## A Connectionist View on Document Classification

## Dieter Merkl

Department of Information Engineering, University of Vienna Liebiggasse 4/3, A-1010 Vienna, AUSTRIA

dieter@ifs.univie.ac.at

### Abstract

Properly structured software libraries are crucial for the success of software reuse. Specifically, the structure of the software library ought to reflect the functional similarity of the stored software components in order to facilitate the retrieval process. We propose the application of artificial neural network technology to achieve such a structured library. In more detail, we rely on full-text indexing of the software manual in order to obtain the software representation. This software representation is further used as the input data during the training process of an artificial neural network adhering to the unsupervised learning paradigm. The distinctive feature of this very model is to make the semantic relationship between the stored software components geographically explicit. Thus, the actual user of the software library gets a notion of the semantic relationship between the components in terms of their geographical closeness.

## **1** Introduction

Software reuse is concerned with the technological and organizational issues of using already existing software components to build new applications. There is common agreement that the reuse of software increases both the productivity of the software suppliers and the quality of the software itself. The former is due to the amount of time that is saved each time a component is reused. The latter is due to the fact that the same component is used and tested in many different contexts. However, in order to make software reuse operational, the actual software developers ought to be equipped with libraries containing reusable software components. Concerning these so-called software libraries, the most stringent requirements are the following.

• Software libraries should provide a large number of reusable components in a wide spectrum of application domains. These components may be either reused as they are or may easily be adapted to the needs of the application currently under consideration.

• Software libraries should be organized in such a way that locating the most appropriate component is easy for the actual user. Particularly, the library should provide assistance to the user in locating components that meet some specified functionality.

In this paper we address the second requirement, namely structuring the contents of software libraries in such a way that locating the needed component is facilitated. For a more detailed discussion of other aspects of software reuse we refer to [2], [13], [22].

In the remainder of this paper we will describe an approach relying on artificial neural network technology to achieve a semantically structured software library. By the term semantically structured we denote structuring according to the functional similarity of the stored software components. In other words, components that exhibit similar behavior should be stored near to each other and thus be identifiable. From a rather global point of view, our approach employs a keyword-based software representation where the various keywords are extracted automatically from the manual of the software components. Specifically, each component is described by using a set of terms extracted from the full-text of the respective part of the manual. During the indexing process we utilize only a small list of stop-words to clean up the resulting index. Subsequently, each software component is represented as a binarily-valued vector according to the vector space model. These vectors are further used as the input data for an artificial neural network adhering to the unsupervised learning paradigm. The artificial neural network is responsible for the structuring of the software library.

The rest of this paper is organized as follows. In the next section we will provide a brief overview on related work concerning software representation and library structuring. In Section 3 we will describe the architecture and the learning rule of the artificial neural network which we used for our experiments. Section 4 in its turn will contain the results of structuring a software library. Finally, in Section 5 we will draw some conclusions.

### 2 Related Work

Related work may roughly be assigned to two groups. First, approaches to classify reusable software components and second, approaches to structure software libraries.

An approach to software classification which has received much attention is faceted classification as proposed in [24], [25]. This classification schema may be described as consisting of a set of categories, i.e. the so-called facets, each of which has several terms that may possibly be filled in. Thus, the classification schema remains flexible with respect to extensibility in such a way that components derived from new applications may easily be classified by using the same categories, yet with adjusted terms. Only recently, faceted classification has been adapted to meet the requirements of object-oriented software components [9], [27], [28]. Other approaches rely on indexing techniques from the area of information retrieval, namely manual assignment of keywords [1], scanning the source code and extracting comments [3], scanning the full-text of the documentation and assigning document descriptors automatically [4], [5]. In [15] an approach is described that relies on the concept of lexical affinities which may be paraphrased as the selection of pairs of words that occur frequently together within one sentence of the documentation [14]. These pairs of words are further used as the index to the respective software components. An approach to software classification by using a connectionist model is suggested in [21]. In particular, feed-forward and recurrent artificial neural networks are used to assign comments and identifiers which are extracted from the source code to concepts. Thus, the artificial neural network performs an analysis of the informal information contained in a program.

