SELF-ORGANIZING MAPS AND SOFTWARE REUSE

DIETER MERKL
Institut für Softwaretechnik
Technische Universität Wien
Resselgasse 3/188, A-1040 Wien, Austria
dieter@ifs.tuwien.ac.at

Abstract

Software reuse is the process of building new systems from existing components instead of developing these systems from scratch. For a long time now software reuse is repeatedly acknowledged for playing an essential role in overcoming the so-called software crisis, i.e. the late delivery of then still faulty software products. Current development practice as for example object-oriented analysis, design, and programming should in principle assist the proliferation of the reuse idea. However, before existing components may be considered for reuse they have to be found in a software library. As ever in any area relying on the retrieval of particular objects from a large data store, the process of retrieval may turn out to be rather cumbersome, especially when a large number of objects is contained in the data store and the success of the whole operation is dependent on the retrieval of a small number of relevant objects. With this work we address the assistance of such a retrieval process by means of using a connectionist representation of the contents of the software library. More precisely, we rely on the self-organizing map for software library organization. What makes this model especially attractive for an information retrieval task such as software library organization is the topology preserving learning process leading to a highly intuitive similarity visualization.

1 Introduction

The field of software reuse is concerned with the technological and organizational issues arising from the usage of already existing software components to build new applications. Software reuse is widely acknowledged for equipping the software industry with the tools and techniques to enhance both productivity—due to the substantial amount of work

In W. Pedrycz and J. F. Peters, editors. *Computational Intelligence in Software Engineering.* World Scientific, Singapore, 1998.

that may be saved when a component is reused instead of redeveloped—and quality—due to the fact that the component is used and tested in many different contexts. In other words, software reuse is widely promoted as a panacea for the software crisis, i.e. low productivity of the software industry combined with the inability to satisfy the needs of the customers. For an eloquent treatment of the software crisis and suggestions to address it we refer to [1].

As the technological basis to make software reuse operational, the developers have to be equipped with so-called software libraries that store the various reusable items. In order to be useful, such a library should provide a large number of reusable components within a wide spectrum of application domains. These components may be either reused as they are or may be adapted to the specific needs of the application at hand. We should note that software reuse is not necessarily restricted to the reuse of code. In a much broader sense we regard all products of the software development process as candidates for reuse; examples are process models, requirements analysis, designs, code, documentation, test plans and test data. In this work, however, we restrict ourselves to code as the reusable entities.

Searching for a stored item in a possibly large software library may turn out as a tricky task. The vicious ingredients are an often ill-defined target description of the needed software component on the one hand. On the other hand, the thus following bad initial retrieval results may lead to the "insight" from the users of the library, i.e. the software developers, that developing the needed component from scratch is faster and easier. Consequently, the issue of software library organization is of central concern for the success of the whole idea of software reuse. The basic question is, how should a software library be structured in order to facilitate the retrieval process most? In general, software libraries should be organized in such a way that locating the most appropriate component is easy for the users. Particularly, the library should provide assistance to the user in locating components that meet some specified functionality. A large variety of approaches to software library organization have been suggested in the literature, yet an agreed-upon standard is still not in sight.

In this work we will present an approach to software library organization based on connectionist models. The central aim is to make the semantic relationship of the stored software components as intuitively accessible to the user as possible. By semantic relationship we refer to the functional closeness of the various components which is, obviously, of interest to the actual software developer. As a simple example consider the case where one is looking for a component capable of sorting a number of data items. In general, we would expect a component implementing a *QuickSort* routine to be functionally similar to a component implement-

ing a *MergeSort* routine and a component implementing a *BubbleSort* routine—which of them is best suited for the application at hand may only be determined with deeper knowledge about the system that is to be developed, like the number and the location of the data items to be sorted. We would expect further that these components are functionally dissimilar to a component implementing, say, a string comparison routine such as the *Boyer-Moore algorithm*.

In order to assess the functional similarity of the various software components, we rely on an automatically derived software representation obtained from the textual description of the various components as present in the software manual. We have chosen a keyword-based software representation where the various keywords are extracted by means of full-text indexing of the software manual. Based on such a software representation, we use the self-organizing map, i.e. an unsupervised neural network model, to determine the mutual similarity of the software components and to organize the library accordingly. As a highly convenient benefit, the software components are arranged within a two-dimensional grid of neurons where similarity corresponds to geographical closeness. Hence, the functional similarity of two software components is expressed in terms of spatial closeness of the neurons representing these components—a concept that is intuitively accessible for the user.

The remainder of this work is organized as follows. In Section 2 we give a brief overview of software reuse. A particular aspect of software reuse is described in Section 3 in greater detail, namely approaches to software library organization. Section 4 contains a description of self-organizing maps, the neural network model we rely on for software library organization. The experimental environment for our study in software library organization is described in Section 5. Our approach to software library organization is outlined in Section 6. In particular we give experimental results from organizing a software library with self-organizing maps. We describe an alternative visualization technique that enables a more intuitive similarity representation. Additionally, we show results from using the growing grid model, i.e. a neural network model based on the self-organizing map with adaptive architecture that develops according to the specific requirements of the input data. Finally, we provide some conclusions in Section 7.

2 Software Reuse

Software productivity has been steadily increasing over the past three decades. This increase, however, is still not enough to close the gap between the demands placed on the software industry and what the current state of practice is able to deliver[2,3]. The reuse of software products is one of the few realistic alternatives to bring about the gains of pro-

ductivity and quality that the software industry needs to meet today's challenging requirements. Software reuse involves building software that is reusable in design and building systems with reusable software components. Software reuse includes reusing both the products of previous software projects and the processes developed to produce them. Thus, a wide spectrum of approaches to software reuse have been suggested [4] ranging from *building block* approaches, i.e. reusing and assembling specific products, to *generative* approaches, i.e. reusing processes of previous software development.

The notion of reusability is an old idea that proved to be highly effective in a wide range of applications. As solutions to current problems are found, these solutions are tried in solving similar new problems. Typically, the old solutions are modified, combined, and adapted to solve the new problem. As some solutions are used time and again to solve the same type of problem, these solutions tend to get accepted, generalized, and standardized. Especially the engineering disciplines are full of examples for successful implementation and execution of the reuse idea in an industrial fashion. The development of large and complex systems relies heavily on the reuse of standardized components because it is impossible otherwise. We refer to civil engineering, mechanical engineering, electrical engineering, aircraft construction, etc. for areas that may be characterized by having a highly mature reuse culture. Software engineering is deliberately omitted from this list. The general idea of reusing software components in order to build new systems and the knowledge of its anticipated benefits is well established in literature since three decades, the practical wide-range implementation of this idea, however, is still in its infancy.

Contrary to this knowledge, the typical situation in a large number of development projects is marked by little exchange of information between departments relative to on-going work and production. Hence, when—rarely enough—shared requirements are discovered it is often too late to take advantage of them. Poor coordination between project teams developing similar products causes them to choose their architectures independently, which makes later reuse difficult if not impossible.

The failure to build a few reusable products instead of many specialized products also results in a growing proportion of effort dedicated to maintenance instead of development of new products. Even worse, the proportion of perfective maintenance, i.e. the adaptation of a software system to new or changed requirements, is by far the largest part of the overall maintenance tasks [5]. In other words, much effort is invested in adaptation and evolution of software systems while sticking to the technological framework of the existing one—with all its limitations and possible design flaws. This is aggravated by monolithic systems in which it is increasingly difficult to isolate and replace specific functionalities.

