant parameters.

6. *How to measure the extent of generalization?* Several metrics can be used to measure the extent of generalization such as: the number of generic parameters, the number of instances that can be generated from a generic component, or the amount of generated code [30]. Though all these metrics may be useful, they do not take the *quality* of generalization into account. If a generic component has too many generic parameters, it may cause the over-generalization problem. Not all generated instances may be required or useful. Much of the code generated from a generic component may be redundant. Thus, there is a need for a metric that describes the extent of generalization and is domain-independent.

The existing domain analysis methods do not provide a comprehensible solution to these problems. Thus, a combination of various, usually heuristic, methods is usually used. Component similarity is usually determined heuristically, based on their ”look-alikeness” [31]. The partitioning of a set of initial components for generalization is often described as ”library scaling” problem [16]. Some authors prefer a small number of large, ”coarse-grained” generic components [32], while others argue that better reuse can be achieved using a large number of small, flexible, more widely applicable ”fine-grained” generic components [33]. The parameters of a generic component and their relationships are modeled using feature models [34, 19]. The success of generalization is evaluated by measuring the reuse metrics of the developed generic components [35], which in many cases may be meaningless or uninformative.

We argue that the MDS method can be used as a step towards the solution of these domain analysis problems. In next Section, we present a brief introduction into the MDS method.

3. Introduction to Multidimensional Scaling

Multidimensional Scaling (MDS) [36, 37] is a mathematical method to map complex multidimensional data into lower-dimensional space, which allows easier analysis of data. MDS is used in the similar context to analyze and visualize document collections [38], databases [39], web pages [40], as well as for document categorization [41], text mining [42], knowledge discovery [43] and clustering software for change [44].

Suppose we, have a set of objects characterized by a number of features and that a measure of the similarity between objects is known. This measure indicates how similar or dissimilar two objects are. Since the number of object features can be very large, such data can be very difficult to analyze and visualize, unless the data can be represented in a smaller number of dimensions. Some sort of dimension reduction is usually necessary.

MDS maps the high-dimensional data into a lower-dimensional space, in which each object is represented by a point and the distances between points resemble the original similarity information; i.e., the larger the dissimilarity between two objects, the farther apart they should be in the lower dimensional (usually 2D) space. A mapping from a multidimensional space to a 2D space ensures some similarity between the structure of an original data and of its image. This geometrical configuration of points reflects the hidden structure of the data and may help to make it easier to understand.

Further, we consider only metric MDS, in which distances between objects in multidimensional space are related to their dissimilarities linearly. Dissimilarity between elements in a vector space can be measured by a norm of their difference, i.e. by a distance between the corresponding points.

Let $X_i \in R^n, i = 1,..,k$ be the data intended to visualize. We are searching for a set of two dimensional points $Y_i \in R^2, i = 1,..,k$ whose inter-point distances $d_{ij}(Y)$ well approximate the inter-point distances $\delta_{ij} = \|X_i - X_j\|$.

A distance in the multidimensional original space can be considered as a measure of dissimilarity, and it is defined by a corresponding norm. The most widely used norm is the Euclidean norm defining distance between X_i and X_j by the formula

$$\delta_{ij} = \sqrt{\sum_{r=1}^{k} (x_{ir} - x_{jr})^2} \qquad (1)$$

where $X_i = (x_{i1},..,x_{in})^T$.

The Minkowski metric is a generalization of the Euclidean metric. The corresponding distance is defined by formula:

$$\delta_{ij} = \left(\sum_{r=1}^{k} |x_{ir} - x_{jr}|^p \right)^{\frac{1}{p}} \qquad (2)$$

A special case of Minkowski metric $p = 1$ is the so called City Block metric:

$$\delta_{ij} = \sum_{r=1}^{k} |x_{ir} - x_{jr}| \qquad (3)$$

A mapping is precisely preserving the structure of a data set, if the distances between the original points and the distances between their lower-dimensional images are equal. Practically, we want to minimize an error of approximation of the distances in the original space by the distances in the lower-dimensional space. This error is estimated by a measure of goodness-of-fit, often called ”stress”, between the configuration distances d_{ij} and the dissimilarities. A range of formulas for this measure is used:

$$STRESS : s = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} \omega_{ij}(d_{ij}(Y) - \delta_{ij})^2} \qquad (4)$$

$$STRESS1 : s = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{\omega_{ij}(d_{ij}(Y) - \delta_{ij})^2}{\delta_{ij}^2}} \qquad (5)$$

$$SSTRESS : s = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} \omega_{ij}(d_{ij}(Y) - \delta_{ij}^2)^2} \qquad (6)$$

where $\omega_{ij} \geq 0$ are weights.