The organization of software libraries is commonly performed either by applying techniques of information retrieval, namely cluster analysis, or by utilizing semantic networks to model the semantic similarity between components. An approach adhering to information retrieval techniques is described in [15]. The authors describe the organization of a library containing operating software system commands by using hierarchical agglomerative clustering methods. The clustering is based on the lexical affinities extracted from the software documentation as described above. An approach combining faceted classification with semantic networks is reported in [23]. More precisely, the similarity between components is indicated by using weighted connections within a semantic network which describes the various facets of the software classification schema.

### **3** Self-Organizing Feature Maps

The self-organizing feature map [10], [11], [12] is one of the major unsupervised learning models in the area of artificial neural networks. Unsupervised learning may roughly be described as adapting the structure of an artificial neural network to enable the construction of internal models that capture regularities present in the input domain without any additional information beyond the input data.

The self-organizing feature map consists of a layer of n so-called input units and a grid of so-called output units. Each input unit is connected to every output unit. The purpose of the input layer is to collect the input data in form of an n-dimensional feature vector and to propagate this input vector onto the output units. Concerning the output units, each of them is assigned a weight vector with *n* weight components. Initially, the weight components are assigned random values in the range of [0, 1]. Each of the output units produces one output value which is proportional to the similarity between the current input vector and the unit's weight vector. This value is commonly referred to as the unit's activation or the unit's response to a given input. Similarity between the vectors is usually measured in terms of Euclidean distance.

The adaptation of the weight vectors is performed in an unsupervised learning process. This process may be described in three steps which are performed repeatedly. Together, these three steps are referred to as one learning iteration. First, one input vector at a time is randomly selected out of the set of input vectors. Second, this input vector is mapped onto the grid of output units of the self-organizing feature map and the unit with the strongest response is determined. This unit is referred to as the winning unit or the bestmatching unit, the winner in short. Third, the weight vector of the winning unit as well as the weight vectors of units in topological neighborhood of the winning unit are adapted in such a way that these units will exhibit an even stronger response with the same input vector in future.

More formally, we may describe the learning rule of a self-organizing feature map as follows.

- (1) Random selection of one input-vector *x*.
- (2) Determination of the best matching unit *i* by using the Euclidean distance measure.

$$i: \|w_i - x\| \le \|w_i - x\| \quad \forall j \in O$$

In this formula  $w_i(w_j)$  denotes the weight vector assigned to unit i(j) in output space O.

(3) Adaptation of the weight vectors  $w_j$  in the neighborhood of the best matching unit *i*.

 $\Delta w_i = \varepsilon \cdot \delta(i, j) \cdot [x - w_i] \ \forall j \in O$ 

The strength of the adaptation is determined with respect to parameter  $\varepsilon$ , i.e. learning-rate, the neighborhood relation between the best matching unit *i* and the output unit *j* which is currently under

consideration, and the difference between input vector x and weight vector  $w_j$  assigned to unit j. The function  $\delta$  has to guarantee the following property: the larger the distance between units i and j, i.e. ||i-j|| in the output space O, the smaller the adaptation of the weight vector  $w_i$ .

(4) Repeat steps (1) thru (3) until no more changes to the weight vectors are observed.

To guarantee convergence of the map the learningrate  $\varepsilon$  as well as the adapted neighborhood have to shrink in the course of time. Hence, the adaptation of the weights as well as the set of units that are subject to weight changes decrease gradually.

During the numerous repetitions of these three steps of the learning process, the weight vectors get better approximations to the input distribution. Furthermore, neighboring units respond similarly to input vectors, thus resulting in a global order. In other words, the weight vectors of the units are tuned to specific features of the input space in such a way that topological relations which are present in the input domain are retained as faithfully as possible in the output space, i.e. the grid of output units. Apparently, due to the mapping from a high dimensional input space to a much lower dimensional output space some distortion is unavoidable. Yet, nearby vectors in the input space are mapped onto nearby units in the output space. To conclude, the response of the self-organizing map to the presentation of an input vector is a localized pattern of activity which is similar for closely related input vectors.

