An Application of Machine Learning to the Organization of Institutional Software Repositories

Sidney Bailin and Scott Henderson
CTA Incorporated
6116 Executive Boulevard, suite 800
Rockville Maryland, 20852
(301) 816-1218

Walt Truszkowski
NASA/Goddard Space Flight Center
Data Systems Technology Division
Greenbelt, Maryland 20771
(301) 286-8821

Abstract

Software reuse has become a major goal in the development of space systems, as a recent NASA-wide workshop on the subject made clear.[1] The Data Systems Technology Division of Goddard Space Flight Center has been working on tools and techniques for promoting reuse, in particular in the development of satellite ground support software. One of these tools is the Experiment in Libraries via Incremental Schemata and Cobweb (ElvisC). ElvisC applies machine learning to the problem of organizing a reusable software component library for efficient and reliable retrieval. In this paper we describe the background factors that have motivated this work, present the design of the system, and evaluate the results of its application.

1. Repositories vs. Architectures in Reuse

The work described in this paper concerns the self-organizing capability of a repository of reusable software components. The recent trend in software reuse support technology has been away from a dependence on unguided searching of component repositories, and towards a more model-based approach [D'Ippolito '89; Simos '91; Mark '92]. In this introduction we explain the difference between the two approaches, the role of the model-based approach at NASA/Goddard, and our reasons for focusing on repository organization.

The model-based approach to reuse emphasizes the development of generic architectures within a given application domain. A generic architecture shows how systems in the domain are typically composed of reusable components. The field is moving towards knowledge-based support for generic architectures, e.g., formalizing the constraints under which components can be combined and the commitments entailed by reusing specific components. The ultimate goal is to build new systems by selecting features or specifying parameter values, thus instantiating the generic architecture with little or no additional software development.

NASA/Goddard has a long and successful history of using generic architectures, since even before the terminology became fashionable. The Standard Software System (SSS) [WEC '79]was developed in the late 1970s to provide a kernel of support for satellite operations control centers, and several missions were based upon it. The Multi-Satellite Operations Control Center Application Executive (MAE) [Sperry '85] was a successor to SSS in the mid-1980s, motivated in large part by the replacement of processor hardware and operating systems. Again, several missions were based upon MAE. Most recently, the Transportable Payload Operations Control Center (TPOCC) [Measday '91] was developed to take advantage of workstation technology and open system interconnection. The degree of parameterization and configurability has increased with each of these successive software architectures.

In a narrower domain—that of monitoring telemetry housekeeping data—the Generic Spacecraft Analyst Advisor (GenSAA) [Hughes '91] is generalizing an architecture for rule-based monitoring systems that was first illustrated in the Communications Link Expert Advisory Resource (CLEAR). GenSAA will provide a high degree of automated support in the creation of monitoring systems for different spacecraft.

---

[1] Organized by Langley Research Center and held at the Research Triangle Institute in April, 1992.

The Flight Dynamics Division of Goddard is developing the Combined Operational Mission Planning and Attitude Support System (COMPASS), which is an Ada-generic framework for creating orbit analysis systems for different spacecraft. The role for such generic architectures in the development of satellite ground software seems to be firmly established.

The need for specialized software has not disappeared, however. In order to meet the ever expanding requirements of the scientific community with available resources, new technologies must continually be inserted into ground support systems. Direct-manipulation user interfaces and knowledge-based systems are typical of the kinds of technology being introduced into control centers.

The Data Systems Technology Division has as its charter the development of new technologies to the point where they can be safely integrated into a control center. In such an environment there is a greater need to respond to unprecedented requirements, and less of a role for generic application architectures. The goal of reuse-based software development remains, however. Thus the emphasis in this group is on reuse within *horizontal domains*: these are the supporting domains that provide basic application-independent services, upon which vertical (i.e., application) domains are built (for example, databases, communications, and user interfaces are horizontal domains).

The Data Systems Technology Division's Code 522 develops a substantial portion of their software using the programming language C++. Each software project spins off a set of potentially reusable modules in the form of C++ component classes. These components implement, for the most part, application-independent functions. Because the components are produced as they are needed, their capabilities tend to be scattered over several horizontal domains. In order for developers to take full advantage of this collection—including developers who may not have contributed components—some rational method of organizing the library is required. A rational organization is one that allows developers to find what they need efficiently and reliably.

A key consideration in this challenge is that a given organization will not suffice as the contents of the repository evolve. As requirements evolve so do solution techniques. The contents of the library will change as new reusable components are developed—also as older components are removed, because their usefulness has diminished or was overestimated to begin with. As the contents of the library change, we cannot expect the same organization to remain appropriate.

Why not use the "given" hierarchy of C++ classes, which is implied by their public inheritance structure, to organize the repository? Public inheritance in C++ is supposed to represent the "is-a" relation [Meyers '91, Cargill '91]. In principle, a global inheritance hierarchy could function as the organization of the repository, and new components could be created within this inheritance framework. Our conclusion, however, is that this approach is not pragmatic. Design teams elaborate local inheritance hierarchies within a project or segment of a project to achieve maintainable code. A typical project may include a dozen independent inheritance hierarchies. The additional levels of abstraction required to relate these hierarchies to each other and to the organization's past work are typically missing because it would be expensive to develop and no additional functionality would be gained by this extra effort. When multiple individuals and multiple organizations are involved in a project, irreconcilable hierarchies may be developed to support similar functions. Adjusting source code to retrofit it into a unified inheritance hierarchy may not be economically feasible or even possible (in the case of object-code libraries). Finally, we want the organization of the repository to be malleable—to evolve as the characteristics of the components evolve. We would like to achieve such flexibility without having to revamp the C++ class definitions continually.