There is no definite rule to determining, which stress value can be judged as representing good or poor goodness-of-fit. Kruskal [45] provided the following ”rules-of thumb” for STRESS1 values as follows: 0 – perfect, 0.025 – excellent, 0.05 – good, 0.1 – fair, $>$ 0.2 poor.

In next Section, we describe the application of MDS for analyzing multidimensional component feature space and uncovering clusters of similar components for generalization.

4. Component domain analysis for generalization based on MDS

First, we begin with some basic definitions as follows:

Definition 1: Component c_j is an object uniquely characterized by a set of its features F. Component can be represented as a point in a n-dimensional feature space F as follows:

$$c_j = (f_i, f_2, .., f_n), f_i \in F, c_j \in C \tag{7}$$

Definition 2: Feature f_i is a computable metric of a component c_j:

$$f_i = \|c_j\| \tag{8}$$

Definition 3: Similarity between two components c_i and c_j is defined as a distance δ between two points in a multidimensional feature space:

$$\delta(c_j, c_k) = \sqrt{\sum_{r=1}^{m} (f_{ir} - f_{jr})^2} \tag{9}$$

Definition 4: Cluster of components K is a group of similar components separated by a small distance.

$$K = \{c_j, .., c_k\}, \delta(c_j, c_k) \leq const \tag{10}$$

for each pair of components in cluster K.

Definition 5: Generalizability g of a set of components C is evaluated using a stress criterion.

$$g = stress(C) \tag{11}$$

We propose the following procedure based on the application of the MDS method for performing analysis of components for generalization and identifying groups (clusters) of similar components as prime candidates for generalization:

1. Identify a set of components C available for generalization.
2. Identify a set of features F of each component. The features may be extracted from component source code, feature models or domain business models (ontology, thesaurus) using visual inspection, domain analysis tools (e.g., parsers) and may include syntactical features that characterize the source code of components or semantic features that characterize the functionality (behavior) of a component.
3. Build a component feature matrix M (component $\times$ feature). The feature matrix must include at least 6 features. It represents a set of points in a multidimensional feature space.

Table 1. Summary of Buffer class library

Level	Feature dimension	Features	No. of classes
1	buffer data element type	byte, char, int, float double, long, short	7
2	memory allocation scheme	direct, non-direct	35
	byte ordering	native, non-native, Big endian, Little endian	
3	access mode	writable, read-only	32

4. Digitize a feature matrix. The numerical values for natural language descriptions of features, if any, must be provided.
5. Select a distance metric δ (see Eq. 9) to measure the dissimilarity between components in component feature space. Commonly used metrics are Euclidean (Eq. 1), Minkowski (Eq. 2) and City Block (Eq. 3), though there are others, too.
6. Select a stress criterion (see Eq. 4, 5 or 6) that estimates the error of the mapping between the multidimensional feature space and its 2D image.
7. Perform MDS on a feature matrix to obtain its 2D projection and a stress value.
8. Identify clusters in the 2D projection. Identification is usually performed by visual inspection of the 2D projection.
9. Use clusters of components to build generic components. The number of identified clusters determines the number of generic components.
10. Evaluate generalizability using stress value. Smaller stress value means more successful partitioning of the initial component set into clusters of similar components and, consequently, provides more capabilities for generalization.

In the following Section, we present the experimental validation of the proposed MDS-based component analysis framework.

5. Experimental validation of the proposed method

5.1. Java Buffer library

Buffer library is a part of JDK 1.5 class library (package java.nio.*). Buffer library contains 74 classes describing different buffers. Below, we briefly describe features of the Buffer classes and explain how those features are reflected in Buffer classes (for a more extensive description, see [46]). The class hierarchy of the Buffer library is organized in 3 levels as follows (Table 1):

- At Level 1, there are 7 classes that differ in buffer element data types. These classes contain methods for providing access to buffer functionalities implemented in the classes at Level 2.
- At Level 2, classes implement 2 memory allocation schemes (direct, non-direct) and 4 types of byte orderings (native, non-native, Little Endian and Big Endian). 20 classes result from combining memory access and byte ordering features, excluding MappedByteBuffer class, which is just a helping class. Also there are 7 heap classes that implement the non-direct memory access scheme

for a buffer. Classes with suffixes ’U’ and ’S’ implement direct memory access scheme with native and non-native byte ordering, respectively. There is only one class DirectByteBuffer, as byte ordering does not matter for byte buffers.