## 4 Structuring Software Libraries

### 4.1 The Software Representation

Throughout the remainder of this paper we will use the NIH Class Library, the NIHCL in short, as an example of a software library. NIH is the acronym for National Institutes of Health, a US governmental organization. The NIHCL is a collection of classes developed in the C++ programming language. The NIH Class Library includes generally useful data types such as String, Date, and Time, as well as a number of container classes such as OrderedCltn, LinkedList, Set, Dictionary which are similar to the Smalltalk-80 classes. Moreover, the NIHCL provides the facilities to store arbitrarily complex data structures comprised of NIH Library and user-defined objects on disk. For more detailed information about the NIH Class Library we refer to [6] and to the reference manual of Version 3.10 [7].

In order to build the software representation we rely on the textual description of the various classes that are contained in the NIHCL. As an illustrative example consider Figure 1 containing the description of the class Set as contained in [7]. These manual entries are further full-text indexed by omitting only a very small set of stop-words. The stop-word list, in fact, comprises only articles, conjunctions, and pronouns. All other words are included in the index of the respective class. Just to give the concrete figure, 489 distinct terms were extracted from the various textual descriptions of the classes.

# Set - Unordered Collection of Non-Duplicate Objects

Base Class: Collection Derived Classes: Dictionary, IdentSet Related Classes: Iterator

A Set is an unordered collection of objects. The objects cannot be accessed by a key as can the objects in a Dictionary, for example. Unlike a Bag, in which equal objects may occur more than once, a Set ignores attemps to add any object that duplicates one already in the Set. A Set considers two objects to be duplicates if they are isEqual() to one another.

Class Set is implemented using a hash table with open addressing. Set::add() calls the virtual member function hash() and uses the number returned to compute an index to the hash table at which to begin searching, and isEqual() is called to check for equality.

### Figure 1: NIHCL manual entry of class Set

To provide the means to interpret the results of library structuring we show the NIHCL class hierarchy in Figure 2. Furthermore, we provide a graph representing the 'related-class' relationship in Figure 3 which is derived from the respective sections in the manual.

The final software representation follows the vector space model where input data and queries are represented as vectors in an *n*-dimensional hyperspace [29]. More specifically, each component of the vectors corresponds to a possible document feature, i.e. index term, and is binarily-valued. Thus an entry of zero denotes the fact that the corresponding feature is not used in the description of the software component. Contrary to that, an entry of one means that the corresponding feature is used to describe the software component. To conclude, the software components are represented by vectors of a 489-dimensional hyperspace. These feature vectors are used as the input data during the learning process of the self-organizing feature map.

### 4.2 The Structured Software Library

Based on the component representation as outlined above we trained a self-organizing feature map to structure the classes of the NIHCL according to their functional relationship. The main reasons for choosing



Figure 2: NIHCL class hierarchy

ArrayOb	lterator		LookupKey	Dictionary	IdentDictKeySortCltn
Assoc	Dictionary	IdentDict KeySortCltn	OrderedCltn	Iterator	
AssocInt	Dictionary	IdentDict Integer	Range	Regex	String
Bag	AssocInt	Iterator Set	Regex	Range	
Collection	Iterator		SeqCltn	Iterator	
Date	Time		Set	Iterator	
Dictionary	Assoc	Iterator LookupKey	SortedCltn	Iterator	Range
Heap	Iterator		Stack	Iterator	
IdentDict	Assoc	Iterator LookupKey	Time	Date	
IdentSet	Iterator		OlOifd	OlOofd	
Integer	AssocInt		OlOistream	OlOostream	
KeySortCltn	Assoc	lterator LookupKey	OlOnihin	OlOnihout	
Link	LinkedList		OlOin	OlOout	
LinkedList	Iterator	Link LinkOb	ReadFromTbl	StoreOnTbl	
LinkOb -	LinkedList				
Iterator	ArrayOb	Bag Collection	Dictionary	Неар	]
·	IdentDict	IdentSet KeySortCltn	LinkedList	OrderedCltn	
	SeqCltn	SortedCltn Stack			
		,,			

Figure 3: NIHCL related classes

an artificial neural network to structure the contents of software libraries are the following. First, artificial neural networks are robust in the sense of tolerating noisy or inexact input data. This property is especially important during query processing simply because a query may be regarded as an inexact representation of the needed software component. Such inexact queries appear rather common since the software developer might have only a limited idea of what she is actually searching for or of the universe of components stored in the library. Second, artificial neural networks serve as associative memories. In other words, such models are capable of retrieving software components when supplied just with the description of the needed component. Finally, artificial neural networks exhibit the ability of generalization. Pragmatically speaking, an artificial neural network learns conceptually relevant features of the input domain and is thus able to structure the software library accordingly.