Thus we have been led to consider a method for incrementally defining categories of components. In the machine learning literature, such a method is characterized as an *unsupervised incremental* learning algorithm: unsupervised, because the set of available categories is not given to the algorithm *a priori*, but is developed instead by the algorithm itself; incremental, because the set of categories, as well as their boundaries, can change every time a component is added or removed from the repository [Fisher and Pazzani 91]. Unsupervised incremental learning is sometimes known as *concept formation*. ElvisC employs a concept formation algorithm called Cobweb to perform automatic classification of C++ components. A slightly different version of the algorithm is used to retrieve components that best match a given query.

2. The ElvisC Repository

ElvisC attempts to provide an organized repository with minimal construction and maintenance costs to its user community. It does so by assisting in the formal characterization of repository submissions, and automatically organizing the repository contents to reduce searching. ElvisC is composed of three major components: 1) a case base of submitted assets, 2) an extensible asset characterization language similar to that first demonstrated in TEIRESIAS [Davis '82], and 3) an unsupervised classification system derived from the Cobweb inductive concept formation system [Fisher '87].

One novelty of this work is its use of an unsupervised classification system to solve the "distance metric" problem of case-based systems: determining the closest match to a given problem specification within the existing case base when a perfect match does not exist. It does this by treating the problem specification as if it were the characterization of a new case to be classified: the bin into which this problem characterization would have been placed is the bin that contains the closest matches to the new problem. The advantage of this classification technique is that it requires no specific knowledge of the application domain beyond the n-dimensional shape[2] of the existing case base. Thus the repository is both self-organizing and ontologically open.

2.1 Acquisition and The Feature Space

Characterization of assets serves two purposes: it describes the asset for potential reusers, and it provides the basic data with which to organize the library for reuse. To assist in the organization of the library, the characterization must be able to differentiate this asset from dissimilar assets and to relate this asset to similar assets. Thus characterizations must draw from a common vocabulary which can be extended as needed to distinguish new cases from existing cases. To meet these demands ElvisC acquires its characterization language interactively from users when they submit assets. This technique was first demonstrated in TEIRESIAS [Davis and Lenat '82] for the acquisition of diagnostic rules for blood diseases.

The asset characterization language, or *feature space*, is organized as a tree of keyword schemata. This tree is rooted in the *Feature* feature. The children of *Feature* are top level characterizations of an artifact, for instance the asset's function or the environment in which the asset was defined. These children may have children themselves, each of which represents a further refinement of its parent's concept. The entire feature space below the 'Feature' feature is malleable and can be extended or edited by contributing authors. Modifications to the feature space are tracked by the interface so that old definitions can be repaired to fit the new space.

Each schema specifies the keyword name, a textual definition of the keyword, an indication of how the keyword can be refined, and prompts for selecting from existing child features or soliciting new child features. The *Feature* feature is defined as follows:

> name: "Feature"
> definition: "This is the mother of all features."
> refinement type: Alternatives
> solicitation prompt: "What new feature would you like to add?"
> child selection prompt: "What features characterize this asset?"

When a user interacts with this schema they are presented with the child selection prompt, followed by a list of the refinements currently available for refining the *Feature* feature. If the user enters the name of one of the refinements presented, they then interact with the schema for that refinement. If the user enters a name that is not on the list of existing refinements, they enter a dialog for instantiating a keyword schema as a new refinement for the *Feature* feature.

---

[2] The *dimensions* of the space are the attributes by which the cases are classified; the *shape* is determined by the relative frequencies of cases along these dimensions.

The refinement type for the *Feature* feature is *Alternatives*. This refinement type allows the selection of one or more children in the description of an asset or the specification of a query. Alternative refinements of the root feature might be *Feature* ⇒ *Language* and *Feature* ⇒ *Performance*. An asset may be characterized by either or both of these alternatives. The *Alternatives* refinement type can be thought of as an inclusive "or" in the refinement tree. The existence of alternative refinements in the feature space is what differentiates our approach from those that use the feature space as a decision tree for directly sorting assets into asset type categories.

A second refinement type is *Options*. This method of refinement allows the selection of one and only one refinement. Subsequent selection of another refinement deselects previous refinements. The features *C* and *C++* could be options for the refinement of *Feature* ⇒ *Language* . This type of refinement can be thought of as an exclusive "or" in the refinement tree.

The third refinement type is *Arguments*. This method of refinement indicates that the refinements of a feature are jointly required for the feature to be present in a characterization. For instance, if one of the options for *Feature* ⇒ *Function* was *Storage*, we might specify two arguments for *Storage*: *What* (what is being stored) and *Where* (where is it being stored). Thus an asset characterized by the storage keyword would possess *Feature* ⇒ *Function* ⇒ *Storage* ⇒ *What* ⇒ ... and *Feature* ⇒ *Function* ⇒ *Storage* ⇒ *Where* ⇒ ... attributes. This type of refinement can be thought of as an "and" in the refinement tree.

Finally, a feature may not have any type of refinement. Such a feature is said to be "terminal". In the language example above, *C* and *C++* are terminal since no further refinement of these features is deemed necessary. If at some later point it became important to distinguish *ANSI C* and *K&R C*[3], then the previously terminal *C* feature token could be refined to provide these new options.

An asset is characterized by selecting or adding progressively more detailed refinements from the feature space. The description of an asset is equivalent to the set of nodes from the feature space that were traversed during asset definition. The query specification process works identically.

2.2 The Solution Space

The solution space is composed of a case base of assets and a concept hierarchy of asset types. The asset case base is organized as a forest of inheritance trees to allow derived class assets to inherit, optionally, the features of their parent class assets, thereby simplifying asset characterization. Assets that have no parent class specified, or assets which are written in a non-object-oriented language, are treated as base class assets and stored as children of a root asset. Thus in practice the asset case base tends to be top-heavy rather than deep.

