Analysis of Software Library Components for Similarity using Multidimensional Scaling

Robertas DAMAŠEVIČIUS
Software Engineering Department, Kaunas University of Technology
Studentų 50, Kaunas, Lithuania
E-mail: robertas.damasevicius@ktu.lt

Abstract

Identification of similar or 'look-alike' software components is essential for design of generic components. However, feature analysis and identification of components as candidates for generalization usually is done ad hoc. We propose to apply a Multidimensional Scaling (MDS) method to analyze source code components in the multidimensional feature space. Multidimensional feature data that represent component properties are mapped to 2D space. The results of MDS are used to partition an initial set of software components into several clusters and to identify groups of similar components as prime candidates for generalization. STRESS value is used to estimate the generalizability of a given set of components.

1. Introduction

Component-based software engineering aims at decreasing software development costs and time-to-market by building software systems from prefabricated building blocks (components) (Szyperski, 1999). Component-based software engineering focuses on software reuse, i.e., the use of existing artifacts for the construction of software systems. Reuse cannot be achieved without some form of generalization (Baum and Becker, 2000), which is based on some similar or 'look-alike' source code components or fragments of it. Such similar code fragments or *clones* (Basit and Jarzabek), if detected, represented a useful source of design information that can be used for easing software maintenance and identifying reuse opportunities.

The ability to generalize is an essential part of any intellectual and scientific activity. Generalization is fundamental to mathematics and philosophy and important in computer science as well. For example, in mathematics, laws and theorems are often generalizations. They state some property that is true over a large group of things. Generalization provides a form of knowledge representation. Strictly speaking, generalization means a transition from narrow and specific principles and concepts to the wider and 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 (Becker, 2001; Sadaoui and Yin, 2004; Frakes and Kang, 2005) and plays a key role for supporting automatic program generation in the context of generative programming (Jarzabek et al., 2006).

Generalization identifies commonalties 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. Thus, *separation and identification of common and variable concerns* in the domain is a step towards achieving generalization.

The result of generalization is a generic component that represents a family of similar software components. 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 as follows.

Multi-Dimensional Separation of Concerns (Ossher and Tarr, 2001) understands *design concerns* in terms of an *n*-dimensional design space. Each dimension is associated with a set of similar concerns, such as a set of component instances; different values along a dimension are different instances.

Feature-Oriented Domain Analysis (Kang et al., 1990) analyses distinctive *features* of software systems that define both common domain aspects as well as differences between related systems. The underlying concepts are: *aggregation* (composition of separated concerns into a generic component), *decomposition* (abstraction and isolation of domain commonalties and variations), *generalization* (capturing domain commonalties, while expressing the variations at a higher level of abstraction), and *specialization* (tailoring a generic component into a specific component that incorporates the application-specific details).

Family-oriented Abstraction, Specification and Translation (Weiss and Lai, 1999) groups similar entities or concepts into *families*. Grouping is based on *commonalties* that fall along many dimensions (structure, algorithm, etc.). Commonality defines shared context that is invariant across abstractions of the application. The individual family members are distinguished by their *variabilities*. Variability captures the properties that distinguish abstractions in a domain. The aim is to uncover variability and provide means for its implementation.

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.

Multidimensional Scaling (MDS) (Borg and Groenen, 1997; Cox and Cox, 2001) is a mathematical method to map complex multidimensional data into lower-dimensional (usually 2D) space, which allows easier analysis of data. MDS is used in the similar context to analyze and visualize document collections (Becks et al., 2000), databases (Popescul et al., 2000), web pages (Haveliwala, 2002), as well as for document categorization (Stein and Meyer, 2003), text mining (Munoz and Martin-Merino, 2002), knowledge discovery (Fayyad et al., 2002) and clustering software for change (Beyer and Noack, 2005).

The novelty of this paper is as follows. We propose to use a mathematical method, called Multidimensional Scaling (MDS), for visualizing software component feature space and identifying component clusters 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 library. Section 6 presents the related works and evaluation of results. Finally, Section 7 presents conclusions.