Continuously maintaining software by ad hoc patching is also inferior to replacing larger units. Nevertheless, company internal budgeting policies frequently discourage the production, marketing, and support of reusable components. The usual picture is that managers believe reuse is an important issue, but think it will happen by itself, and thus do little to encourage it, like rewarding the reuse of components, the insertion of components originating from the project into the software library, or discouraging the redevelopment of components. The role of management is frequently reactive, trying to meet deadlines and minimizing cost and time over-runs, rather than visionary, forecasting opportunities and threats.

Moreover, insufficient time is dedicated to the earlier phases of the development process such as analysis and design. During these phases the possibility for identifying the reusability of existing components and defining new reusable components is at its greatest. This chance all too often flows by unnoticed. Software development is not considered as an investment over the whole life span of the product line, but rather as a one-shot development ending when the first release is out of the door and further duty is with the maintenance group.

The widespread misconception is that since it is easy to fix the software, we can start to implement now, and adjust the system later. This is in strong contrast to the traditional manufacturing industries, where a mistake in design is very costly to correct in later development phases. In general, the quality of a system is rarely increased by bug fixes, and often the final product does not correspond to the initial specification.

However, some sort of reuse is performed in most organizations. This reuse originates from individuals who reuse their own components developed during earlier projects or pick up solutions from other developers. Additionally, especially with the widespread usage of object-oriented programming languages, at least the utilization of standard libraries becomes habitual.

From all this it should be obvious that software reuse has a wide spectrum of targets—the question of software library organization which will be further detailed in this work is just one aspect. Reuse is not an end in itself, but it definitely is a means of achieving the general objectives of the company. Companies today are faced with new and more challenging market pressures. In response, companies have to reduce the time-to-market with new or enhanced products, increase the diversity of products available to the customers, and enhance the standardization and interoperability of the products. Reuse is a promising way to achieve such objectives.

In this work we cannot, however, cover the scope of software reuse in sufficient detail, we may just refer the interested reader to the extensive literature on this topic. In particular we left out important issues

such as new organizational structures to assist software reuse [6,7,8], new software process models incorporating software reuse explicitly [9,10,11], the analysis of reuse costs and reusability metrics [12,13,14,15,16,17,18,19,20], the generative approach to software reuse, i.e. application generators, [21,22,23], or the wide area of object-oriented system development with all its positive implications for software reuse [24,25,26]. Good starting points for a more thorough treatment of software reuse are [27,28,29,30,31,32].

3 Organization of Software Libraries

Software libraries are repositories where software components are stored and searched. Hence, they represent a highly valuable resource for the software engineer. Several usage patterns for software libraries can be observed. An engineer can study library components to become familiar with a programming language or more generally with a particular programming style, look for common patterns of usage of a particular component, get acquainted with the requirements of a particular application domain, or—last but not least—reuse library components instead of rewriting them from scratch.

A variety of different approaches to software library organization assisting the software engineer in locating appropriate components have been suggested during the last years. These approaches may roughly be classified into three large groups with are outlined below.

The first group of approaches relies on knowledge-based systems for software library organization. Here the similarity between software components is explicitly encoded by means of a knowledge-representation formalism as, for instance, semantic networks. These approaches share in common that the knowledge-base has to be designed manually. Moreover, the knowledge-base has to be developed anew for each new application domain. Even worse, the knowledge-base has to be adapted manually each time a new component is inserted into the library—a situation that is to be expected to occur quite often especially when software reuse has its well established position within the development process. This laborious task has its benefits in increased retrieval efficiency, yet at the expense of losing generality. We do not want to go into more detail concerning knowledge-based approaches, we just refer to [33,34,35] for prominent proponents of this direction of research.

Secondly, a number of approaches are based on a formal specification describing the functional properties of software components. The abstract nature of formal specifications allows to focus on that aspects of a component which are presumably the most important ones for later retrieval. Following such an approach, better retrieval results in terms of finding appropriate components can be anticipated. The price for these retrieval results, however, is expensive in that the formal specification

has to be developed manually. Recent achievings in this direction of research are presented in [36,37,38].

Finally, ideas from information retrieval and library science have been utilized relying on the textual description of the software components as found in the software reference manual. These approaches count for their high degree of automatization and general applicability. In the remainder of this Section we will give a more detailed account on information retrieval approaches to software library organization.

As usual in information retrieval approaches the various documents, i.e. manual pages describing the software components, have to be mapped onto a representation language. This process is termed *indexing* in the information retrieval literature. The representation language might consist of predefined terms or keywords, i.e. indexing with a controlled vocabulary, or might consist of terms that are extracted from the document at hand, i.e. full-text indexing. In the remainder of this work we will use the words *term* and *keyword* interchangeably to refer to entities that are selected to represent a document.

Controlled vocabularies are beneficial in the sense that the set of possible keywords is well-defined and the user of the library has less degrees of freedom in describing her or his information needs. In particular, each component and each search request is described by using a set of relevant keywords taken from a predefined catalogue. The obvious drawback of a controlled vocabulary is that the assignment of a particular keyword is done manually and is thus rather expensive because highly skilled personnel is required. Furthermore, the process of keyword selection is highly subjective by nature since different people tend to assign different keywords to describe the same object—a phenomenon known as the *vocabulary problem* in information science research [39].

The utilization of controlled vocabularies has found some attraction in the form of *faceted classification* [40,41,42] in the software reuse community. Such a classification schema consists of a set of categories, i.e. the so-called facets, each of which has several predefined keywords that may possibly be filled in. Thus, the classification schema remains flexible with respect to extensibility because components derived from new applications may easily be classified by using the same categories, just the keywords might need an adjustment, especially when components from a new application domain are added to the software library. The selection of appropriate categories, however, is done in a manual process and represents thus a rather expensive task. In [43] a hypertext interface to a library with objects described by means of facted classification is presented. While such an interface certainly is useful for the software engineer, the general problem of manually derived component descriptions is not addressed.

Full-text indexing on the other hand is done automatically and is

hence less subjective. Commonly, the document representation is restricted to those words that appear with sufficient frequency within the document and within the whole document collection. In other words, terms that are expected to provide the best discrimination between documents are used in the representation language [44]. As the major shortcoming we have to note that a particular keyword is used to represent the document only if it is found in that document. Consequently, the retrieval process is susceptible to omissions of relevant documents. A number of papers have been published on the utilization of uncontrolled vocabularies to represent software components. As one example we refer to [45] describing an environment where the full-text of the software documentation is accessed in order to extract the keywords for software representation. In [46] the usage of *lexical affinities* [47], i.e. pairs of frequently co-occurring keywords, is suggested in order to improve the retrieval result. These approaches using full-text indexing are typically combined with statistical cluster analysis to find groups of similar software components. Such a group is then presented together as a retrieval result. In [48,49] the usage of self-organizing maps instead of conventional cluster analysis is suggested because of improved retrieval results. The utilization of fuzzy clustering techniques for software library organization is described in [50].

In Figure 1 we provide a simple pictorial representation of the software retrieval process. On the left hand side, we show the description of the needed software component starting with an information requirements statement that is formalized into a query expression. Essentially, the formalization comprises the transformation of the requirements statement into search terms for the component library. On the right hand side, we show the various reusable software components. The starting point is the component as such that has to be formalized in order to be comparable to the information requests. The comparison between the query and the component representation is termed *matching process* in the figure. Depending on the chosen retrieval model a set of components will be selected as the retrieval result and shown to the user of the library. An overview of different retrieval models may be found in [51,52,53,54]. In our work we use the *vector-space* model where queries and components are represented by means of vectors of keyword occurrences [55]. Depending on the similarity between query vector and component vectors several software components are retrieved as the query result. In particular we rely on the Euclidean vector norm for similarity analysis. Other comparison methods, like the Cosine of the angle between the vectors, should work comparatively well.