Each asset record in the hierarchy contains that asset's features (as described above) and housekeeping data. Housekeeping data encompasses all information that is required for the asset but that does not contribute to its classification. Examples of housekeeping data include the asset's name, location, and author. Asset names are guaranteed to be unique so that they can serve as identifiers in the hierarchy. The asset's location is intended to be an accessible path name for the source or object files constituting the asset. In the future we hope to use this path name to support automatic delivery of selected artifacts to clients of the repository.

2.3 Organization of the Solution Space

Earlier versions of ElvisC (then called Elvis) viewed a query to the case base as constraints on which existing cases could be retrieved. If no existing cases satisfied all the constraints, Elvis had to selectively relax some or all of those constraints until a match was found. The problem was knowing which constraints to relax.

---

[3] ANSI C refers to the American National Standards Institute's definition of the C language. K&R C refers to the definition of the C language published by Kernighan and Ritchie prior to the ANSI work.

Suppose that a user is looking for an asset written in "C" which functions as a sorted collection of pointers, any of which can be located on average in O(log n) time. The features for this request would be:

Feature ⇒ *Language* ⇒ *C*

Feature ⇒ *Function* ⇒ *Collection* ⇒ *Sorted*

Feature ⇒ *Performance* ⇒ *Average case* ⇒ *O(log n)*

Suppose that no asset in the case base fulfills all these criteria. Relaxing the language constraint one level means that any language can be used. Relaxing the function constraint means that a collection of any type can be used. Relaxing the performance constraint means that any level of performance is adequate. The combined result of all of these individual relaxations would be the union of a number of possibly disjoint sets of assets. If we could order the relaxation options *a priori* based on their expected impact on the appropriateness of retrieved assets, we could successively apply those options until we got a set of assets of reasonable size and appropriateness. Otherwise we could end up with a very large set of assets, and no clue of how to order them for appropriateness other than Hamming Distance (the number of matching attributes). The uninformed constraint relaxation approach is equivalent to a brute force exhaustive search in the feature space, beginning with the specified constraints, in order to find exemplars in the asset space. The contents of the case base provide no insight into how to direct this search other than knowing when it is done.

ElvisC uses the Cobweb inductive concept formation system to locate the closest match to a desired case. Cobweb is an unsupervised incremental concept formation system originally developed by Douglas Fisher in his doctoral work at the University of California, Irvine [Fisher '87] . A thorough description of the algorithm and its Lisp implementation in Cobweb/3 is available in [McKusick and Thompson '90]. This method uses the contents of the solution space to guide the search. As a result, smaller sets of closest matches are retrieved, and a method for rank ordering the appropriateness of the retrieved cases is available. The Cobweb algorithm appears to perform in O(log *n*) time for classification, where *n* is the number of cases in the the case base. Since the size of the case base grows slowly and probably never exceeds thousands of cases, this performance should be adequate. Furthermore, the accuracy of an inductive method should grow as the case base grows.

2.3.1 Cobweb

Cobweb takes a stream of object identifiers and their descriptions, and incrementally organizes them into a concept hierarchy. As one proceeds deeper into the hierarchy the concepts formed become more and more specific until one reaches the leaves of the hierarchy in which a concept describes the attributes of a single object. Cobweb uses relative frequencies of attributes to construct the concepts, and uses those frequencies as probabilities when finding the best concepts (bins) to house the next object seen. Thus Cobweb does not employ any domain specific heuristics to do its classification, nor does it resort to human supervision in its construction or use of these bins.

ElvisC uses Cobweb to determine the closest matches to a prospective re-user's stated requirements. We treat the features comprising the requirement as the specification of a hypothetical object to be classified. Cobweb then filters the new hypothetical object down its classification hierarchy with the additional constraint that no new bins can be created to house the concept. The bin in which this problem characterization would have been placed (the host bin) is the bin which contains the closest match to the new problem. The bin which contains the host bin (the super-host bin) contains the next closest matches for the stated requirement. ElvisC currently returns the contents of these two bins in response to a query, with the asset from the host bin at the front of the list.

2.3.2 How Cobweb Works

Cobweb uses a metric called Category Utility (CU) to determine the relative goodness of different placements of a new submission in the current case base. The version of category utility used in this implementation is:

$$\text{CategoryUtility} = \{ \sum_{k=1..nk} P(C_k) * [\sum_{i=1..ni} \sum_{j=1..nji} P(A_i = V_{ji} \mid C_k)^2] - [\sum_{i=1..ni} \sum_{j=1..nji} P(A_i = V_{ji})^2] \} \div nk$$

where:

nk is the number of categories at the current level of the hierarchy
$P(C_k)$ is the probability of membership in category k relative to all other categories 1..nk
ni is the number of possible attributes
n_{ji} is the number of possible values for the ith attribute
$P(A_i = V_{ji} \mid C_k)$ is the probability that an attribute i has the value j given membership in category k
$P(A_i = V_{ji})$ is the probability that an attribute i has the value j for all categories 1..nk

This metric was first proposed and discussed in detail in [Gluck and Corter '85].

Cobweb categorizes a new submission to the library using a recursive algorithm. The algorithm begins at the root of the classification hierarchy which represents the bin containing all examples. At this root node the algorithm performs as follows:

1. Add the example as an exemplar of the node, updating the node's value frequencies according to the example's description.

2. If the node has no exemplars other than this new exemplar, terminate (classification is complete).

3. If the node now has two exemplars, create two child nodes (bins) to hold each of the exemplars and terminate (classification is now complete).