2. Main problems of component analysis for generalization

Based on the above definitions, 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 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 parameters. However, such generic component may be difficult to comprehend and to handle, and it may be over-generalized. Or, we may design several smaller generic components, which better capture commonalties in a domain.

3) How to partition a set of components into subsets, each for every generic component?

Different partitioning may lead to different quantity of generic components 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 generated instances, the amount of generated code, etc. 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 overgeneralization 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" (Damaševičius and Štuikys, 2002). The partitioning of a set of initial components for generalization is often described as "library scaling" problem (Biggerstaff, 1994). Some authors prefer a small number of large, "coarse-grained" generic components (Estublier and Vega, 2005), while others argue that better reuse can be achieved using a large number of small, flexible, more widely applicable "fine-grained" generic components (Jonge, 2003). The parameters of a generic component are modeled using feature models (Beuche et al., 2004). The success of generalization is evaluated by measuring the reuse metrics of the developed generic components (Washizaki et al., 2003), which in many cases may be meaningless or uninformative.

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

3. Introduction to Multidimensional Scaling

Multidimensional Scaling (MDS) is a set of mathematical techniques that allow to uncover hidden structure in data. 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.

One of the most important goals in visualizing data is to learn how near or far points are from each other. Often, it can be done with a scatter plot. However, with a large number of variables, it is very difficult to visualize distances, 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.

There are two large groups of MDS methods: metric MDS and non-metric MDS. In metric MDS, distances between objects in multidimensional space are related to their dissimilarities linearly. In non-metric MDS, proximity is considered as a monotonous function of distance, so that the *order* of distances reflects the order of dissimilarities. Thus, metric scaling uses the actual values of the dissimilarities, while non-metric scaling effectively uses only their ranks.

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. There are several distance metrics as follows.

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 $\delta_{ij} = \left\| X_i - X_j \right\|$.

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

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

where $X_i = \left(x_{i1}, \ldots, x_{in}\right)^{\mathrm{T}}$.

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

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

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

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

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:

$$s = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} w_{ij} \left(d_{ij}(Y) - \delta_{ij} \right)^2} \quad (4)$$

STRESS1:

$$s_1 = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{w_{ij} \left(d_{ij}(Y) - \delta_{ij} \right)^2}{\delta_{ij}^{\ 2}}} \quad (5)$$

SSTRESS:

$$ss = \sqrt{\sum_{i=1}^{k} \sum_{j=i+1}^{k} w_{ij} \left(d_{ij}^{\ 2}(Y) - \delta^2_{\ ij} \right)^2} \quad (6)$$

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

There is no definite rule to determine, which stress value represents good or poor goodness-of-fit. Kruskal (1964) 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 the following 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:

$$c_j = \left(f_1, f_2, \ldots, f_n\right), f_i \in F, c_j \in C \quad (7)$$

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

$$f_i = \left\| c_j \right\| \quad (8)$$

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:

$$\delta\left(C_j, C_k\right) = \sqrt{\sum_{r=1}^{m} \left(f_{ir} - f_{jr}\right)^2} \quad (9)$$

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

$$K = \left\{C_j, \ldots, C_k\right\}, \delta(C_j, C_k) \leq const, \textit{ for each pair of components in cluster } K \quad (10)$$

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

$$g = stress(C) \quad (11)$$

In this paper, 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 components 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 feature matrix (component × feature). The feature matrix must include at least 6 features. It

represents a set of points in a multidimensional feature space.

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 on 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 (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 next Section, we present the experimental validation of the proposed MDS-based component analysis framework.

5. Experimental validation of the proposed method

For our analysis, we have selected a Buffer library, which is a part of JDK 1.5 class library (package *java.nio.**). Buffer class 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 detailed description, see (Jarzabek and Li, 2003).

The class hierarchy of the Buffer library is organized in 3 levels as follows (Table 1).