A prototypical component-based reuse process is shown in Figure 2. This process starts with an informal description of the needed subsystem that is further transformed into a set of search terms. This search

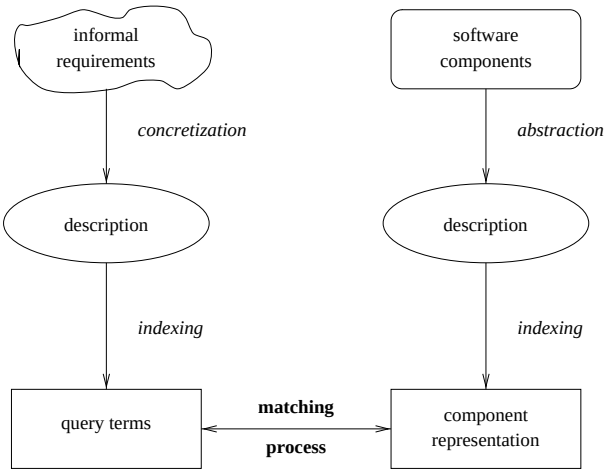

Figure 1: Process model of software retrieval

terms are used to query the component library. Assume, that the retrieval result of this initial query returns no reusable components—not a particularly satisfying yet not an uncommon initial result. The reason for such a rather dissatisfying result might be due to an inappropriate selection of search terms or due to the non-existence of appropriate components in the library. In either case, the retrieval process may be continued with a more detailed subsystem design and thus with a more specific description of the needed components. In our picture we assume that the original subsystem is further decomposed into a number of smaller subsystems for which reusable components are requested from the library. This second retrieval result contains a number of possibly relevant components which are referred to as *candidate components* in the figure. One of those may be reused as such, i.e. the component can be used without modification. Two other components may be reused as well. In the first case, the component has to be adapted in order to be useful. In the second case, it is sufficient to adapt the specification of the component in order to be reusable. In other words, in this case a modification of the subsystem's design enables the reuse of an existing component. In our example one component of the subsystem has to be developed from scratch. This new component and the two adapted components are now candidates for inclusion in the component library for reuse in further projects. Such an inclusion, however, has to be accompanied with careful design and analysis of the components. With such a

reuse process model development *with* reuse and development *for* reuse are tied together in a nice and natural fashion.

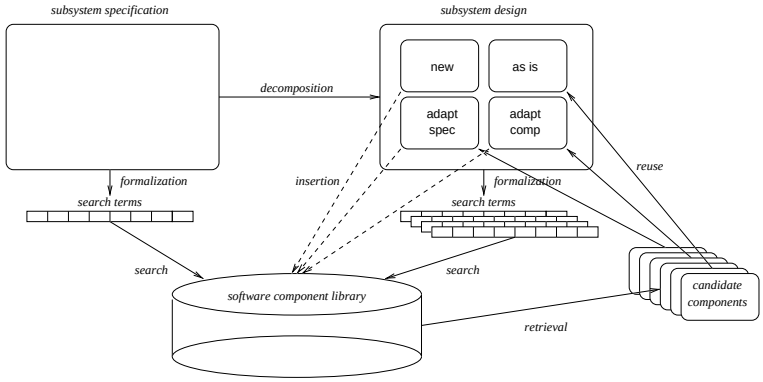

Figure 2: Component-based software reuse

4 Self-Organizing Maps

The *self-organizing map* as proposed in[56,57] and described thoroughly in [58] is one of the most distinguished unsupervised artificial neural network models. This model consists of a layer of input units each of which is fully connected to a grid of output units. These output units are arranged in some topology where the most common choice is represented by a two-dimensional grid.

Input units receive the input patterns x, $x \in \Re^n$, and propagate them as they are onto the output units. Each of the output units i is assigned a weight vector m_i. These weight vectors have the same dimension as the input data, $m_i \in \Re^n$. In the initial setup of the model prior to training, the weight vectors are filled with random values.

During each learning step, the unit c with the highest activity level with respect to a randomly selected input pattern x is adapted in a way that it will exhibit an even higher activity level at future presentations of that specific input pattern. Commonly, the activity level of a unit is computed as the Euclidean distance between the input pattern and that unit's weight vector. Hence, the selection of the winner c may be written as given in Expression (1).

$$c : ||x - m_c|| = \min_i \{||x - m_i||\} \tag{1}$$

Adaptation takes place at each learning iteration and is performed as a gradual reduction of the difference between the respective components of the input vector and the weight vector. The amount of adaptation is guided by a learning-rate α that is gradually decreasing in the course of time. This decreasing nature of adaptation strength ensures large adaptation steps in the beginning of the learning process where the weight vectors have to be tuned from their random initialization towards the actual requirements of the input space. Furthermore, the ever smaller adaptation steps towards the end of the learning process enable a fine-tuned input space representation.

As an extension to standard competitive learning, units in a time-varying and gradually decreasing neighborhood around the winner are adapted, too. Pragmatically speaking, during the learning steps of the self-organizing map a set of units around the winner is tuned towards the currently presented input pattern enabling a spatial arrangement of the input patterns such that alike inputs are mapped onto regions close to each other in the grid of output units. Thus, the training process of the self-organizing map results in a topological ordering of the input patterns. According to [59] we may thus refer to the self-organizing map as a neural network model performing a spatially smooth version of k-means clustering.

The neighborhood of units around the winner may be described implicitly by means of a neighborhood-kernel h_{ci} taking into account the distance—in terms of the output space—between unit i under consideration and unit c, the winner of the current learning iteration. This neighborhood-kernel assigns scalars in the range of [0, 1] that are used to determine the amount of adaptation ensuring that nearby units are adapted more strongly than units further away from the winner. A Gaussian may be used to define the neighborhood-kernel as given in Expression (2) where $||r_c - r_i||$ denotes the distance between units c and i within the output space with r_i representing the two-dimensional vector pointing to the location of unit i within the grid.

$$h_{ci}(t) = e^{-\frac{||r_c - r_i||}{2 \cdot \delta(t)^2}} \tag{2}$$

It is common practice that in the beginning of the learning process the neighborhood-kernel is selected large enough to cover a wide area of the output space. The spatial width of the neighborhood-kernel is reduced gradually during the learning process such that towards the end of the learning process just the winner itself is adapted. Such a reduction is done by means of the time-varying parameter δ in Expression (2). This strategy enables the formation of large clusters in the beginning and fine-grained input discrimination towards the end of the learning process.

In combining these principles of self-organizing map training, we may

write the learning rule as given in Expression (3). Please note that we make use of a discrete time notation with t denoting the current learning iteration. The other parts of this expression are α representing the time-varying learning-rate, h_{ci} representing the time-varying neighborhood-kernel, x representing the currently presented input pattern, and finally m_i denoting the weight vector assigned to unit i.

$$m_i(t+1) = m_i(t) + \alpha(t) \cdot h_{ci}(t) \cdot [x(t) - m_i(t)] \tag{3}$$

A simple graphical representation of a self-organizing map's architecture and its learning process is provided in Figure 3. In this figure the output space consists of a square of 36 units, depicted as circles. One input vector $x(t)$ is randomly chosen and mapped onto the grid of output units. In the second step of the learning process, the winner c is selected. Consider the winner being the unit depicted as the black node in the figure. The weight vector of the winner, $m_c(t)$, is now moved towards the current input vector. This movement is symbolized in the input space in Figure 3. As a consequence of the adaptation, unit c will produce an even higher activation with respect to input pattern x at the next learning iteration, $t+1$, because the unit's weight vector, $m_c(t+1)$, is now nearer to the input pattern x in terms of the input space. Apart from the winner, adaptation is performed with neighboring units, too. Units that are subject to adaptation are depicted as shaded nodes in the figure. The shading of the various nodes corresponds to the amount of adaptation and thus, to the spatial width of the neighborhood-kernel. Generally, units in close vicinity of the winner are adapted more strongly and consequently, they are depicted with a darker shade in the figure.

