An Intelligent Tool For Re-engineering Software Modularity

Robert W. Schwanke

Siemens Corporate Research, Inc.
755 College Rd. East
Princeton, NJ 08540

This paper describes a software tool that provides heuristic modularization advice for improving existing code. A heuristic design similarity measure *is defined, based on Parnas's* information hiding principle. *The measure supports two services:* clustering, *which identifies groups of related procedures, and* maverick analysis, *which identifies individual procedures that appear to be in the wrong module. The tool has already provided useful advice in several real programming projects. The tool will soon incorporate an automatic tuning method, which allows the tool to "learn" from its "mistakes", adapting its advice to the architect's preferences. A preliminary experiment demonstrates that the automatically tuned similarity function can assign procedures to modules very accurately.*

1. Introduction

A medium or large scale software project's success depends heavily on how well the software is organized, because the organization affects understandability, modifiability, integratability, and testability. Unfortunately, because software changes rapidly, even during maintenance, its organization often deteriorates. Each time that a programmer adds a new procedure to the system, he must decide which existing module he should place it in. Sometimes, he should form a new module, containing this object and objects drawn from existing modules, but the mental and administrative effort involved often deters him. Either way, the programmer often has only a worm's eye view of the system, from the corner where he is working, and makes his organizational decisions accordingly.

This problem is exacerbated by the fact that most widely-used programming languages still have inadequate scope-control facilities, so that modularity is a matter of programer self-discipline, and is not normally enforced by the language support tools.

Sooner or later, someone on the project usually notices that the organization has deteriorated. Then, a small team of experts is appointed as "architects", to analyze and reorganize the system. However, their task is even more formidable than the programmer's, because they must understand many more system-wide interrelationships, and must carry out widespread changes without causing the system to break. Furthermore, because the programming language and tools do not support modularity adequately, they must analyze actual cross-reference information to deduce the scopes of many program units, rather than relying on specifications.

The goal of the Arch project is to help rescue the architects from their predicament, by providing them with intelligent tools for analyzing the system's structure, reorganizing it, documenting the new structure, and monitoring compliance with it, so that significant structural changes can be detected and evaluated early, before they become irreversible.

Arch is a graphical and textual "structure chart editor" for maintaining large software systems. It extracts cross reference data from the code itself and, using the current subsystem tree as a guide, creates several kinds of graphical and textual views of the cross reference data, at varying levels of detail. In order to help create subsystem trees where none existed before, Arch provides a clustering algorithm that groups related procedures into modules. In order to improve the quality of existing modules, Arch provides a "critic", which identifies individual procedures that apparently violate good information hiding principles.

Overview of the paper

This paper describes a set of methods for providing heuristic advice on modularity, including an adaptation mechanism that automatically tunes the heuristic to the preferences of the software architects.

We begin by discussing the *information hiding principle* [1], and then describe a heuristic measure of information sharing. Next we describe two services that provide heuristic advice for modularizing existing code, and the results we have achieved with these services. One service, *clustering*, identifies clusters of procedures that share enough design information that they belong together in the same module. The other service, *maverick analysis*, identifies individual

CH2982-7/91/0000/0083$01.00 © 1991 IEEE

Recommended by: Dan Berry

procedures that appear to be in the wrong module, because they share more information with procedures in other modules than with procedures in their own module.

Both services present lists of suggestions, which the architect can accept or reject. The lists are long enough that they must be prioritized, so that the architect can tackle the problems "worst first". As she does so, she sometimes finds that she disagrees with Arch's recommendations, because (for example) she believes that encapsulating one data type is more important than encapsulating another. Since the similarity measure incorporates a weight representing the importance of each non-local identifier in the system, it can be adapted to the architect's preferences by increasing the weights of some identifiers and decreasing others. Informal experiments on real, production code show that heuristic analysis provides useful information to practicing maintainers, and that hand-tuning a few of the weights can make Arch and the maintainer agree most of the time.

However, the tuning process is too tedious and demanding to expect an architect to do it. Instead, we have developed an automatic tuning method. It is essentially a curve-fitting method, which takes a set of approved modules and their approved members, and finds coefficients for the similarity measure that minimizes the number of apparently misplaced procedures. The method is a gradient descent method that combines and extends several neural network design and training methods. Although the implementation details are beyond the scope of this paper, we describe the results of our experiments, which show that an automatically-tuned similarity function can assign a new procedure to the correct existing module with very high accuracy.

One potential problem with automatic tuning is that, if the measure is tuned too closely to the data, then Arch will have no suggestions to make, because the fitting process assumes that the given modules are correct. To prevent this, we give the weight coefficients initial values based based on objective information measures of the code itself, without any architect's input, and create an initial list of suspect procedures. The weights are changed only when the architect rejects a suggestion, and are only changed "just enough" to make Arch agree with the architect.

By this adaptation method, the architect is freed from laborious hand-tuning. She only needs to say "yes" or "no" to specific suggestions, and can expect the tool to adapt to her preferences.