4. If the node now has more than two exemplars, it must have two or more child nodes. The options for placement of the new example are:

   4a: Create a new child (sub-bin) which will hold the example as its sole exemplar.
   4b: Place the exemplar in the existing child (sub-bin) which best fits the exemplar.
   4c: Create a new child (sub-bin) which will hold the exemplar by merging the two existing children which best fit the exemplar.
   4d: Replace the existing child which best fits the exemplar by its children, and then place the exemplar in one of those new children based on where it fits best.

   Each option is tried individually, with the results of option 4b used to identify the children to merge in option 4c and the child to split in option 4d. The category utility score is computed for the node after an option is performed, and then the results of the option are undone. After all options have been tried, the option which resulted in the best category utility score is selected. The algorithm, beginning at step 1, is then recursively applied to the selected sub-bin.

The $P(C_k)$ multiplier in the category utility function has the effect of limiting the number of children of a node to a range between two and some small constant (in our experience less than a dozen). Since this algorithm never considers more than the immediate children of the current node, and then selects at most one of those children for further classification, the performance of the algorithm is roughly O(log n).

2.3.3 Modifications to Cobweb

Cobweb has been applied to domains where the number of attributes is known *a priori* , and every exemplar is described by values for every attribute. In our reuse library, the attribute space is growing and exemplars are described only by a subset of that space.

The first problem is easy to solve. When our version of Cobweb is confronted with a previously unseen attribute, it updates all existing artifact descriptions with an "unknown" value for the new attribute.

The second problem is more fundamental. An attribute such as *Performance => O(1)* could be missing because the artifact was inadequately described, or because the attribute does not apply to the artifact (e.g., the artifact is a representation for calender dates, and thus a characterization of algorithm performance is not appropriate). The default implementation of CU and Cobweb will determine that two descriptions match on an attribute if both have a value of "unknown" for that attribute. This is correct for the latter case where the attribute is not appropriate to the definition of the artifacts. A modification to the calculation of CU is possible such that an unknown value for an attribute within an artifact description is treated as unique to this asset description, thus matching no other descriptions with a value of "unknown" for that attribute. This is correct for the former case where the artifact was inadequately described. Operationally, the use of unique unknowns causes a bushy classification hierarchy with a large number of bins holding a small number of exemplars. Many artifacts which we would have expected to be present in the same bin, at some level of the hierarchy, are instead only collectively present in the root node of the hierarchy. If we use non-unique unknowns these artifacts do tend to be present in the same bin at a level of the hierarchy below the root node, thus showing their similarity at that level of abstraction.

In ElvisC we treat "unknown" values for attributes as significant (non-unique) during the classification of contributed assets, but downplay the importance of unknown values during retrieval (treat them as unique). This decision is based on the assumption that retrieval requests are intentionally loosely described, but that asset descriptions are not. Since unknowns are treated as unique during retrieval, traversal of the classification hierarchy is guided more by attributes with known values than those with unknown values. Since during retrieval no actual modifications are made to the classification hierarchy, the deeper form caused by the normal (non-unique) algorithm is preserved. Finally, since the retrieval algorithm returns the one exemplar from the leaf of the classification hierarchy which best matched the request, followed by the exemplars of the parent of that leaf, a smaller and more focused set of closest matches is returned by the deeper hierarchy.

2.3.4 Archetypes and Prototypes

Cobweb has been augmented in ElvisC to keep two *synthetic artifact descriptions* at every classification bin. The *prototype description* records for each attribute the most probable value for that attribute given the contents of the bin. The *archetype description* records for each attribute the most probable value for that attribute other than the "unknown" value if there exist exemplars in the bin which have known values for that attribute.

We hope to use the archetype descriptions to assist the submitted artifact description process in guaranteeing that an artifact description is adequately specified. Once an author has made a preliminary characterization of his or her artifact, the interface will do a preliminary classification to identify an appropriate archetype. The interface will then use archetype attributes with known values which correspond to unknown values in the proffered artifact description to prompt the author for additional information to complete the description.

We hope to use the prototype descriptions as the basis for functionality-based asset browsing, an alternative to the inheritance-based asset browsing currently employed.

3. An Experiment

ElvisC is, as its full name implies, an experiment. The hypothesis is that the Cobweb algorithm can organize software components for efficient and reliable retrieval of closest matches without human supervision, and without the direct encoding of closeness information. The experimental apparatus is the ElvisC implementation itself, augmented by methods for quantifying its performance in an attempt to refute the hypothesis. The experimental subjects are the potential users of the system, and their collective solutions and problems.

There are two types of errors that could refute the hypothesis. The first type is errors of omission, where an applicable case is not retrieved in response to a query. These errors could be quantified by recording examples of retrieval requests, and comparing the responses of ElvisC to the responses of a human librarian who is familiar with the case base. Errors of omission are inversely related to the reliability of retrieval. The second type is errors of inclusion, where patently irrelevant cases are returned as the

primary candidate solutions. Such errors could also be quantified by comparing the responses of a human librarian to those of the system. Errors of inclusion are inversely related to retrieval efficiency. Of the two types of errors, errors of omission would provide a more serious challenge to the hypothesis.

In this experiment outside subjects were not available. Instead we evaluated an automatically generated classification hierarchy directly, looking for characteristics that would lead to the two types of errors. To minimize errors of omission, the classification must co-locate (at some level) assets that solve the same problem, rather than scatter them about the hierarchy. To minimize errors of inclusion, the classification must be deep enough to effectively partition small numbers of solutions. In a hierarchical classification scheme these two characteristics of good organization are themselves related: the higher the level at which similar assets are co-located, the larger the number of assets that must be inspected in detail to identify a best candidate (and hence the lower the efficiency of the overall retrieval process).