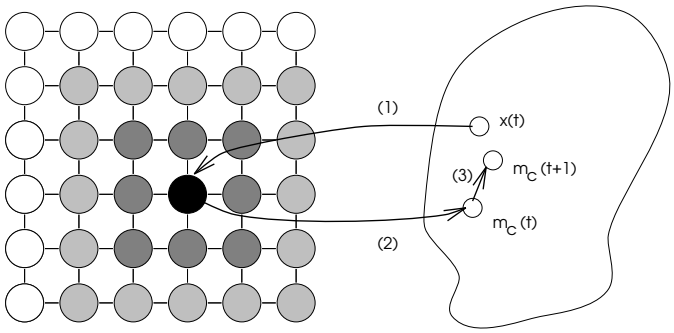

Figure 3: Architecture and training process of a self-organizing map

5 The Experimental Software Library

5.1 The NIH class library

Throughout the remainder of this work we will use the various components of the NIH Class Library [60], the NIHCL in short, as a sample software component library. The NIHCL is a collection of classes developed in the C++ programming language. The class library covers classes for storing and retrieving arbitrarily complex data structures on disk like OIOifd and OIOostream, generally useful data types such as String, Time, and Date, and finally a number of container classes as, for example, Set, Dictionary, and OrderedCltn. A more complete description of the class library may be found in [61].

In an information retrieval approach to software library organization the task of similarity recognition between the various software components is replaced by similarity recognition between natural language documents describing these components, i.e. their respective manual pages. However, since natural language processing is still far from understanding arbitrarily complex document structures, the various documents have, in a first step, to be mapped onto some representation language in order to be comparable. Still one of the most widely used representation languages is single-term full-text indexing [53]. Roughly speaking, the documents are represented by the set of words they are built of. As a result, the various text documents are represented by vectors of equal dimension. Each vector component corresponds to a keyword from the representation language, and each vector entry in a particular component relates to the importance of that very keyword in describing the component at hand. In its most basic form, i.e. binary single-term indexing, an entry of one indicates that this specific keyword was extracted from the description of the component at hand. Contrary to that an entry of zero means that the corresponding keyword is not contained in that component's description [54]. A simple example of binary single-term indexing is provided in Table 1.

	$Keyword_1$	$Keyword_2$	$\cdots$	$Keyword_n$
$Component_1$	1	1		0
$Component_2$	0	1		0
$Component_3$	0	0		1
$\cdots$				
$Component_m$	1	0		1

Table 1: Binary indexing of software components

Assuming a suitable representation, the similarity between two documents corresponds to the distance between their vector representations, i.e. the rows in Table 1. The benefit of such a vector-based representation is the ease of implementing a best-match retrieval strategy, where the retrieved software components may be ranked according to decreasing similarity between the vector representing the actual query, i.e. the description of the needed software component, and the vectors representing the various text documents describing the components stored in the archive.

In order to obtain the final document representation, we accessed the full-text of the various manual pages describing NIHCL classes. As an illustrative example we refer to Figure 4 containing a portion of the textual description of class Set as provided in [60].

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 attempts 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 4: NIHCL manual entry of class Set

The natural language text of the various manual pages describing the classes is accessed and full-text indexed in order to generate a binary vector-space representation of the documents. Just to provide the exact figure, the indexing process identified 489 distinct content terms and thus, each component is represented by a 489-dimensional feature vector. During the indexing process we excluded terms contained in a small stop-word list—mostly articles, pronouns, and conjunctions—and terms that occurred in less than two documents. These vectors are subsequently used as the input data to the self-organizing map.

5.2 Cluster analysis: The baseline for comparison

In order to provide the baseline for the discussion of the results achieved with unsupervised neural networks, we will first show the classification results from statistical cluster analysis [62]. Table 2 contains the result from k-means clustering where the software components are assigned to 4 clusters. The limitation of statistical cluster analysis is quite obvious from that result. There are two clusters with a large number of components. As a consequence, the functional closeness of these components is questionable to say the least. Especially *Class 3* contains a broad variety of rather unrelated components. On the other hand, two clusters consist of a very small number of components. It is obvious that these clusters are too specialized to be useful.

	Software Components
Class 1	Arraychar, ArrayOb, Assoc, AssocInt, Bag, Collection, Dictionary, Heap, IdentDict, IdentSet, Iterator, KeySortCltn, Link, LinkedList, LookupKey, OrderedCltn, SeqCltn, Set, SortedCltn, Stack
Class 2	Vector
Class 3	Bitset, Class, Date, Exception, FDSet, Float, Integer, LinkOb, Nil, Object, OIOifd, OIOin, OIOistream, OIOnihin, OIOnihout, OIOofd, OIOostream, OIOout, Point, Random, ReadFromTbl, StoreOnTbl, Time
Class 4	Range, Regex, String

Table 2: k-means clustering of the NIHCL—4 clusters

A result from k-means clustering where the software components are assigned to 10 clusters is shown in Table 3. Here the separation comes closer to the functionality of the components but the result is still not completely satisfying. *Class 1*, *Class 6*, and *Class 8* contain mostly components implementing data structures. Some of the assignments are, however, not justifyable within their particular cluster like AssocInt, LookupKey, and ArrayOb. *Class 9* contains mostly data types—the exclusion of String and Vector from that cluster, however, is far from natural. A completely satisfying cluster is formed with *Class 2* consisting exclusively of file I/O components.

	Software Components
Class 1	Arraychar, Assoc, Bag, Bitset, Dictionary, Heap, IdentDict, IdentSet, KeySortCltn, OrderedCltn, SeqCltn, Set, SortedCltn, Stack
Class 2	OIOifd, OIOin, OIOistream, OIOnihin, OIOnihout, OIOifd, OIOostream, OIOout, ReadFromTbl, StoreOnTbl
Class 3	Iterator
Class 4	Vector
Class 5	Object
Class 6	AssocInt, Link, LinkedList, LinkOb, LookupKey
Class 7	Class, Exception, Nil, Regex
Class 8	ArrayOb, Collection
Class 9	Date, FDSet, Float, Integer, Point, Random, Range, Time
Class 10	String

Table 3: k-means clustering of the NIHCL—10 clusters

6 Unsupervised Learning in Software Library Organization

6.1 Results with the standard self-organizing map model

A typical result from the application of self-organizing maps to data describing reusable software components is provided in Figure 5. In this case we have used a 10×10 self-organizing map to represent the software library. Generally, the graphical representation may be interpreted as follows. Each output unit of the self-organizing map is represented by means of either a dot or a class name. The class name appears where the respective unit is winner of the input pattern representing that specific software component. Contrary to that, a dot marks units that have not won the competition for an input pattern. These units, however, have their important role in determining the spatial range of the various data clusters.

In the upper right part of the map we recognize the arrangement of all classes performing file I/O, these classes are designated by the 'OIO'-part of their respective class name. The region itself is organized in such a way that a class performing an input operation is mapped neighboring its output counterpart, e.g. OIOifd and OIOofd.