From the large number of different artificial neural network models proposed in literature we selected the self-organizing feature map as the architecture to represent the software library. The motivation to utilize exactly this architecture is due to its learning rule and additionally, to its ability to visualize the relationship of the various software components in terms of the two-dimensional grid of output units. Concerning the learning rule, we feel that unsupervised learning is convenient for that kind of application since the adaptation of the artificial neural network is based exclusively on the input data and no additional information is required to guide the learning process. With additional information we refer to the desired output of the network which has to be specified when using supervised learning as for example with the multi-layer Perceptron. The benefit of such an unsupervised learning rule is most obvious in case of already existing yet unsatisfactorily structured software libraries where the definition of input-output-mappings the proper beforehand certainly is a laborious task. However, similar results might be achievable with other architectures, too. A unique characteristic of self-organizing feature maps is the representation of intra- and intercluster relationships. In less bulky terms we may state that the property which makes the self-organizing feature map well suited to our application is its ability to visualize the relationship of the input data both within and between clusters. More precisely, the semantic similarity of the input data corresponds to the distance of the respective software components within the grid of output units. Contrary to that, other artificial neural network models behave merely as predicates whether or not a given component is a member of a certain cluster.

In the following we will present some results

which were obtained by applying the self-organizing feature map to the component description. The tests were performed with rectangular two-dimensional self-organizing feature maps. The graphical representation of the result contains the final position of the various NIH classes after the learning process. These results may be interpreted as follows. Each entry in the graphical representation corresponds to an output unit of the self-organizing feature map. Each of the units is either assigned the name of a NIH class or a dot. The assignment of a class denotes the fact that this very unit exhibits the highest activation level with respect to the component description corresponding to the class, i.e. the weight vector of this unit has the smallest Euclidean distance to the component description in terms of the input vector. The assignment of a dot means that none of the classes are assigned to that unit, i.e. the unit does not exhibit for any class the highest activation level. Note that the topological arrangement of the NIH classes in the map is an indication of the semantic similarity between the various classes.

As a first result consider the map as contained in Figure 4. This final arrangement was obtained with an initial value of the learning-rate  $\varepsilon$  of 0.7. In order to arrive at the stable state the self-organizing feature map needed 8,000 learning iterations. So much for the technical details of the learning process.

Perhaps even more exciting is the final arrangement of the various classes within the twodimensional grid of output units. For the ease of comparison we marked some of the clusters manually. In general, the arrangement of the classes may be regarded as a combination of the class hierarchy and the 'related-class' relationship. Consider for example the left upper part of the map as depicted in Figure 4. This region consists of all classes implementing I/O operations. Moreover, within this cluster the classes are arranged according to the 'related class' relationship as indicated in their respective descriptions in the manual. For another cluster of related classes we refer either to the right lower corner of the map containing Range, Regex, and String or to the directly neighboring cluster with Time and Date. However, the largest portion of the map is reserved to class Collection and its derived and related classes. This cluster is located in the left lower and right upper corner of the map, respectively. Such a separation appears quite natural since the former contains virtually all sequential collections whereas the latter comprises the unordered collections. The only exception is raised by class KeySortCltn which is in fact a sequential collection. Yet, due to its very nature of providing keyed access this class is closely related to class Dictionary. It is exactly this relationship that is visualized in the right upper

corner of the final map. Finally, we want to direct the attention to yet another cluster which is neither obvious from the class hierarchy nor from the 'relatedclass' relationship. Particularly, we refer to the region containing the classes Float and Integer. Both classes are used to implement basic numerical data types and thus have a highly similar description in the manual.