In [Bewtra and Lide '92], investigators manually characterized and classified a set of 56 source code components from a Code 522 reuse directory. In this experiment, we entered the same classes into ElvisC using the feature descriptions provided in that paper[4]. The fifty-six classes were added to eleven classes from the National Institute of Health C++ class library, which had been previously entered into ElvisC, so that in total 67 classes were described and classified. The resulting classification hierarchy was printed out and evaluated in order to determine its support for efficient and reliable retrieval of artifacts by an unfamiliar user. The description language which was developed while entering these classes and the classification hierarchy that resulted are presented in detail in [Henderson '92].

4. Results

Figure 1 presents the classification derived manually in [Bewtra and Lide '92]. Dotted boxes denote categories of C++ classes that were treated in that paper but were not included in this experiment because of time limitations. The manual classification was constructed without any knowledge of the results (indeed, the existence) of this experiment. Conversely the experiment was performed without access to the results of the manual classification.

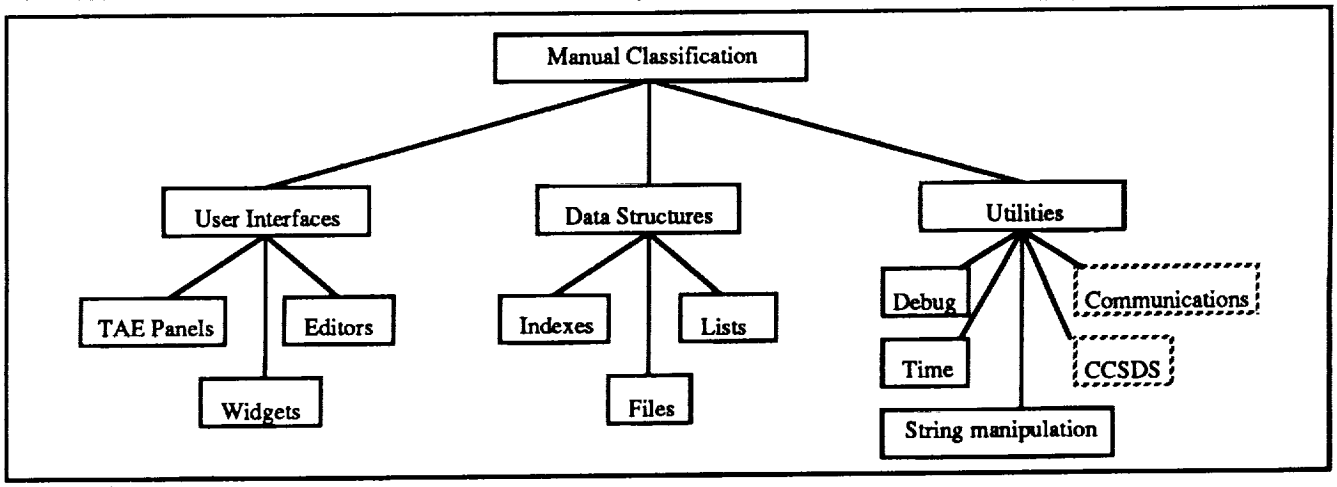

Figure 1.

It is interesting to note that this manually derived hierarchy went through several revisions during the writing of [Bewtra and Lide '92]. Since there is no performance task associated with this hierarchy, its construction is the result of subjective decisions on how to partition the C++ classes that were evaluated. It is reasonable to assume that these decisions were influenced by the authors' contact with a large number of software modules that were not part of that evaluation set.

[4]In some cases the provided descriptions were very terse, and several descriptions referred to base classes that were not described elsewhere in the paper. Thus we were not always able to provide distinctive or complete descriptions for each asset, and this may have diminished the effectiveness of the automatic classification.

Figure 2 presents the top levels of the automatically generated classification hierarchy. The nodes in this tree will be called *concepts* to differentiate them from the nodes in the manual classification hierarchy which we will call *categories*. Since ElvisC does not currently generate titles or descriptions for concepts, we have entered titles based on examination of the concept exemplars and characteristic features. Characteristic features were identified by a combination of two statistics. The first statistic is the probability that an exemplar of the concept has the feature. The second is the probability that an asset which has that feature is an exemplar of the concept. To be a characteristic feature for a concept, both of these probabilities had to be above 50%[5].