Just below the file I/O region we find a large area of the map consisting of the various classes representing data structures. Within this large region we recognize a smaller area in the lower middle of the map com-

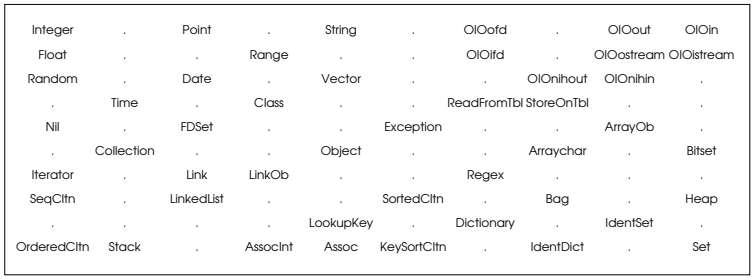

Figure 5: 10×10 *Self-Organizing Map* representation of the NIHCL

prising classes that allow an access to their elements via a key-attribute, i.e. Dictionary, IdentDict, and KeySortCltn. Within this particular area we find the classes that actually implement this type of access, i.e. Assoc, AssocInt, and LookupKey. Please note that this important relationship was never obvious with the results of statistical cluster analysis as presented above.

Finally, we want to shift the attention towards the upper left-hand side of the map where the classes representing data types are located. Again, their arrangement reflects their mutual similarity. As some examples we refer to the assignment of Date and Time, String and Vector, and Float and Integer to geographically close units, respectively. Their class names are fully self-contained, we thus refrain from a detailed discussion of the functionality of these classes. Furthermore, please note the placement of class Random, a random number generator producing (pseudo-) random numbers of Float data type. This class is mapped onto a unit neighboring the basic numerical data types Float and Integer.

A lot more interesting areas may be located in this representation. We cannot, however, cover them in detail here, we rather refer to [63] especially for a comparison of these results with those obtained from statistical cluster analysis.

However, in case one is not familiar with the NIHCL as such it is still problematic to identify regions of functionally similar C++ classes. From the pure two-dimensional representation as provided in Figure 5 a casual user might for instance conclude that the classes Point and String or String and OIOofd are of comparable similarity since they are located at units of equal distance in the upper middle of the final map. The similarity of the former two classes is evident in that both classes represent data types. The latter pair, however, is formed from two completely unrelated classes, the one being a data type, the other a file I/O class. Such

a representation might be misleading for the unexperienced user of the software library, to say the least. With a novel visualization technique, outlined in the next subsection, we are addressing exactly this type of misleading representation.

6.2 Adaptive coordinates for improved similarity representation in self-organizing maps

We extended the basic learning rule in order to capture the movements of the various weight vectors within a two-dimensional 'virtual' output space for subsequent visualization of the clustering result. We will refer to this extension as the *adaptive coordinate* visualization technique[64,65].

The initial setup of the 'virtual' output space is such that each of the output units of the self-organizing map is shown in its position within the two-dimensional grid defined by the neural network architecture. In other words, the initial coordinates $\langle ax_i, ay_i \rangle$ of the unit i are identical to the unit's position within the grid of the map. In the beginning of each learning step the distances between the weight vectors and a randomly selected input pattern are stored in a table, $Dist(t)$, where the distance of a particular unit i is denoted as $Dist_i(t)$. After the adaptation of weight vectors the new distance table is calculated, $Dist(t+1)$. As the next step, the relative change in distance according to the actual learning iteration is computed for every unit i, i.e. Δ_{Dist_i}. This computation is given in Expression (4).

$$\Delta_{Dist_i}(t+1) = \frac{Dist_i(t) - Dist_i(t+1)}{Dist_i(t)} \tag{4}$$

The movement of the various weight vectors within the input space due to the adaptation process is performed analogously within the 'virtual' output space. Pragmatically speaking, the adaptive coordinates $\langle ax_i, ay_i \rangle$ are used in order to mimic the movement of this unit's weight vector during the training process. Since the presented input signal was mapped onto the winning unit c, no adaptation of the position of the winning unit is performed, with c now being the representative of the selected input signal in terms of the 'virtual' output space. This unit's position is further used as an attractor for other units in the neighborhood of the winner. The adaptive coordinates of all units but the winner are now moved by the fraction given in $\Delta_{Dist_i}(t+1)$ towards the position of the winning unit c given by $\langle ax_c, ay_c \rangle$. Note that the larger the width of the neighborhood-kernel the more units' coordinates are adapted—this is similar to the adaptation process of weight vectors in the basic learning rule.

In Expression (5) we provide the exact formulation of adaptation performed with the ax-coordinate of unit i. The calculation of the new

ay-coordinate of unit i, ay_i, is performed analogously.

$$ax_i(t+1) = ax_i(t) + \Delta_{Dist_i}(t+1) \cdot (ax_c(t) - ax_i(t)) \qquad (5)$$

In Figure 6 we present a highly idealized schematic representation of the effect of the adaptive coordinate visualization technique. The upper part of this figure is already known from Figure 3. The shaded nodes represent units that are subject to weight vector adaptation and are thus also subject to coordinate adaptation. The adaptive coordinates of these units are moved closer to the winner as shown in the lower part of Figure 6. Units that are not adapted due to the limited range of the neighborhood-kernel do not change their location within the 'virtual' output space.

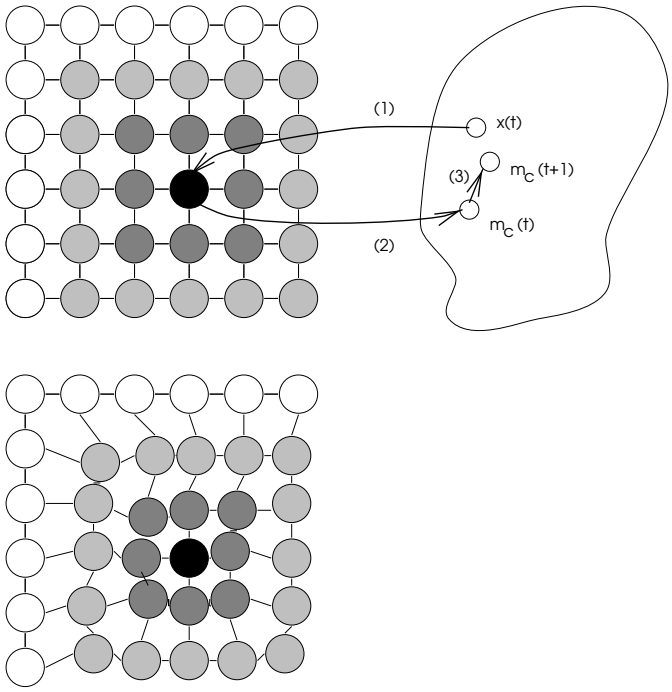

Figure 6: Adaptive coordinate visualization technique

Thus, the clustering of units around the winning unit resembles the clustering of the units' weight vectors around the presented input signal after the current training cycle. After convergence of the training process

the clusters learned by the self-organizing map can be visualized by using the coordinates $\langle ax_i, ay_i \rangle$ to plot the position of unit i in the 'virtual' output space.

We have to note that we usually do not start with coordinate adaptation right from the beginning of the learning process. We rather wait until the first stage of rough clustering is finished when the neighborhood-kernel has a width of, say, half the initial one. The reason for the later beginning of coordinate adaptation is simply the fact that during the initial phases of the learning process the weight vector adaptation results in a rough ordering of the input patterns and the necessary evolution of the random initial weight vector positions towards the actual requirements of the input data set. Starting that early would mean that the rather large movements of weight vectors are repeated with the adaptive coordinates. Moreover, due to the large adaptation movements affecting a wide area of the network, chances are high that the adaptive coordinates of each unit converge towards—almost—the same position in the 'virtual' output space. Starting with coordinate adaptation in a later stage of the learning process ensures that these unwanted effects are avoided.