For another result consider the final map as depicted in Figure 5. This time we used the same learning parameters as above. However, we utilized a slightly modified learning rule. More precisely, the modifications are related to the adaptation of the weight vectors. The original learning function of selforganizing feature maps only moves the weight vectors of the winner and neighboring units towards the current input vector. In other words, the weight vector of a unit is either adapted in such a way that its distance to the current input vector decreases or the weight vector is not changed at all. For a concrete realization of such a function we refer to [26]. Contrary to that, we propose with our modified learning function a more biologically plausible process, namely the inclusion of lateral inhibition of output units [18]. Roughly speaking, lateral inhibition is a process where the activation of one neuron blocks the activation of other neurons. Thus, the activation of one neuron reduces the inclination of activating other neurons. This phenomenon is observed, for example, in the visual system of the brain [8]. In terms of the self-organizing feature map as described above, the activation of a unit corresponds to the similarity

between the unit's weight vector and the current input vector. Hence, by using the term 'blocking the activation of units' we denote the enlargement of the distance between the respective vectors. Thus, the activation of this unit will be smaller in future with respect to the current input. We perform lateral inhibition by using the following neighborhood function.

### $\delta(i, j) = \beta(t) \cdot \sin(\alpha \cdot ||i-j||) / (\alpha \cdot ||i-j||)$

This neighborhood function  $\delta(i, j)$  determines the strength of adaptation of the weight vectors  $w_i$  based on the distance between the winner *i* and unit *j*, i.e. ||i*i*||. The parameter  $\alpha$  is used to influence the width of the function in terms of lateral excitation, i.e. adaptation towards the input, and lateral inhibition, i.e. adaptation away from the current input vector. Pragmatically speaking, units in close vicinity to the winning unit are pulled towards the current input whereas distant units are pushed slightly away. Finally, the time-varying parameter  $\beta$  is used similarly to the learning-rate  $\varepsilon$ . This parameter represents an inhibition-rate which determines the strength of the adaptation. In analogy to the learning-rate,  $\beta$  is limited within the range of [0, 1] decreasing gradually in time, i.e. with increasing learning iterations.

The first observation regarding the final map of Figure 5 relates to the time needed to train the selforganizing feature map. Particularly, due to the modifications of the learning rule, the artificial neural network needed only 7,000 learning iterations to arrive at the stable state.

On a closer look at the final map as shown in





Figure 4: 10×10 map of the NIHCL

Figure 5: 10×10 map of the NIHCL

Figure 5 we notice that similar clusters to Figure 4 were formed. Again the keyed data structures such as Dictionary, IndentDict and KeySortCltn are grouped together. Additionally, the classes implementing such a keyed access are arranged in close vicinity, i.e. Assoc, AssocInt, and LookupKey. Similarly, the classes contained in the I/O cluster mirror the 'related-class' relationship. The remaining clusters of Figure 4 have their counterparts in Figure 5 as well. Just to direct the attention to yet another observation, class Random is mapped neighboring of Float. As may be guessed by the names of the classes, Random produces pseudorandom numbers of Float data type.

A more expansive view on the learning process of self-organizing feature maps as well as examples of the retrieval efficiency as compared to cluster analysis is contained in [19]. The results of structuring other software libraries are reported in [16] and [17]. A more detailed exposition of the effects related to the learning functions is published in [20].

### 4.3 A Comparison with Cluster Analysis

In this subsection we address the comparison of the results obtained by using the self-organizing feature map with results obtained by applying a more traditional approach to structure software libraries, namely hierarchical agglomerative clustering. In order to make the results comparable we utilized the same software representation as input to the cluster algorithm. Again, the cluster algorithm is based on the Euclidean distance between the various vectors representing the software components. Tests were performed with single linkage, complete linkage, and Ward method. As expected, single linkage completely failed to uncover any meaningful clusters. Since complete linkage and Ward method produced comparable results, we will summarize the former in this subsection.

The following figures present the arrangement of classes within a varying number of clusters. In particular, Figure 6 provides the assignment of the classes to six clusters by using complete linkage. At first sight, we observe a rather large cluster labeled #1. This cluster contains more than half of the NIHCL classes. As expected, cluster #1 comprises a highly unrelated mix of classes. More precisely, most of the data structure classes are merged with the classes for I/O operations. Moreover, the classes for keyed-access are assigned to cluster #2 together with a number of other, yet unrelated, classes.