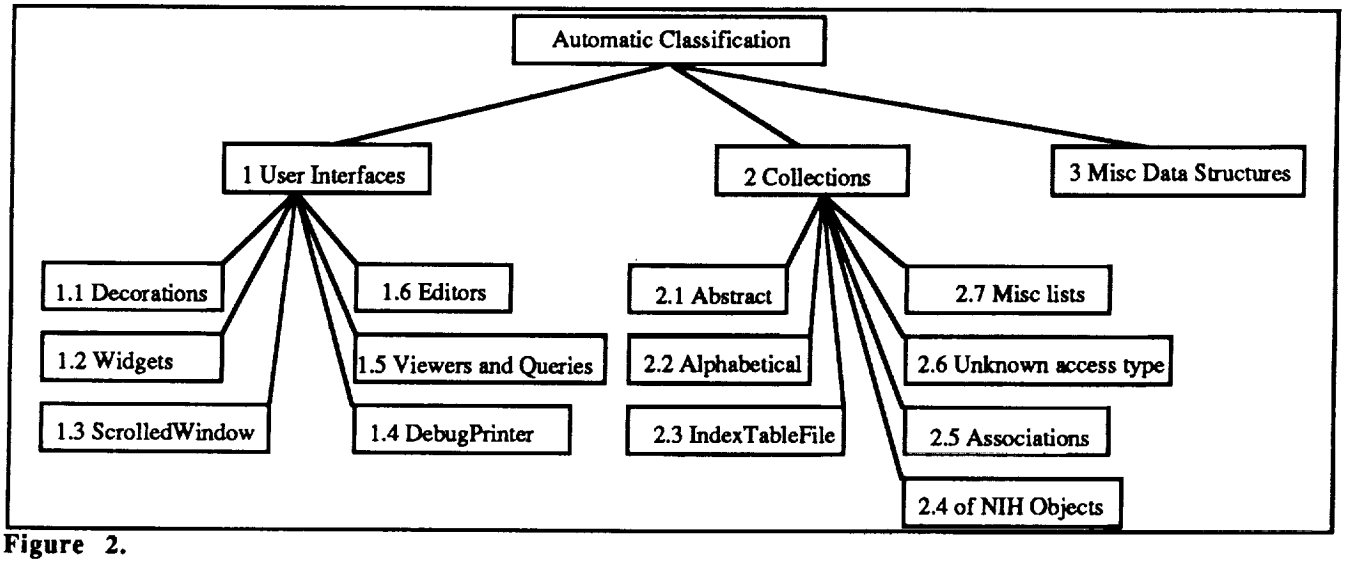

Figure 2.

The top level of the automatically generated classification hierarchy contains three concepts. Examination of the characteristic features reveals that they represent user interfaces, lists, and data structures respectively. These top level concepts are then further divided into sub-concepts that have been labeled in the diagram based on their contents and characteristic features.

4.1 Description of the Cobweb Classification Hierarchy

Concept 1 contains twenty five user interface classes. There are no user interface assets contained in any of the other top level concepts, so the retrieval of user interface components will be reliable at this level. The first sub-concept of concept 1, labeled Decorations, includes two widget classes which perform no function other than to provide visual feedback for a host window. Neither of these widget classes supports user input, and their collocation within this concept seems reasonable. The second sub-concept (1.2) also includes widget classes, but these classes allow user interaction. Next there are two singleton sub-concepts, one for the asset ScrolledWindow (1.3) and one for the asset DebugPrinter (1.4). Although DebugPrinter is a novel class in the library and, as such, deserves its own concept, ScrolledWindow seems subjectively to be a good fit with the Viewers and Query Dialogs concept. The placement of ScrolledWindow in a concept of its own could adversely effect the efficiency of retrieval. The next sub-concept (1.5) represents interaction windows which are not used for editing a file. The final sub-concept of concept 1 represents editors.

Concept 2 contains eighteen *collection* assets; code components that collect multiple instances of some type such as linked lists. There are no collection assets contained in any of the other top level concepts, so the retrieval of collections will be reliable at this level. The concept is further subdivided into sub-concepts of abstract base classes, alphabetically organized lists, lists of NIH Objects, associations, lists with unknown

[5] We found through experimentation that this threshold revealed features which both described the concept and differentiated it from its siblings.

access methods, and a concept for lists which presumably did not fit elsewhere. The clustering of assets is good for each lower level concept with two exceptions. First, NIHDictionary (a sub-concept of 2.4) and the elements of the Associations concept (2.5) both provide associations between a key and a value. A query leading to the associations concept will not turn up NIHDictionary and vice versa. This problem points to a weakness in constructing exclusive categories (categories that do not share elements with their siblings): at lower levels a decision must be made on how to discriminate classes, and that decision may not perfectly represent the relative importance of different features in the class descriptions. In this case, the classification decision appears to have been made based on the types of values collected: pointers to NIH objects for concept 2.4 and untyped pointers for concept 2.5. The impact of these decisions could be reduced through the use of multi-branch search within the classification tree.

Concept 3 contains every asset which is not an interface or a collection. The sub-concepts within concept 3 are not well formed, having few if any characteristic features to distinguish them. Several appropriate clusters seem to be forming, such as iterators and date/time representations, but overall these sub-concepts seem to be in a very nascent state. This could be symptomatic of poor descriptions, or of a lack in regularity within this subdomain.

4.2 Reliability Evaluation of the Cobweb Classification Hierarchy

Retrieval reliability is dependent on the system's ability to appropriately cluster similar assets within the repository. For the purposes of this evaluation we have presumed that the clustering presented in the manual classification is correct, although that organization has gone through multiple revisions and the assessment of its quality is somewhat subjective.

Table 1 shows how the assets within the automatically generated concept hierarchy fall within the manual hierarchy for the *User Interface* category. The column headings represent the sub-categories of *User Interface* within the manual classification. The row headings represent the sub-concepts of *User Interface* for the automatic classification, followed by the total number of assets identified with that concept in parentheses. Numbers within the cells of the table indicate how the assets from a concept fall into the manual categories.

Generated Concepts	Panels	Widgets	Editors	Non user interface
Decorations (2)		2		
Widgets (5)		5		
Viewers (12)	9	1	2	
Editors (4)			4	
Other UI (2)		1		1

Table 1.

Every asset in the *User Interface* category was identified within the *User Interface* concept, so reliability for this concept was 100%. One additional asset was included in the *User Interface* concept which was not included in the *User Interface* category, so the match at the *User Interface* level was 24 + 25 = 96%.

Although the sub-concepts formed within the *User Interface* concept do not match the manual partition of the *User Interface* category, a mapping between them is evident from inspection of the data and from the statistically generated concept descriptions. First, the *Decorations* and *Widgets* concepts represent a partition of the manually generated *Widget* category. Second, the *Viewers* concept is very similar to the manual *Panels* category. Given this mapping, correct classification inside the User Interface Concept is 20 + 25 = 80%.

Table 2 shows how assets within the *Collections* concept fall within the partitions of the *Data Structures* and *Utilities* categories. Within the manual classification most of these assets inhabit multiple categories. These assets are arbitrarily labeled by letters to indicate how a member of a concept falls into the manual categories.

Generated Concepts	Indexes	Files	Lists	Time	String	Debug
Abstract (2)			a, b			
Alphabetical (2)	c	c	d		d	
Linked Lists (3)			e, f, g		g	
Misc Lists (2)			h, i	h	i	
Other (1)	k	k				

Table 2.