By using the adaptive coordinate visualization technique the training result of the self-organizing map is displayed as shown in Figure 7.

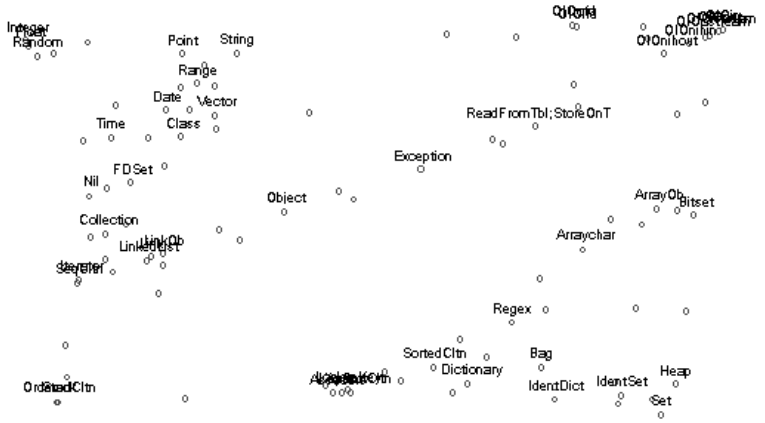

Figure 7: *Adaptive coordinate* representation of a 10×10 *Self-Organizing Map*

The most obvious observation from the result as depicted in Figure 7 is that the regions we described for Figure 5 are readily recognizable thanks to the fact that they are better separated graphically. Consider for example the pairs Point and String or String and OlOofd. These pairs

have been identified as a sample area of misleading visualization in the conventional output space representation. With the adaptive coordinate visualization technique they are now separated substantially in the output space. There is certainly no longer the risk that these classes might be considered as being comparably similar.

Moreover, some parts of the final map contain such highly similar classes that the learning process allocates a fairly condensed region in the map for them. As an example consider the file I/O classes in the right upper part of Figure 7. For convenient comparison we show an enlarged visualization of this area in Figure 8. A similar observation holds true for the left upper part of the map containing the various classes implementing data types, and here especially for the area containing the classes Integer, Float, and Random. This region is enlarged in Figure 9. The fact that these software components belong to a fairly homogeneous group is even stressed by using the adaptive coordinate technique for output space visualization.

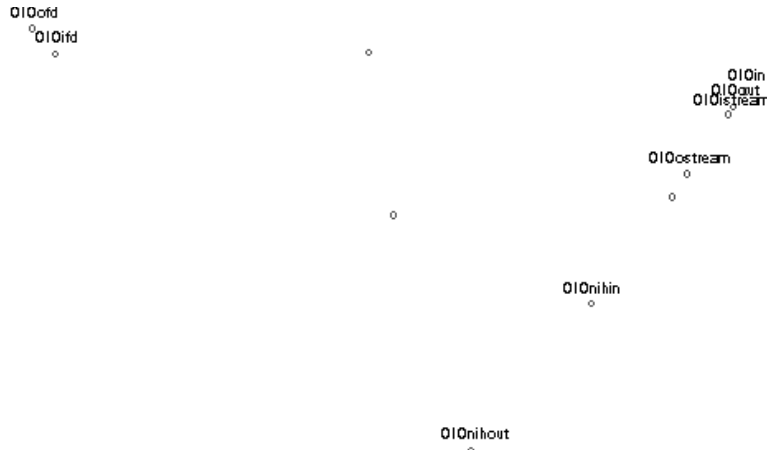

Figure 8: Right upper region of Figure 7

6.3 Results with the growing grid model

When using the self-organizing map as the underlying model for software library organization we have to define the dimension of the output space in terms of the number of units in advance. This might be somewhat difficult when the contents of the library is not known exactly. A model that overcomes this limitation by relying on an adaptive architecture

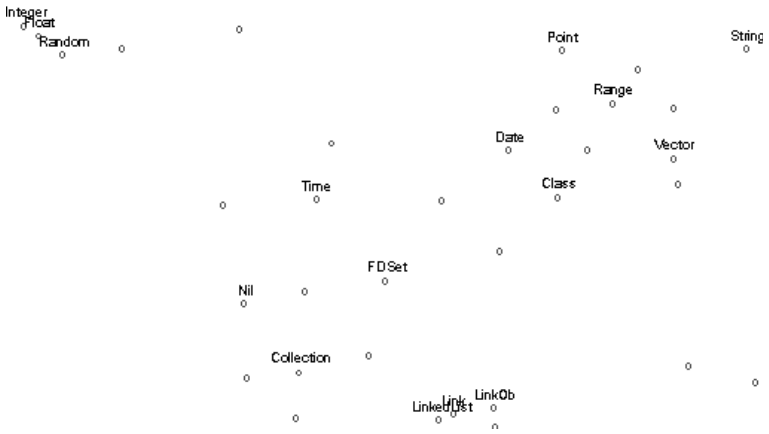

Figure 9: Left upper region of Figure 7

that grows according to the specific needs of the input space is the only recently suggested *growing grid* [66]. The training process starts with a square of just four units and the network grows in the course of the training process in order to improve the representation of the input space. The growth process retains the rectangular structure of the output space.

During each learning iteration the winner as well as its neighboring units are adapted in order to resemble the current input pattern more closely. This adaptation strategy is thus similar to self-organizing maps. The first remarkable difference in the training processes is that the growing grid model is trained with a fixed value of the learning-rate and constant neighborhood-kernel as opposed to the time-varying behavior of their counterparts in self-organizing maps. The second difference is marked by in the so-called *signal counter*, where for each unit the number of times this very unit served as the winner is recorded. This figure represents an indication for the location in the output space where the input space is not represented adequately. The underlying assumption is that if a unit serves often as the winner it represents a number of—presumably—different input patterns. Ever after a fixed number of training iterations the unit with the highest signal counter is selected and either a new row or a new column is inserted depending of the direction of that unit's most dissimilar neighbor. The dissimilarity of two units, obviously, is calculated as the distance between their respective weight vectors.

The growth process is symbolized in Figure 10. On the left hand side

of the figure we find the architecture of the network before insertion. On the right hand side, the architecture after insertion of a column of units in between the unit with the highest signal counter, depicted as a black node, and its most dissimilar neighbor, depicted as a hatched node, is shown.

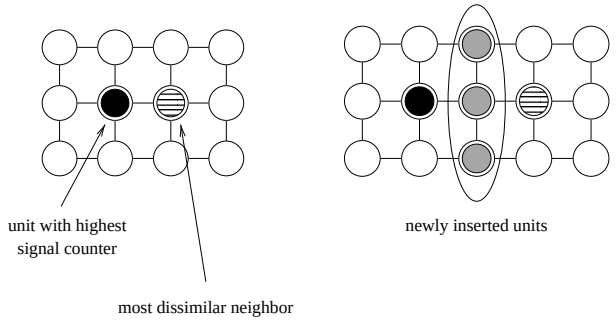

Figure 10: Growth process during *Growing grid* training

Subsequently, the weight vectors of the new units are initialized and the signal counter values are redistributed between already existing and newly inserted units. The growth process terminates when a sufficiently exact input pattern representation is reached in terms of the mean quantization error, i.e the mean of the remaining deviation of input vector and best matching weight vector after training. After the termination of the growth process the training continues with weight vector finetuning according to the standard self-organizing map training rule.