- At Level 3 in the class hierarchy, classes implement different access modes. In total, 25 classes at Level 3 implement the read-only variants of buffers.

The Buffer library contains many ”look-alike” components and has a great deal of redundancy. It is difficult to manage and maintain. Also component selection and reuse is difficult. Developing a smaller number of generic components, which allows for easier class selection via parameters, more convenient maintenance and elimination of unnecessary redundancy, thus contributing to higher reusability, can solve these problems. However, the Buffer library components have multiple features alongside many dimensions, their features depend upon each other, and thus the partitioning of components for generalization is not an easy task. We formulate our aim as the identification of clusters of similar component that are most susceptible for generalization, thus alleviating the work of a generic components’ designer.

First, we should identify and extract features of given components and build a component feature matrix. The components may have different feature dimensions, e.g., syntactical (based on component source code properties) or semantic (based on functionality of the components).

We have examined two large groups of component features:

- Syntactical features are based on 14 common software metrics: 1) LOC (Lines of Code) – a count of each line that has any code element, 2) eLOC (Effective LOC) – count of code statements, 3) lLOC (Logical Statements LOC), 4) Interface Complexity (number of parameters), 5) Interface Complexity (number of returns), 6) Cyclomatic Complexity – the number of linearly independent paths through a program’s source code, 7) Number of public, private, protected data attributes, 8) Number of public, private, protected methods, 9) Number of loops, 10) Number of conditional branches, 11) Number of memory allocation statements, 12) Number of parenthesis, 13) Number of braces, 14) Number of brackets.

- Semantic features are based on 6 identified functional, data and class type parameters: 1) Element type = {Byte, Char, Float, Int, Long, Short, String}; 2) Memory allocation scheme = {non-direct, direct, not-specified}; 3) Byte ordering = {native, non-native, Big Endian, Little Endian, not specified}; 4) Access mode = {read-only, writable}; 5) Content = {Byte, Char, File, Float, Int, Long, Short}; 6) Class type = {Abstract, Final}.

Based on the selected component feature dimensions, the Buffer library classes were analyzed and feature matrices were built (74×14 matrix for syntactical features and 74×6 matrix for semantic features). As the can see, we have 14D feature space in the first case, and 6D feature space in the second case, which both are difficult to comprehend and analyze.

We have applied MDS on each of the matrices to obtain a 2D projection of the feature space. The result in both cases is a 74×2 matrix, which is interpreted as the coordinates of 74 points on a 2D plane. The results are depicted graphically in Figure 1 and Figure 2 (each dot represents a different Buffer class on the 2D projection of its feature space).

As we can visually see from Figure 1, no clusters could be identified. Thus, we can state that MDS using syntactical features has failed. We do not provide stress values for different distance metrics and stress criteria, as they no longer present an interest here. We can explain the result as follows.

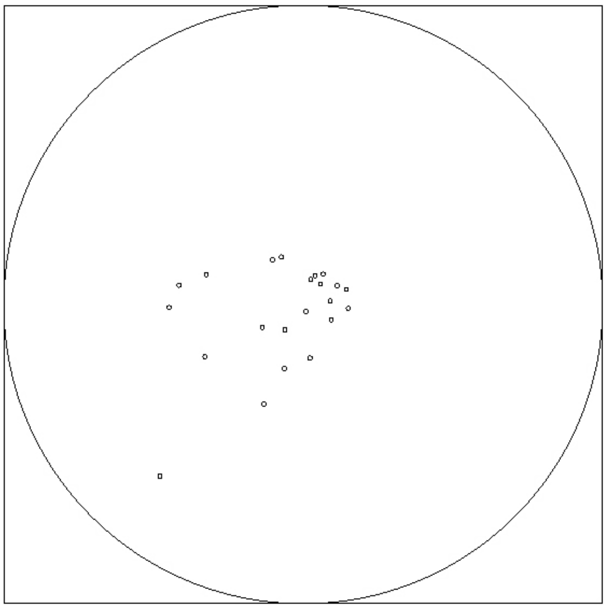

Figure 1. MDS of Buffer classes using syntactic features (Metric MDS; Euclidean distance metric; STRESS criterion).