At Level 1 in the class hierarchy, there are seven 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. A Java programmer can directly use only Level 1 classes.

At Level 2 in the class hierarchy, classes implement two memory allocation schemes and two byte orderings. In the direct memory allocation scheme, a contiguous memory block is allocated for a buffer and native access methods are used to read and write buffer elements, using a native or non-native byte-ordering scheme. In the non-direct memory allocation scheme, a buffer is accesses through Java array access methods. Byte ordering matters for buffers, whose elements consist of multiple bytes. There are two byte orderings supported: Little Endian ordering and Big Endian ordering. Twenty 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.

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

The Buffer library contains many “look-alike” components and has a great deal of redundancy (Jarzabek and Li, 2003). 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, 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 source code properties of the components) or semantic (based on functionality of the components).

We have examined two large groups of component features: syntactical features and semantical features.

Syntactical features based on 14 commonly software metrics as follows:

1) *LOC* (Lines of Code),

2) *eLOC* (Effective LOC),

3) *lLOC* (Logical Statements LOC),

4) *Interface Complexity* (Parameters),

5) *Interface Complexity* (Returns),

6) *Cyclomatic Complexity* (Logical Branching),

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 based on 6 identified functional, data and class type parameters as follows:

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 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).

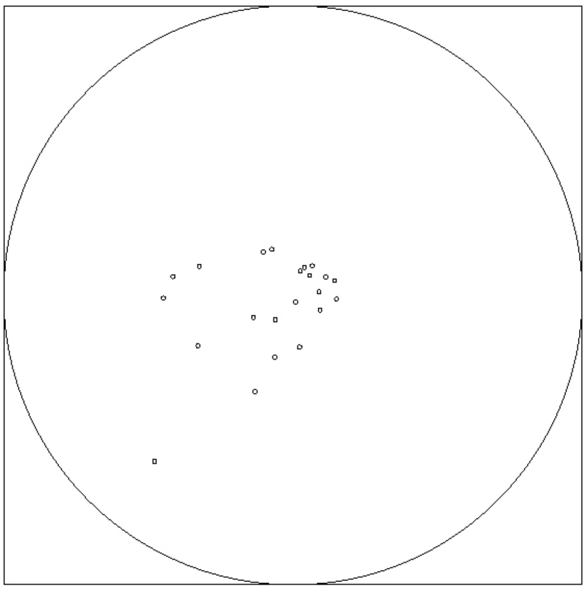

Figure 1. MDS results using syntactic features (Metric MDS; Euclidean distance metric; STRESS criterion)

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. 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.

Table 1. Stress values for metric MDS and different distance metrics

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

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.

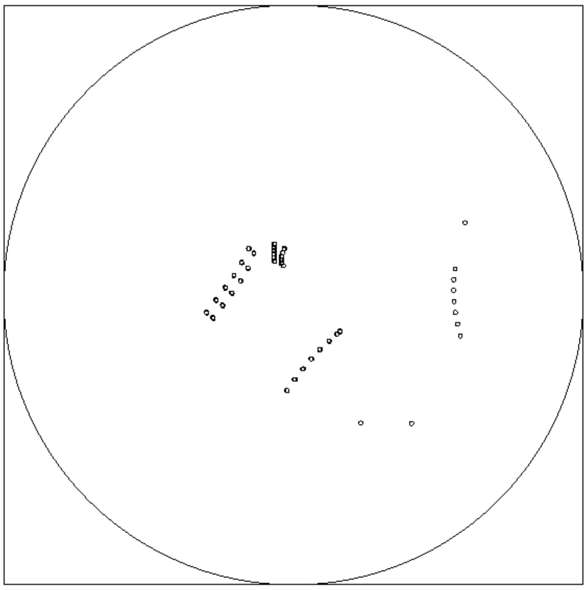

Figure 2. MDS results using semantic features (Metric MDS; Euclidean distance metric; STRESS criterion)

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 1.