With more clusters complete linkage tends to produce more isolated clusters. In other words, a number of clusters are formed containing one class only. Even worse, cluster analysis fails in uncovering the close relationship of class Iterator to the various container classes. Moreover, the data structure

- (#1) Arraychar, Arrayob, Bag, Bitset, Class, Heap, IdentDict, IdentSet, KeySortCltn, LinkedList, LinkOb, OlOifd, OlOin, OlOistream, OlOnihin, OlOnihout, OlOofd, OlOostream, OlOout, OrderedCltn, ReadFromTbl, Regex, SeqCltn, Set, SortedCltn, Stack, StoreOnTbl
- (#2) Assoc, AssocInt, Date, Dictionary, Exception, FDSet, Float, Integer, Link, LookupKey, Nil, Point, Random, Range
- (#3) Collection, Iterator
- (#4) Object
- (#5) String
- (#6) Vector

(#1)	Arraychar, Arrayob, Bag, Bitset, KeySortCltn, LinkedList, OlOifd, OlOin, OlOistream, OlOnihin, OlOnihout, OlOofd, OlOostream, OlOout, OrderedCltn, Regex, SeqCltn, SortedCltn, Stack
(#2)	Assoc, AssocInt, Date, Dictionary, Exception, FDSet, Float, Integer, Link, LookupKey, Nil, Point, Random, Range, Time
(#3)	Class, ReadFromTbl, StoreOnTbl
(#4)	Collection
(#5)	Heap, IdentDict, IdentSet, Set
(#6)	Iterator
(#7)	LinkOb
(#8)	Object
(#9)	String
(#10)	Vector

and I/O classes are still assigned to the same cluster as can be observed from Figure 7 containing the assignment of NIHCL classes to ten clusters. As yet another deficiency of complete linkage consider the classes representing collections with keyed-access. These classes are spread over three distinct clusters, namely KeySortCltn is included in cluster #1, Dictionary in cluster #2, and IdentDict in cluster #5.

To summarize the observations, the self-organizing feature map is obviously superior in uncovering the relations within and between the various clusters. In other words, the self-organizing feature map is less susceptible to noisy input data which is common in an environment relying on full-text indexing with a limited stop-word list.

### 5 Conclusion

In this paper we described a novel approach to structure the contents of a software library according to the functional similarity of the stored software components. As an illustrative example we used a C++ class library. The software representation of the various classes was built by full-text indexing of the textual description of the classes as contained in the software manual. The resulting index was cleaned up by means of a small-sized stop-word list. This software description was used as the input data during the learning process of a self-organizing feature map, i.e. an artificial neural network adhering to the unsupervised learning paradigm. As a result of the learning process, the self-organizing feature map proved to be successful in structuring the software library according to the intended functionality of the various components. The final arrangement of the classes resembled their association as provided by the inheritance hierarchy and а 'related-class' relationship. Moreover, the results were clearly superior to those obtained by means of hierarchical agglomerative clustering of the same input data. This fact is especially remarkable since hierarchical cluster analysis represents the traditional way to achieve structured software libraries in case of a software representation that is based on keywords extracted automatically from the software manual.

### Acknowledgment

Thanks are due to Brigitta Galian for her patient assistance in shaping the English.