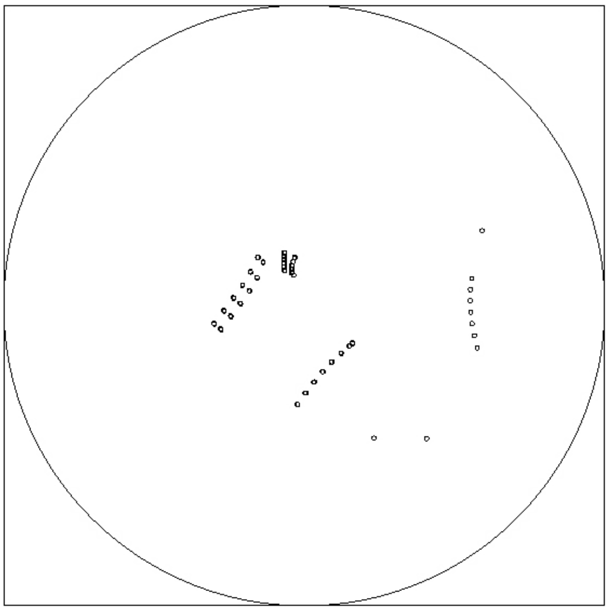

Figure 2. MDS of Buffer classes using semantic features (Metric MDS; Euclidean distance metric; STRESS criterion).

Table 2. Stress values and number of clusters for metric MDS and different distance metrics

Stress criterion	**Euclidean (p=2)**	**City Block (p=1)**	**Minkowski (p=3)**
SSTRESS	0.0296 (6 clusters)	0.0157 (3 clusters)	0.0380 (6 clusters)
STRESS	**0.0127 (6 clusters)**	0.0275 (5 clusters)	0.0298 (3 clusters)
STRESS1	0.1142 (6 clusters)	0.1386 (5 clusters)	0.1607 (3 clusters)

The selection of the component feature dimensions is very important, as the further partitioning of the component set directly depends upon it. Analysis of components along the feature dimensions that are not really important for a given component set and design aim does not allow to reveal the hidden structure and dependability within the analyzed set of components. However, this does not mean that syntactical features are not important at all. Syntactic features can be used, e.g., to classify software components based on their implementation language or programming style.

In Figure 2, we can see 6 different clusters, which can be implemented as generic components, and 3 separate classes, which differ significantly from other classes and thus should not be generalized.

Other MDS methods and distance metrics used may produce different projections and consequently reveal a different number of clusters in the component feature space. Generally, the partitioning that has the lowest stress value must be selected. We present the stress values for the metric MDS method and different distance metrics in Table 2.

We estimate the generalizability of the Java Buffer class library according to the Kruskal's rules-of thumb [45] as ”excellent” (best stress value < 0.025).

Finally, we partition the Buffer library based on the MDS results obtained in the semantic feature space. Using this partitioning, we can develop 6 different generic components:

1. Generic buffer (includes 7 user-accessible buffer classes at Level 1);
2. Generic heap buffer (includes 14 classes);
3. Generic direct buffer with non-native byte ordering (includes 14 classes);
4. Generic direct buffer with native byte ordering (includes 12 classes);
5. Generic byte buffer with Big Endian (includes 12 classes);
6. Generic byte buffer with Little Endian (includes 12 classes).

Note that 3 classes are too different and remain stand-alone (i.e., these classes should not be generalized).

5.2. Java Geom package

Java Geom package is a part of JDK 1.5 class library (package java.awt.*), which describes 2D geometric shapes such as lines, ellipses, and quadrilaterals. The Geom library has 25 classes and interfaces for describing 2D geometric forms, shapes and figures, and for defining and performing operations on objects related to two-dimensional geometry. Some important features of the package include:

- Classes for manipulating geometry, such as AffineTransform and the PathIterator interface which is implemented by all Shape objects;

- Classes that implement the Shape interface, such as Ellipse2D, Line2D, and Rectangle2D;
- The Area class, which provides mechanisms for add (union), subtract, intersect, and eXclusiveOR operations on other Shape objects.