We estimate the generalizability of the Java Buffer class library according to the Kruskal's rules-of thumb (Kruskal, 1964) as "excellent" (best stress value < 0.025).

Finally, we present a partitioning of the Buffer library based on the MDS results obtained in the semantic feature space in Figure 3. 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).

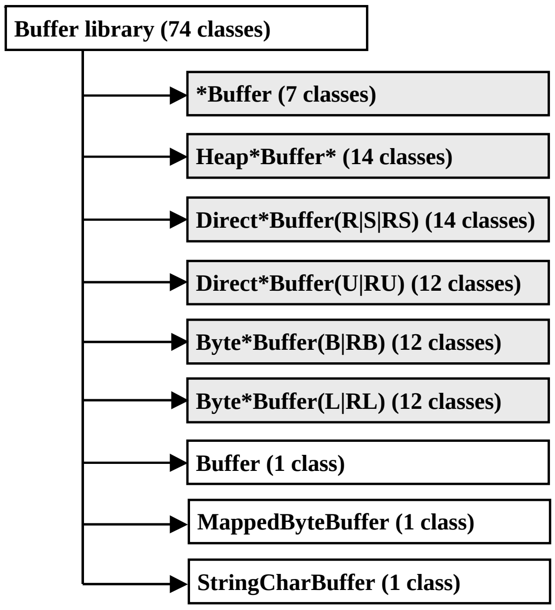
Buffer library (74 classes)
*Buffer (7 classes)
Heap*Buffer* (14 classes)
Direct*Buffer(R|S|RS) (14 classes)
Direct*Buffer(U|RU) (12 classes)
Byte*Buffer(B|RB) (12 classes)
Byte*Buffer(L|RL) (12 classes)
Buffer (1 class)
MappedByteBuffer (1 class)
StringCharBuffer (1 class)

Figure 3. Decomposition of design space

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

6. Related works and evaluation of results

Though many document search and retrieval systems such as internet search engines and large document databases consider and compute similarity of documents, similarity of software components is not a widely covered topic. Amongst the related work we can mention:

Manber (1994) identifies similar software files using a fingerprinting technique. Similarity is understood in purely syntactic terms. Files containing similar information but using different words will not be considered similar.

Merkl *et al.* (1994) propose to structure libraries of reusable software components according to the semantic similarity of the software components using artificial neural network. The result is represented as a self-organizing map (SOM) and used for more convenient component storage and maintenance.

Jarzabek and Li (2003) also analyze the problem of decomposing a set of components for generalization using Java Buffer class library as a benchmark. However, the result is achieved heuristically and no systematic method is proposed, no evaluation of the selected component partitioning is given.

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 prime 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.

7. Conclusions and Future work

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; (3) to evaluate the generalizability of the given set of software components.

The future work will focus on the development of feature-independent similarity metric and adoption of other mathematical and statistical data analysis, visualization and clusterization methods for component-based software engineering and, particularly, for generic component design.

References

Basit, H.A., and Jarzabek, S. (2005). "Detecting higher-level similarity patterns in programs", *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2005),* Lisbon, Portugal, September 5-9, 2005, pp. 156-165.

Baum, L., and Becker, M. (2000). "Generic Components to Foster Reuse", *Proc. of the 4th IEEE International Conference on Tools of Object-Oriented Languages and Systems (TOOLS-Pacific 2000),* Sydney, Australia, pp. 266-277.

Becker, M. (2001). "Generic Components – A Symbiosis of Paradigms", in Butler, G., Jarzabek, S. (Eds.): *Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000,* Erfurt, Germany, October 9-12, 2000. Lecture Notes in Computer Science, Vol. 2177, Springer, pp. 100-113.

Becks, A., Sklorz, S., and Jarke, M. (2000). "A Modular Approach for Exploring the Semantic Structure of Technical Document Collections", *Proc. of AVI 2000, International Working Conference on Advanced Visual Interfaces,* May 23-26, Palermo, Italy, pp. 298–301.