### References

- S. P. Arnold and S. L. Stepoway. The REUSE System: Cataloging and Retrieval of Reusable Software. *Proceedings of the IEEE Int'l Computer Conference* (COMPCON'87). 1987.
- [2] T. J. Biggerstaff and A. J. Perlis (Eds.). Software Reusability. Volume I: Concepts and Models. Volume II: Applications and Experience. Addison-Wesley. Reading, MA. 1989.
- [3] B. A. Burton, R. Wienk Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes. The Reusable Software Library. *IEEE Software* 4(7). 1987.
- [4] Y. F. Chang and C. M. Eastman. An Information Retrieval System for Reusable Software. *Information Processing & Management* 29(5). 1993.
- [5] W. B. Frakes and B. A. Nejmeh. An Information System for Software Reuse. *Proceedings of the* 10th Minnowbrook Workshop on Software Reuse. 1987.
- [6] K. E. Gorlen, S. Orlow, and P. Plexico. Data Abstraction and Object-Oriented Programming in C++. John Wiley & Sons. New York. 1990.
- [7] K. E. Gorlen. NIH Class Library Reference Manual (Revision 3.10). National Institutes of Health. Bethesda, MD. 1990.
- [8] E. R. Kandel, S. A. Siegelbaum, and J. H. Schwartz. Synaptic Transmission. in: *Principles* of Neural Science (E. R. Kandel, J. H. Schwartz, and T. M. Jessel, Eds.). Elsevier. New York. 1991.
- [9] E.-A. Karlsson, S. Sørumgård, and E. Tryggeseth. Classification of Object-Oriented Components for Reuse. Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 7). Dortmund. Germany. 1992.
- [10] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics* 43. 1982.
- [11] T. Kohonen. Self-Organization and Associative Memory (3rd edition). Springer. Berlin. 1989.
- [12] T. Kohonen. The Self-Organizing Map. *Proceedings of the IEEE* 78(9). 1990.
- [13] C. W. Krueger. Software Reuse. ACM Computing Surveys 24(2). 1992.
- [14] Y. S. Maarek and F. A. Smadja. Full Text Indexing Based on Lexical Relations - An Application: Software Libraries. *Proceedings of* the 12th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval. 1989.
- [15] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering* 17(8). 1991.

- [16] D. Merkl. Structuring Software for Reuse The Case of Self-Organizing Maps. Proceedings of the Int'l Joint Conference on Neural Networks (IJCNN'93). Nagoya. Japan. 1993.
- [17] D. Merkl, A M. Tjoa, and G. Kappel. A Self-Organizing Map that Learns the Semantic Similarity of Reusable Software Components. *Proceedings of the 5th Australian Conference on Neural Networks* (ACNN'94). Brisbane. 1994.
- [18] D. Merkl, A M. Tjoa, and G. Kappel. Application of Self-Organizing Feature Maps With Lateral Inhibition to Structure a Library of Reusable Software Components. *Proceedings of the IEEE Int'l Conference on Neural Networks* (ICNN'94). Orlando, FL. 1994.
- [19] D. Merkl, A M. Tjoa, and G. Kappel. Learning the Semantic Similarity of Reusable Software Components. *Proceedings of the 3rd Int'l Conference on Software Reuse*. Rio de Janeiro. IEEE Computer Society Press. 1994.
- [20] D. Merkl. The Effects of Lateral Inhibition on Learning Speed and Precision of a Self-Organizing Feature Map. Proceedings of the 6th Australian Conference on Neural Networks (ACNN'95). Sydney. 1995.
- [21] E. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionist models. *Proceedings of the 13th Int'l Joint Conference on Artificial Intelligence* (IJCAI'93). Chambéry. France. 1993.

- [22] R. T. Mittermeir and W. Rossak. Reusability. in: Modern Software Engineering - Foundations and Current Perspectives (P. A. Ng and R. T. Yeh, Eds.). Van Nostrand Reinhold. New York. 1990.
- [23] E. Ostertag, J. Hendler, R. Prieto-Díaz, and C. Braun. Computing Similarity in a Reuse Library. ACM Transactions on Software Engineering and Methodology 1(3). 1992.
- [24] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software* 4(1). 1987.
- [25] R. Prieto-Díaz. Implementing Faceted Classification for Software Reuse. Communications of the ACM 34(5). 1991.
- [26] H. Ritter and T. Kohonen. Self-Organizing Semantic Maps. *Biological Cybernetics* 61. 1989.
- [27] G. Sindre, E.-A. Karlsson, and P. Paul. Heuristics for Maintaining Term Structures for Relaxed Search. in: *Database and Expert Systems Applications* (A M. Tjoa and I. Ramos, eds.). Springer. Wien. 1992.
- [28] G. Sindre and S. Sørumgård. Terminology evolution in component libraries. *Proceedings of* the 3rd Int'l Congress on Terminology and Knowledge Engineering. Köln. Germany. 1993.
- [29] H. R. Turtle and W. B. Croft. A Comparison of Text Retrieval Models. *The Computer Journal* 35(3). 1992.