The Geom package also contains similar classes and many ”look-alike” *Get* and *Set* class methods, which makes it difficult to manage and maintain. Also component selection and reuse is difficult. Our aim is to discover clusters of similar component that are most similar and are primary candidates for redundancy elimination and generalization. Since the package classes have many feature dimensions (syntactic, semantic), here we apply a method for evaluating similarity of software components, which is independent of their syntax and semantics: a similarity metric based on Kolmogorov Complexity [24].

Kolmogorov Complexity measures the information content of an object by the length of the smallest program that generates it. In general case, we have a domain object x and a description system (e.g., programming language) ϕ that maps from a description w (i.e., a program) to this object. Kolmogorov Complexity $K_\phi(x)$ of an object x in the description system ϕ is the length of the shortest program in the description system ϕ capable of producing x on a universal computer:

$$K_\phi(x) = min_\omega\{\|\omega\| : \phi_\omega = x\} \tag{12}$$

Kolmogorov Complexity $K_\phi(x)$ is the minimal quantity of information required to generate x by an algorithm, and is the ultimate lower bound of information content. Unfortunately, it cannot be computed in the general case and must be approximated. Usually, compression algorithms are used to give an upper bound to Kolmogorov Complexity. Suppose that we have a compression algorithm C_i. Then, a shortest compression of ω in the description system ϕ will give the upper bound to information content in x:

$$K_\phi(x) \leq min_i\{C_i(\phi_\omega)\} \tag{13}$$

Based on the definition of Kolmogorov Complexity (Eq. 12) and its approximation (Eq. 13), we can define a similarity metric for two programs x and y as a ratio of shared information content:

$$s(x,y) = 1 - \frac{C(x \cdot y)}{C(x) + C(y)}, \tag{14}$$

where $x \cdot y$ is a concatenation of x and y.

Using Eq. 14 we can examine each pair of the components of Geom package, thus arriving to a 25 $\times$ 25 component similarity matrix. This matrix represents a 25D feature space, which is difficult to comprehend and analyze. For visualization of component similarity we have applied MDS to obtain a 2D projection of the feature space. The result is a 25 $\times$ 2 matrix, which is interpreted as the coordinates of 25 points (classes) on a 2D plane. The results are depicted graphically in Figure 3 (each dot represents a different Geom class on the 2D projection of its feature space). Stress value is 0.064, which allows to estimate the generalizability of the Geom class package according to the Kruskal’s rules-of thumb [45] as ”good” (0.05 $<$ stress $<$ 0.1).

In Figure 3 we can see a relative distance of the Geom package classes. The classes that are more similar are located closer to each other. To have a more detailed view we can use a dendrogram of the Geom classes (see Figure 4 classes are identified by their index numbers given in Table 3). After analysis, we can identify 7 different clusters, which can be implemented as generic components (Table 3).

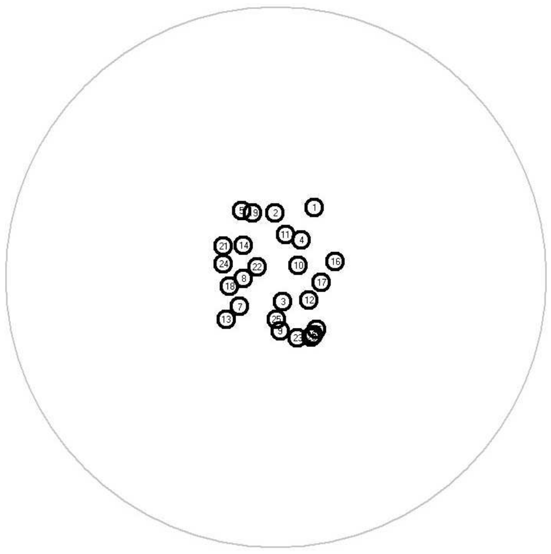

Figure 3. MDS of Geom classes using syntactic features (Metric MDS; Euclidean distance; STRESS criterion).