As a consequence, the growing grid model is able to exhibit the characteristic benefit of self-organizing maps, namely the visualization of input data similarity in terms of geographical closeness within the grid of output units. As a specific benefit of the growing grid model we have to stress the fact that the model determines an appropriate number of units within the framework of an unsupervised training process.

In Figure 11 a typical result of the growing grid model trained with the NIHCL data is depicted. This figure shows the input pattern representation after termination of the growth process with a 6×14 grid and before finetuning. It is quite obvious that the overall input space representation resembles that from self-organizing maps as shown above. The major difference is that some units represent a rather large number of input patterns, e.g. the file I/O classes in the lower right corner.

The benefit of the finetuning process is the better separation of input patterns within the grid of units as can be seen in Figure 12.

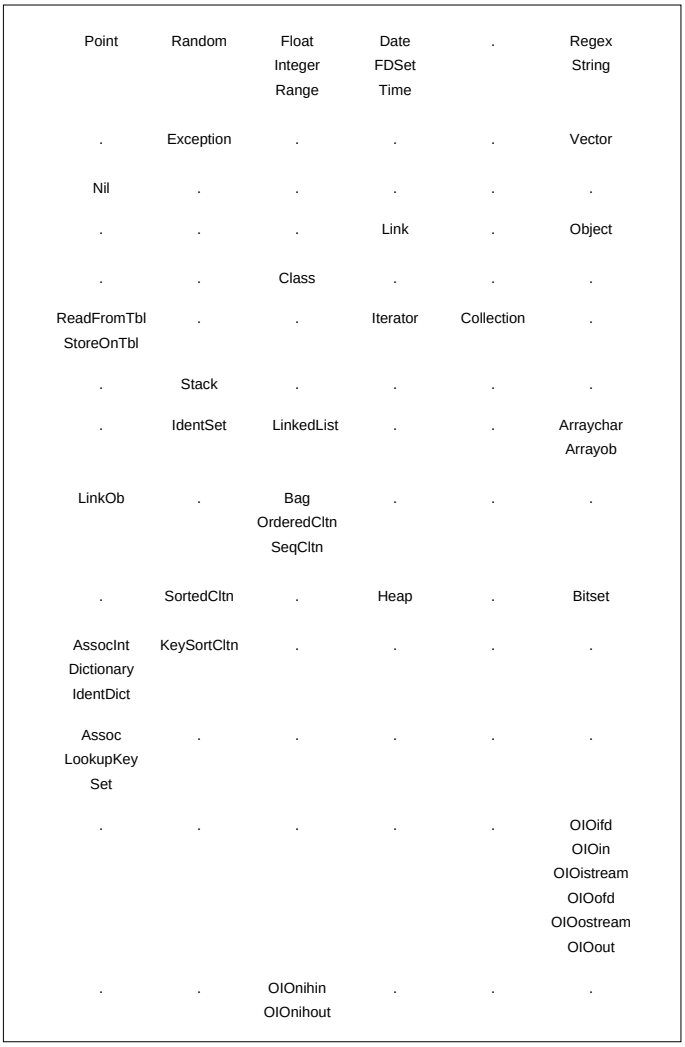

Figure 11: 6×14 *Growing Grid* representation of the NIHCL before finetuning

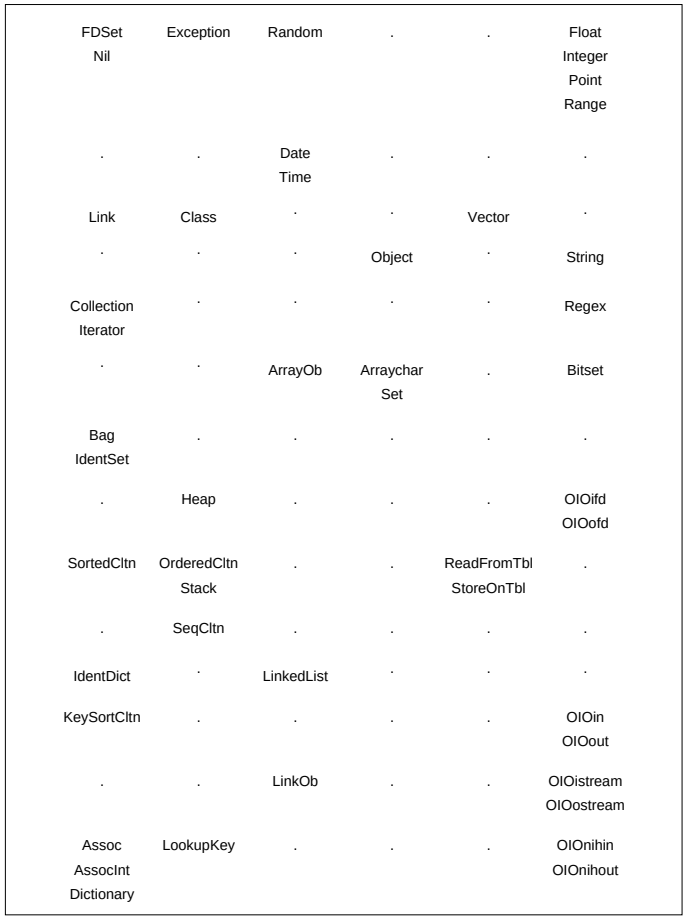

Figure 12: 6×14 *Growing Grid* representation of the NIHCL after finetuning

The observation of the training process of the growing grid model is highly valuable for exploratory data analysis of the software library. In particular it is noteworthy that the overall discrimination into classes implementing data types, container classes, and classes handling file I/O is easily observable in that these classes are mapped onto the same—or a small number of—units respectively during the early stages of the training process when the network is small. In this sense it is rather straight forward to determine functional correspondences even in unknown libraries.

7 Conclusions

In this work we described the utilization of unsupervised neural networks, in particular self-organizing maps, for software library organization. The reason for using this particular type of neural network may be found in its ability to reveal the similarity of software components in a highly intuitive fashion in terms of spatial distance within a two-dimensional plane. We have shown, that the self-organizing map is capable of arranging a number of reusable software components according to their mutual functional closeness where similar components are shown in geographically near regions of the two-dimensional output space of the self-organizing map. The results of a novel visualization technique for self-organizing maps provides a highly accessible representation of similarity where clusters containing similar components are tied together more closely within the two-dimensional plane. Such a library representation has its benefits when developers look for a particular component during the development process of a software system. Additionally, the self-organizing map can be used easily for interactive exploration of the software library—a feature that is of vital importance in software reuse. For exploratory analysis of the contents of the software library we suggested the utilization of the growing grid model which represents, pragmatically speaking, a self-organizing map with adaptive architecture that develops according to the requirements of the input data space.

Acknowledgments

Thanks are due to Andreas Rauber for long and fruitful discussions on cluster visualization for self-organizing maps and the implementation of the adaptive coordinate visualization technique. Additional thanks go to Wolfgang Liedl for providing a prototype implementation of the Growing Grid model.