Beuche, D., Papajewski, H., and Schröder-Preikschat, W. (2004). "Variability management with feature models", *Sci. Comput. Program,* 53(3): 333-352.

Beyer, D., and Noack, A. (2005). "Clustering Software Artifacts Based on Frequent Common Changes", *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05),* 15-16 May 2005, St. Louis, MO, pp. 259-268.

Biggerstaff, T.J. (1994). "The library scaling problem and the limits of concrete component reuse", In Frakes, W. (ed.): *Proc. of 3rd International Conference on Software Reuse*, pp. 102–109. IEEE Computer Society Press.

Borg, I., and Groenen, P. (1997). *Modern Multidimensional Scaling*. Springer.

Cox, T.F, and Cox, M.A. (2001). *Multidimensional Scaling.* Chapman and Hall/CRC.

Damaševičius, R., and Štuikys, V. (2002). “Separation of Concerns in Multi-language Specifications”, INFORMATICA, Vol. 13, No. 3, pp. 255-274.

de Jonge, M. (2003). "Package-Based Software Development," *Proc. of 29th Euromicro Conference (EUROMICRO'03),* 3-5 September 2003, Belek-Antalya, Turkey. IEEE Computer Society, pp. 76-85.

Estublier, J. and Vega, G. (2005). “Reuse and Variability in Large Software Applications”, *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC-FSE’05),* September 5–9, 2005, Lisbon, Portugal, pp. 316-325.

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

Frakes, W,B., and Kang, K. (2005). "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529-536.

Haveliwala, T., Gionis, A., Klein, D., and Indyk, P. (2002). "Evaluating Strategies for Similarity Search on the Web," *Proc. of the 11th International World Wide Web Conference*, May 2002, pp. 432-442.

Jarzabek, S. and Li, S. (2003). "Eliminating Redundancies with a “Composition with Adaptation” Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246.

Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (2006). “Generative Programming and Component Engineering”, *Proc. of 5th International Conference, GPCE 2006*, Portland, Oregon, USA, October 22-26, 2006.

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

Kruskal, J.B. (1964). “Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis”, Psychometrika, Vol. 29, No. 1.

Manber, U. (1994). “Finding similar files in a large file system,” *Proceedings of the USENIX Winter 1994 Technical Conference,* San Fransisco, CA, pp. 1–10.

Merkl, D., Min Tjoa, A., and Kappel, G. (1994). “Learning the Semantic Similarity of Reusable Software Components,” *Proc. of the 3rd International Conference on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil, pp. 33-41.

Munoz, A., and Martin-Merino, M. (2002). “New asymmetric iterative scaling models for the generation of textual word maps”, *Proc. of JADT 2002: 6es Journ´ees internationales d’Analyse statistique des Donn´ees Textuelles.*

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

Popescul, A., Flake, G.W., Lawrence, S., Ungar, L.H., and Lee Giles C. (2000). “Clustering and Identifying Temporal Trends in Document Databases”, *Proc. IEEE Advances in Digital Libraries ADL 2000*, Washington, DC, pp. 173-182.

Sadaoui, S., and Yin, P. (2004). “Generalization for component reuse”, *Proceedings of the 42nd annual Southeast regional conference*, Huntsville, Alabama, pp. 134 - 139.

Stein, B., and Meyer zu Eissen, S. (2003). “Automatic Document Categorization: Interpreting the Performance of Clustering Algorithms”, In Günter, Kruse, Neumann (Eds.): *Advances in Artificial Intelligence*. LNAI 2821, pp. 254-266, Springer.

Szyperski, C. (1999). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

Washizaki, H., Yamamoto, H., and Fukazawa Y. (2003). “A Metrics Suite for Measuring Reusability of Software Components”, *Proc. of 9th IEEE International Software Metrics Symposium (METRICS 2003),* 3-5 September 2003, Sydney, Australia, pp. 211-224.

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