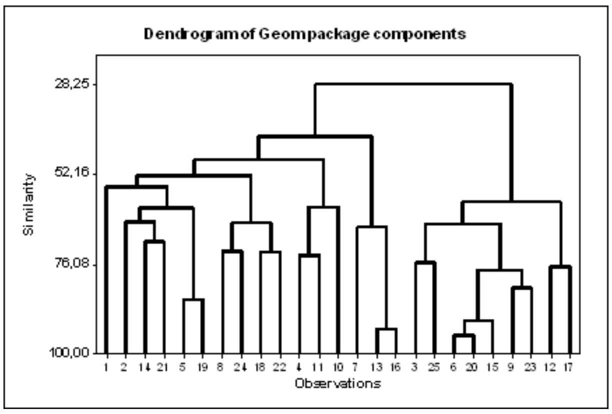

Figure 4. Dendrogram of Geom library.

Table 3. Summary of clusters of classes in Geom library

Cluster	**No. of classes**	**Classes and their indices (in brackets)**	**Description**
1	4	AffineTransform (1), Arc2D (2), Line2D(14), Rectangle2D(21)	AffineTransform performs a linear mapping from 2D coordinates to other 2D coordinates. Arc2D and Line2D define 2D arcs and lines, respectively. Rectangle2D defines a rectangle.
2	2	CubicCurve2D (5), QuadCurve2D (19)	CubicCurve2D and QuadCurve2D implement cubic and quadratic curves that connect two points.
3	4	Ellipse2D (8), Point2D (18), RectangularShape (22), RoundRectangle2D (24)	Ellipse2D and RectangularShape classes describe shapes whose geometry is defined by a rectangular frame. RoundRectangle2D defines a round rectangle. Point2D class defines a point in 2D space.
4	3	Area (4), FlatteningPathIterator(10), GeneralPath (11)	Area represents an arbitrarily-shaped area. GeneralPath represents a path constructed from lines, quadratic and cubic curves. FlatteningPathIterator returns a flattened view of PathIterator object.
5	3	Dimension2D (7), Path2D (16), LineIterator (13)	Dimension2D encapsulates a width and a height dimension. Path2D provides a shape which represents an arbitrary geometric path. LineIterator iterates over the path segments of a line segment.
6	7	ArcIterator (3), EllipseIterator (9), CubicIterator (6), QuadIterator (20), PathIterator (15), RectIterator (23), RoundRectIterator (25)	The Iterator classes return the geometry of shapes by allowing a caller to retrieve the path of boundary a segment at a time.
7	2	GeneralPathIterator (17), IllegalPathStateException (12)	GeneralPathIterator provides interfaces to Iterator classes. IllegalPathStateException represents an exception that is thrown if an operation is performed on a path that is in an illegal state.

6. Evaluation of results and conclusions

In this paper, we propose to use a mathematical MDS method for the analysis of the component feature space systematically, which allows to partition the set of software components into subsets (clusters) of similar components. The identified clusters of similar software components can be further used as primary candidates for generalization in generic component design. Similarity can be understood as syntactic similarity of source code or semantic similarity of components, depending upon analyzed component features. Thus, the method provides for more flexibility and adaptability to the software designer's needs. Similarity of components in component feature space is estimated using several distance metrics. The generalizability of the given component set is estimated using stress criteria.

The proposed feature-based component analysis method allows to solve Problems 1, 2, 3 & 6 of component analysis for generalization presented in Section 2. It allows 1) to identify software components within the available set of components that are more similar to each other than to other members of the component set; 2) to determine the number of generic components required to design a reuse library that covers the given set of components; and 3) to evaluate the generalizability of the given set of software components. The proposed method can be used to aid the development of generic component libraries or re-engineering of legacy component libraries.

References

[1] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1999.

[2] Baum, L., and Becker, M.: Generic Components to Foster Reuse, *Proc. of the 4th IEEE Int. Conf. on Tools of Object-Oriented Languages and Systems (TOOLS-Pacific 2000)*, Sydney, Australia, 2000, 266–277.

[3] Becker, M.: Generic Components – A Symbiosis of Paradigms, *Proc . of Second Int. Symposium on Generative and Component-Based Software Engineering, GCSE 2000*, LNCS 2177, Springer 2001, 100–113.

[4] Sadaoui, S., Yin, P.: Generalization for component reuse, Proc. of the 42nd Annual Southeast Regional Conference, Huntsville, Alabama, 134–139, 2004.