Every asset in the *Lists* partition of the *Data Structures* category was identified by the *Collections* concept, so reliability for Lists was 100%. Two assets representative of the *Collections* concept were not part of the manual *Lists* category, so the match between the concept and the category was 8 + 10 = 80%. For completeness it should be noted that the *Associations* and *NIH List* concepts were not part of the manual classification and so were omitted from this discussion.

Table 3 shows how assets within the *Misc Data Structures* concept fall within the partitions of the *Lists* and *Utilities* categories. Some of these assets inhabit multiple categories, so letters are used to indicate how a member of a concept falls into the manual categories.

Generated Concepts	Indexes	Files	Lists	Time	String	Debug
Reporters (2)		a				b
Other (8)	a	a,b,c,d	e	f, g	h	
Iterators (1)			k			
Dates (3)				m, n, o		

Table 3.

We can compute the reliability of the *Misc Data Structures* concept by comparing it to the partitions of the *Data Structures* and *Utilities* categories except for the *Lists* partition of those categories, which should have been accounted for by the *Collections* concept. Given this interpretation the reliability for the *Misc Data Structures* concept was 12 + 14 ≈ 86%.

The last two tables show that the categories which straddle concepts are *Indexes*, *Files*, *Time*, and *String*. Reliability problems with Time and String retrieval cease if the requestor indicates that a collection of these types is desired. Retrieval reliability for assets manually classified within the Indexes and Files categories is not good, and reflects the nascent state of the associated concepts within the automatic classification.

One cause for the divergence between the manually generated and automatically generated hierarchies is a difference in the definition of the classification process. Cobweb creates exclusive concepts (concepts that do not share elements with their siblings). The manual classification generated in [Bewtra and Lide '92] includes assets in multiple categories. Cross-referencing enhances the reliability of retrieval at the expense of efficiency of retrieval. Thompson [Thompson '92] has suggested that a more sophisticated "beam" search in response to queries would enhance the reliability of retrieval with a fixed decrease in retrieval efficiency. This may represent an alternative to non-exclusive catagories. Another possible solution would be to use a classification algorithm that assigns assets with varying probabilities to different categories, such as AutoClass [Cheeseman '88].

4.3 Efficiency Evaluation of the Cobweb Classification Hierarchy

Retrieval efficiency is inversely related to the number of matches returned for a search request, thus the maximum efficiency that could be attained is 1.0. Since the manually constructed classification hierarchy is only two levels deep, efficiency is limited by the average size of a second tier partition. For the assets considered in this experiment the average size of a second tier partition is 6.67 elements, and so the average efficiency would be 1 + 6.67 or approximately 0.15.

Cobweb produces a concept hierarchy whose leaves contain a single exemplar. Thus if the generated concept hierarchy was perfect, the average number of returned cases for a request would be one! Our data for the hierarchy that was actually constructed suggests that this strategy would be too unreliable. If we were to restrict retrieval to second tier concepts we would attain about 80% reliability. The average size of a retrieved set would then be 3.43 elements, and the average efficiency would be approximately 0.29.

What this suggests is that an automatically constructed but manually corrected concept hierarchy would attain higher efficiency with less effort than a purely manually constructed hierarchy, while still retaining the same reliability.

5. Summary

At its upper level the automatic classification performed by ElvisC is of high quality and fulfills our criteria for reliable retrieval. For the first two categories this claim can be extended to lower level subcategories, as can the claim for efficient retrieval. The last category, however, demonstrates the frailty of weak-theory systems: all that the algorithm has to go on are the descriptions given it of the sixty-seven classes, and specifically of the twenty-four assets that fall into this category. Cobweb's strength lies in its ability to exploit the fact that the world is sparsely populated from the set of things that *could* be. The algorithm is not effective until enough cases have been seen and the regularity in how attributes co-exist in objects becomes evident.

The degree of correlation between the manually constructed classification and the automatically generated classification is remarkable considering that the automatic algorithm was unsupervised and so made its own decisions on the kinds of concepts to form and then on how to further partition those concepts. The results show that, with a relatively small number of samples, a good classification can be arrived at without human supervision or embedded knowledge of the domain.

This technique allows the reuse of work products by a large group of participating individuals when a domain theory on how to organize the repository is not available. We hope to continue to accumulate evidence to support these findings, and thereby establish this novel application of concept formation.

References

Arango, G. and Teratsuji, E. (1987) *Notes on the Application of the Cobweb Clustering Function to the Identification of Patterns of Reuse.* Technical Report ASE-RTP-87. Advanced Software Engineering Project, Department of Information and Computer Science, University of California, Irvine, CA.

Basili, V. (1990) Viewing maintenance as reuse-oriented software development. *IEEE Software*, January 1990.

Basili, V. and Rombach, H.D. (1988) *Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment.* Technical Report UMIACS-TR-88-92, Computer Science Department, University of Maryland, College Park, MD. December 1988.

Berlin, L. (1990) When objects collide: experiences with reusing multiple class hierarchies. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 181-193. ACM Press, October 1990.

Bewtra, M., and Lide, D. (1992) *Code 522 Technology Component Analysis* Technical report to NASA Goddard Space Flight Center. CTA Inc. Rockville, MD.

Boland, D., Booth, E., Green, D., Odt, T. (1991) *Combined Operational Mission Planning and Attitute Support System (COMPASS) Operation Concepts.* NASA/Goddard Space Flight Center, Flight Dynamics Division (Code 550) report number 550-COMPASS-107. May 1991.

Brooks, F. (1987) No silver bullet: essence and accidents of software engineering. *IEEE Computer*, Vol. 20, No. 4. April 1987.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., and Freeman, D. (1988) A Bayesian classification system. Proceedings of the Fifth International Conference on Machine Learning, pages 54-64. Morgan Kaufman Publishers, Ann Arbor, MI.