References

1. F. P. Brooks. No silver bullet—Essence and accidents of software engineering. *IEEE Computer*, 20(4), 1987.
2. B. Boehm. Improving software productivity. *IEEE Software*, 4(6), 1987.
3. B. J. Cox. Planning the software revolution. *IEEE Software*, 7(6), 1990.
4. E. Horowitz and J. B. Munson. An expansive view of reusable software. *IEEE Transactions on Software Engineering*, 10(5), 1984.
5. H. van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, Chichester, UK, 1993.
6. V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora. The software engineering laboratory—An operational software experience. In *Proceedings of the Int'l Conference on Software Engineering (ICSE'92)*, Melbourne, Australia, 1992.
7. M. A. Cusumano. The software factory: A historical interpretation. *IEEE Software*, 6(2), 1989.
8. Y. Matsumoto. Experiences from software reuse in industial process control applications. In *Proceedings of the Int'l Workshop on Software Reusability*, Lucca, Italy, 1993.
9. B. Henderson-Sellers and J. M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9), 1990.
10. R. Prieto-Diaz. Integrating domain analysis and reuse in the software development process. In *Proceedings of the Annual Workshop on Methods and Tools for Reuse*, Syracuse, NY, 1990.
11. M. A. Simos. The domain-oriented life cycle: Toward an extended process model for reusability. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Computer Socienty Press, Piscataway, NJ, 1990.
12. B. H. Barnes and T. B. Bollinger. Making reuse cost-effective. *IEEE Software*, 8(1), 1991.
13. S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.
14. N. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, UK, 1991.
15. W. B. Frakes and C. Terry. Reuse level metrics. In *Proceedings of the Int'l Conference on Software Reuse (ICSR'94)*, Rio de Janeiro, Brazil, 1994.
16. W. B. Frakes and C. Terry. Software reuse and reusability metrics models. Technical Report TR-95-07, Virginia Polytechnic Institute and State University, Falls Church, VA, 1995.
17. M. Hitz. Measuring reuse attributes in object-oriented systems. In *Proceedings of the Int'l Conference on Object-Oriented Information*

Systems, Dublin, Ireland, 1995.
18. M. Hitz and B. Montazeri. Chidamber & Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4), 1996.
19. J. Margono and T. E. Rhoads. Software reuse economics: Cost benefit analysis on a large-scale ADA project. In *Proceedings of the Int'l Conference on Software Engineering (ICSE'92)*, Melbourne, Australia, 1992.
20. J. S. Poulin, J. M. Caruso, and D. R. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4), 1993.
21. C. T. Cleaveland. Building application generators. *IEEE Software*, 5(4), 1988.
22. L. S. Levy. A metaprogramming method and its economic justification. *IEEE Transactions on Software Engineering*, 12(2), 1986.
23. S. K. Misra and P. J. Jalics. Third-generation versus fourth-generation development. *IEEE Software*, 5(4), 1988.
24. I. Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.
25. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
26. R. Wirfs-Brock and R. E. Johnson. Surveying current research on object-oriented design. *Communications of the ACM*, 33(9), 1990.
27. T. J. Biggerstaff and A. J. Perlis, editors. *Software Reusability*. Addison-Wesley, Reading, MA, 1989.
28. W. B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Software*, 11(5), 1994.
29. E.-A. Karlsson, editor. *Software Reuse: A Holistic Approach*. John Wiley & Sons, Chichester, UK, 1995.
30. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), 1992.
31. H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6), 1995.
32. W. Schäfer, R. Prieto-Diaz, and M. Matsumoto, editors. *Software Reusability*. Ellis Horwood, New York, 1994.
33. P. Devanbu, R. J. Brachman, P. J. Selfridge, and B. W. Ballard. LaSSIE—A knowledge-based software information system. *Communications of the ACM*, 34(5), 1991.
34. E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library. *ACM Transactions on Software Engineering and Methodology*, 1(3), 1992.
35. G. Spanoudakis and P. Constantopoulos. Similarity for analogical software reuse—A conceptual modeling approach. In *Proceedings*

of the Int'l Conference on Advanced Information Systems Engineering (CAiSE'93), 1993.
36. B. H. C. Cheng and J.-J. Jeng. Reusing analogous components. *IEEE Transactions on Knowledge and Data Engineering*, 9(2), 1997.
37. R. Mili, A. Mili, and R. T. Mittermeir. Storing and retrieving software components: A refinement based approach. *IEEE Transactions on Software Engineering*, 23(7), 1997.
38. A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2), 1995.
39. G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communications. *Communications of the ACM*, 30(11), 1987.
40. R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1), 1987.
41. R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5), 1991.
42. E.-A. Karlsson, S. Sørumgård, and E. Tryggeseth. Classification of object-oriented components for reuse. In *Proceedings of the Int'l Conference on Technology of Object-Oriented Systems (TOOLS'92)*, 1992.
43. T. Isakowitz and R. J. Kauffman. Supporting search for reusable software objects. *IEEE Transactions on Software Engineering*, 22(6), 1996.
44. G. Salton and C. Buckley. Term weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5), 1988.
45. Y. F. Chang and C. M. Eastman. An information retrieval system for reusable software. *Information Processing & Management*, 29(5), 1993.
46. 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.
47. Y. S. Maarek and F. A. Smadja. Full-text indexing based on lexical relations: An application—Software libraries. In *Proceedings of the Int'l ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'89)*, Cambridge, MA, 1989.
48. D. Merkl. Structuring software for reuse—The case of self-organizing maps. In *Proceedings of the Int'l Joint Conference on Neural Networks (IJCNN'93)*, Nagoya, Japan, 1993.
49. D. Merkl, A M. Tjoa, and G. Kappel. Learning the semantic similarity of reusable software components. In *Proceedings of the Int'l Conference on Software Reuse (ICSR'94)*, Rio de Janeiro, Brazil, 1994.

50. W. Pedrycz and J. Waletzky. Fuzzy clustering in software reusability. *Software–Practice and Experience*, 27(3), 1997.
51. W. B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.
52. G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
53. G. Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA, 1989.
54. H. R. Turtle and W. B. Croft. A comparison of text retrieval models. *Computer Journal*, 35(3), 1992.
55. D. L. Lee, H. Chuang, and K. Seamons. Document ranking and the vector-space model. *IEEE Software*, 14(2), 1997.
56. T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43, 1982.
57. T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9), 1990.
58. T. Kohonen. *Self-organizing maps*. Springer-Verlag, Berlin, 1995.
59. B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, UK, 1996.
60. K. E. Gorlen. *NIH class library reference manual*. National Institutes of Health, Bethesda, MD, 1990.
61. K. E. Gorlen, S. Orlow, and P. Plexico. *Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, New York, 1990.
62. A. K. Jain and R. D. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, NJ, 1988.
63. D. Merkl. A connectionist view on document classification. In *Proceedings of the Australasian Database Conference (ADC'95)*, Adelaide, Australia, 1995.
64. D. Merkl and A. Rauber. On the similarity of eagles, hawks, and cows—Visualization of similarity in self-organizing maps. In *Proceedings of the Int'l Workshop on Fuzzy-Neuro-Systems*, Soest, Germany, 1997.
65. D. Merkl and A. Rauber. Alternative ways for cluster visualization in self-organizing maps. In *Proceedings of the Workshop on Self-Organizing Maps*, Espoo, Finland, 1997.
66. B. Fritzke. Growing Grid: A self-organizing network with constant neighborhood range and adaptation strength. *Neural Processing Letters*, 2(5), 1995.

Exercises

1. Which artifacts of the software development process are usually considered to be reusable?
2. Why is the software retrieval process usually considered to be an iterative one where the early retrieval results normally are not the most satisfying ones?
3. Discuss the differences between the vector-space model of information retrieval and the Boolean retrieval model.
4. What are the benefits of using the self-organizing map as compared to more conventional clustering methods?
5. What are the limitations of the standard self-organizing map visual representation of a software library?
6. Discuss the feasibility of using the adaptive coordinate visualization technique together with the growing grid artificial neural network model.
7. Discuss the benefits that growing neural networks like the growing grid model have to offer for software library organization.