[5] Frakes, W. B., Kang, K.: Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, **31**(7), 529–536, July 2005.

[6] Damaševičius, R.: Visualization and Analysis of Open Source Software Evolution using An Evolution Curve Method, *Proc. of 8th Int. Baltic Conference on Databases and Information Systems (Baltic DB&IS 2008)*, Tallinn, Estonia, June 2-5, 2008, 193–204.

[7] Zou, L., Godfrey, M. W.: Using origin analysis to detect merging and splitting of source code entities, *IEEE Transactions on Software Engineering*, **31**, 2005, 166–181.

[8] Jarzabek, S., Li, S.: Unifying clones with a generative programming technique: a case study, *Journal of Software Maintenance and Evolution: Research and Practice*, **18(4)**, 2006, 267–292.

[9] Koschke, R.: Survey of research on software clones, in: *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, A. Walenstein, Eds.), 2007. IBFI, Dagstuhl, Germany.

[10] Evans, W.: Program compression, in: *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, A. Walenstein, Eds.), 2007. IBFI, Dagstuhl, Germany.

[11] Walenstein, A., Lakhotia, A.: The Software Similarity Problem in Malware Analysis, in: *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, A. Walenstein, Eds.), 2007. IBFI, Dagstuhl, Germany.

[12] Lancaster, T., Culwin, F.: A comparison of source code plagiarism detection engines, *Computer Science Education*, **14**, 2004, 101–112.

[13] Ciesielski, V., Wu, N., Tahaghoghi, S.: Evolving Similarity Functions for Code Plagiarism Detection, *Proc. of 10th Conference on Genetic and Evolutionary Computation, GECCO08*, July 12-16, 2008, Atlanta, GA, USA.

[14] Wiggerts, T. A.: Using clustering algorithms in legacy systems remodularization. *Proc. of the 4th Working Conference on Reverse Engineering*, 6-8 October 1997, 33–43.

[15] van Deursen, A., Kuipers, T.: Finding Classes in Legacy Code Using Cluster Analysis, *Proc. of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering* (S. Demeyer, H. Gall, Eds.), Zurich, 1997.

[16] Biggerstaff, T. J.: The library scaling problem and the limits of concrete component reuse, *Proc. of 3rd Int. Conf. on Software Reuse*, 1994, 102–109.

[17] Armstrong, R., Allan, B. A., Bernholdt, D. E., Elwasif, W. R.: On the Role of Self-Similarity in Component-Based Software, *CompFrame 2005 Workshop on Component Models and Frameworks in High Performance Computing*, 22-23 June, Atlanta, Georgia, USA, 2005.

[18] Clements, P., Northrop, L.: *Software Product Lines: Practices and Patterns*, Addison Wesley, 2001.

[19] Damaševičius, R., Štuikys, V., Toldinas, E.: Domain Ontology-Based Generative Component Design Using Feature Diagrams and Meta-Programming Techniques. *Proc. of 2nd European Conf. on Software Architecture, ECSA2008*, Paphos, Cyprus, LNCS 5292, 338–341, Springer-Verlag, 2008.

[20] Yamamoto, T., Matsushita, M., Kamiya, T., Inoue, K.: Measuring similarity of large software systems based on source code correspondence, *Proc. of 6th Int. Conf. on Product Focused Software Process Improvement, PROFES 2005*, Oulu, Finland, June 13-15, 2005. LNCS 3547, Springer, 2005, 530–544.

[21] Whale, G.: Identification of Program Similarity in Large Populations, *The Computer Journal*, **33**(2), 140–146, 1990.

[22] Damaševičius, R.: Analysis of Software Library Components for Similarity using Multi-dimensional Scaling, *Computing and Information Systems Journal*, **11**(2), 2007, 1–9.

[23] FanChao, M., Dechen, Z., Xiaofei, X.: A Design Method of Reusable Components Based on Feature Matching, *International Journal of Computer Science and Network Security*, **6**(5), 24–29, 2006.

[24] Li, M., Chen, X., Li, X., Ma, B., Vitanyi P.: The similarity metric, *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 863–872, 2003.

[25] Walenstein, A., El-Ramly, M., Cordy, J. R., Evans, W., Mahdavi, K., Pizka, M., Ramalingam, G., von Gudenberg, J. W., Kamiya T.: Similarity in Programs, in: *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, A. Walenstein, Eds.), 2007. IBFI, Dagstuhl, Germany.