Cox, B. Planning the software industrial revolution. *IEEE Software*, Vol. 7, No. 6. November 1990.

Davis, R. and Lenat, D.B. (1982) *Knowledge-Based Systems in Artificial Intelligence* McGraw-Hill Inc.

Deerwester, S., Dumais, S., Furnas, G., Landauer, T., and Harshman, R. (1990) Indexing by latent structure analysis. *Journal of the American Society for Information Sciences*, 41(6), pages 391-407.

D'Ippolito, R. Using models in software engineering. *Association for Computing Machinery*, September 1989, pages 256-261.

Fisher, D. and Pazzani, M. (1991) Models of concept learning. In *Concept Formation: Knowledge and Experience in Unsupervised Learning*, ed. D. Fisher, M. Pazzani, and P. Langley. Morgan Kaufman Publishers, San Mateo, CA.

Fisher, D. (1987) *Knowledge acquisition via incremental conceptual clustering* Doctoral dissertation, Department of Information & Computer Science, University of California, Irvine.

Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O., and Pintado, X. (1990) Class management for software communities. *Communications of the ACM*, Vol. 33, No. 9, pages 90-103. September, 1990.

Goldberg, A. (1984) *Smalltalk-80: The Interactive Programming Environment.* Addison Wesley Publishing Company, Reading MA.

Henderson, S. (1992) *Automated Software Component Classification using ElvisC: Results and Discussion.* Technical report to NASA Goddard Space Flight Center. CTA Inc. Rockville, MD.

Hughes, P. and Luczak, E. (1991) *The Generic Spacecraft Analyst Assistant (GenSAA): A Tool for Automating Spacecraft Monitoring with Expert Systems* 1991 Goddard Congerence on Space Applications of Artificial Intelligence, Greenbelt, Maryland.

Knight, J. (1992) *Issues in the Certification of Reusable Parts.* Technical Report TR-92-14, Department of Computer Science, University of Virginia.

Krone, J. (1988) *The Role of Verification in Software Reusability.* Ph.D. Thesis, Department of Computer and Information Science, Ohio State University. August 1988.

Maarek, Y. and Smadja, F. (1989) Full text indexing based on lexical relations. An application: software libraries. In *Proceedings of SIGIR '89*, pages 198-206, ACM Press, Cambridge MA.

Mark, W., Tyler, S., McGuire, J., and Schlossberg, J. (1992) Commitment -based software development. *IEEE Transactions on Software Engineering*, October 1992.

Meyer, B. (1988) *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science, New York, NY.

McKusick, K., & Thompson, K. (1990). *Cobweb/3: A portable implementation.* Technical Report No. FIA-90-6-18-2. NASA Ames Research Center, Artificial Intelligence Research Branch. Moffett Field, CA.

Moore, J.M. and Bailin, S. (1991) Domain analysis: framework for reuse. In *Domain Analysis and Software Systems Modelling*, ed. R. Prieto-Diaz and G. Arango. IEEE Computer Society Press.

Neighbors, J. (1989) Draco: A method for engineering reusable software systems. In *Software Reusability*, ed. T. Biggerstaff and A. Perlis, Vol. 1, pages 295-319. Addison-Wesley Publishing Company.

Ostertag, E., Hendler, J., Prieto-Diaz, R., and Braun, C. (1992) Computing similarity in a reuse library system: an AI-based approach. *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 3. July 1992.

HR (1989) *Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation.* Staff study by the Subcommittee on Investigations and Oversight, Committee on Science, Space and Technology. U.S. House of Representatives. August 3, 1989.

Measday, A. (1991). *Transportable Payload Operations Control Center (TPOCC) Implementation Guide for Release 3.* Technical report to NASA/Goddard Space Flight Center by Computer Sciences Corporation and Integral Systems, Inc. NASA/Goddard report number 511-4SSD/0291.

Prieto-Diaz, R. (1991) Implementing faceted classification for software reuse. *Communications of the ACM*, Vol. 34, No. 5. May 1991.

Prieto-Diaz, R. and Arango, G., ed. (1991) *Domain Analysis and Software Systems Modelling.* IEEE Computer Society Press.

Reich, Y. (1991a) *Building and Improving Design Systems: A Machine Learning Approach.* Ph.D. Thesis, Engineering Design Research Center, Department of Civil Engineering, Carnegie Mellon University, EDRC 02-16-91.

Reich, Y. (1991b) Constructive induction by incremental concept formation. In *Artificial Intelligence and Computer Vision*, ed. Y.A. Feldman and A. Bruckstein. Elsevier Science Publishers.

Reich, Y. and Fenves, S. (1991) The formation and use of abstract concepts in design. In *Concept Formation: Knowledge and Experience in Unsupervised Learning*, ed. D. Fisher, M. Pazzani, and P. Langley. Morgan Kaufman Publishers, San Mateo, CA.

Simos, M. (1991) The growing of an organon: a hybrid knowledge-based technology and methodology for software reuse. In *Domain Analysis and Software Systems Modelling*, ed. R. Prieto-Diaz and G. Arango. IEEE Computer Society Press.

Sperry (1985). *MSOCC Applications Executive (MAE) Software Requirements Specification.* Technical report to NASA/Goddard Space Flight Center. Sperry Corporation, November 1985.

SPC (1991) *Synthesis Guidebook.* Technical Report SPC-91122-MC, Software Productivity Consortium, Herndon, VA.

Thompson, K. (1992) Personal correspondence.

WEC (1979). *Control Center Standard Software Final Report.* Technical report to NASA/Goddard Space Flight Center. Westinghouse Electric Corporation. July 27, 1979.