[26] Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach, in: *Software Architectures and Component Technology: The State of the Art in Software Development* (M. Aksit, Ed.), Kluwer Academic Publishers, 2001.

[27] Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

[28] Weiss, D. M., Lai., C. T. R.: *Software Product-Line Engineering: A Family-Based Software Development Approach*, Reading: Addison-Wesley, 1999.

[29] Ferre, X., Vegas, S.: An Evaluation of Domain Analysis Methods. *Proc. of 4th Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'99)*, Heidelberg, Germany, 14-15 June, 1999.

[30] Damaševičius, R., Štuikys, V.: Evaluation of Metaprogram Complexity. *Proc. of 15th Conference on Information and Software Technologies (IT 2009)*, Kaunas, Lithuania.

[31] Damaševičius, R., Štuikys, V.: Separation of Concerns in Multi-language Specifications, *INFORMATICA*, **13**(3), 255–274, 2002.

[32] Estublier, J., Vega, G.: Reuse and Variability in Large Software Applications, *Proc. of 10th European Software Engineering Conference and 13th Int. Symp. on Foundations of Software Engineering (ESEC-FSE'05)*, September 5-9, 2005, Lisbon, Portugal, 316–325.

[33] Jonge de, M.: Package-Based Software Development, *Proc. of 29th Euromicro Conference (EUROMICRO'03)*, 3-5 September 2003, Belek-Antalya, Turkey, 76–85.

[34] Beuche, D., Papajewski, H., Schrder-Preikschat, W.: Variability management with feature models. *Sci. Comput. Program.*, **53**(3), 333–352, 2004.

[35] Washizaki, H., Yamamoto, H., Fukazawa, Y.: A Metrics Suite for Measuring Reusability of Software Components, *Proc. of 9th IEEE Int. Software Metrics Symposium (METRICS 2003)*, Sydney, Australia, 211–224, 2003.

[36] Borg, I., Groenen, P.: *Modern Multidimensional Scaling*, Springer, 1997.

[37] Cox, T. F, Cox, M. A.: *Multidimensional Scaling*, Chapman and Hall, 2001.

[38] Becks, A., Sklorz, S., Jarke, M.: A Modular Approach for Exploring the Semantic Structure of Technical Document Collections, *Proc. of International Working Conference on Advanced Visual Interfaces, AVI 200*0, May 23-26, 2000, Palermo, Italy, 298–301.

[39] Popescul, A., Flake, G. W., Lawrence, S., Ungar, L. H., Lee Giles C.: Clustering and Identifying Temporal Trends in Document Databases, *Proc. of IEEE Advances in Digital Libraries ADL 2000*, Washington, DC, 173–182, 2000.

[40] Haveliwala, T., Gionis, A., Klein, D., Indyk, P.: Evaluating Strategies for Similarity Search on the Web, *Proc. of the 11th Int. World Wide Web Conference*, May 2002, 432–442.

[41] Stein, B., Meyer zu Eissen, S.: Automatic Document Categorization: Interpreting the Performance of Clustering Algorithms, in: *Advances in Artificial Intelligence* (A. Gnter, R. Kruse, B. Neumann, Eds.). LNAI 2821, 254–266, Springer 2003.

[42] Munoz, A., Martin-Merino, M.: New asymmetric iterative scaling models for the generation of textual word maps, *Proc. Proc. of the 6th Int. Conf. on the Statistical Analysis of Textual Data, CSIA-CERESTA*, (A. Morin, P. Sbillot, Eds.), vol. 2, 593–603, 2002.

[43] Fayyad, U., Grinstein, G. G., Wierse, A. (eds.). *Information Visualization in Data Mining and Knowledge Discovery*, Morgan Kaufman, London/San Francisco, 2002.

[44] Beyer, D., Noack, A.: Clustering Software Artifacts Based on Frequent Common Changes, *Proc. of the 13th Int. Workshop on Program Comprehension (IWPC'05)*, 15-16 May 2005, St. Louis, MO, USA, 259–268.

[45] Kruskal, J. B.: Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis, *Psychometrika*, **29**(1), March 1964, 1–27.

[46] Jarzabek, S. Li, S.: Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique, *Proc. of ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003, Helsinki, 237–246.