2. Information Hiding

One of the most influential writers on the subject of modularity has been David L. Parnas. In 1971, he wrote of the information distribution aspects of software design (italics his):

> "*The connections between modules are the assumptions which the modules make about each other.* In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions." [1]

The same year he formulated the "information hiding" criterion, advocating that a module should be

> "... characterized by a design decision which it hides from all others. Its interface or definition [is] chosen to reveal as little as possible about its inner workings." [2]

According to Parnas, the design choices to hide are those that are most likely to change later on. Good examples are data formats, user interface (I/O formats, window vs. typescript, choice of window management system), hardware (processor, peripheral devices), and operating system.

2.1. An Example

In practice, the information hiding principle works in the following way. First, the designers identify the role or service that the module will provide to the rest of the system. At the same time, they identify the design decisions that will be hidden inside the module. For example, the module might provide an associative memory for use by higher-level modules, and conceal whether the memory is unsorted or sorted, all in memory or partly on disk, and whether it uses assembly code to achieve extra-fast key hashing.

The module description is then refined into a set of procedures and data types that other modules may use when interacting with the associative memory. For example, the associative memory might provide operations to insert, retrieve, modify, and remove records. These four operations would need parameters specifying records and keys, and some way to determine when the memory is full. It would declare and make public the data types "Key" and "Record", and the procedures "Insert", "Retrieve", "Modify", and "Remove".

Next, the associative memory module is implemented as a set of procedures, types, variables, and macros that together make, for example, a large in-core hash table. The implementation can involve additional procedures and types beyond the ones specified in the interface; only the procedures belonging to that module are permitted to use these "private" declarations. Many design decisions are represented by specific declarations, such as

```
HashRecord array
    HashTable[TableSize]
```

which embodies the decision to store hash records in a fixed-size table rather than, say, a linked list or tree. Procedures that depend on such design decisions normally use the corresponding declarations, for example,

```
proc Retrieve(KeyWanted: Key)
    Index = Hash(KeyWanted)
    if HashTable[Index].Key
            equals KeyWanted
        return HashTable.Record
    else return FAILURE
```

Procedures outside the associative memory module cannot, for example, determine which order the records are stored in, because they cannot use the name `HashTable`. Later, if the implementer should decide to replace the hashing algorithm, or even to use a sorted tree, all of the code that he would need to change would be in the associative memory module.

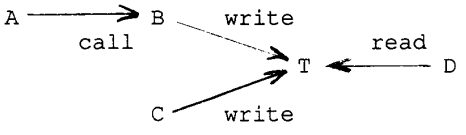
2.2. A Heuristic

The example above leads us to a simple *information sharing heuristic* for detecting when two procedures share a design decision:

> If two procedures use several of the same unit-names, they are likely to be sharing significant design information, and are good candidates for placing in the same module.

2.3. Coupling: Control, Data, or Design?

A unique aspect of our research is that we measure design coupling, rather than data or control coupling. A simple example will illustrate the difference:



This diagram contains four procedures, A, B, C, and D and a table, T. Procedure A calls procedure B to write information into table T, and calls D to read information from the table. Procedure C also writes information into table T. Procedures A and B have a *control link* between them, because A calls B. Procedures B and D have a *data link* between them, because data passes from B to D through the table. Likewise, A and B are data-linked through parameters, and C and D are data-linked through T. However, B and C are **not** data-linked, because because both of them put data into T, but neither one takes data out. Finally, B, C, and D have a **design link** among them, because all three share assumptions about the format and interpretation of table T. If one of the procedures ever needs to be rewritten in a way that affects the table T, the other two should be examined to see if they require analogous changes.

Before Parnas's work, it was commonplace to divide a system into modules that each represented a major computational step of the program. For example, a compiler would be divided into a lexical analyzer, a syntax analyzer, a semantic analyzer, and an optimizer. The lexical analyzer would include a procedure for inserting symbols into the symbol table; the other modules would contain routines for retrieving information from the symbol table. The format of the symbol table itself would be exposed to all of the modules, so that a change in its format required the programmer to review every module to see what the impact would be. Nowadays, programmers generally agree that it is more important to group together procedures that share data, than to group procedures that call one another.

2.4. Real Life Is Not So Simple

It would be nice if the clear, simple concepts contained in a system's original design were faithfully adhered to throughout the software's lifetime. However, the implementation process always uncovers technical problems that lead to changes in the design. Furthermore, design decisions are almost never so clearly separable that they can be neatly divided into subsystems and sub-subsystems. Each decision interlocks with other decisions, so that inevitably there are some decisions that cannot be concealed within modules, even though they are likely to change. These typically show up as public variables and unprotected data types.

Private declarations are not the only design decisions that may be shared among procedures. Module interface specifications also represent design decisions, although the designers hope that they will change less often. Even so, in many cases a certain interface procedure is only used in one or two other modules in a system, and represents a design decision on which all of the using procedures depend.

In the final analysis, good modularity is highly subjective. Not only must the designers select good abstract roles for the modules to implement, but they must try to predict what kinds of changes are likely to happen to the system in the future. Then they must determine which design decisions can be hidden within modules, and which ones must be shared. Finally, they must adapt the module specifications to the project team that is building them, incorporating both technical and non-technical influences.

Therefore, modularization as a reverse-engineering process must be treated heuristically, rather than by a formal set of rules. The information hiding heuristic suggests that "belonging together" is proportional to "shared declarations". Arch uses a similarity function that measures information sharing based on shared declarations, and uses it to give the architect advice on how to modularize or remodularize a system.

3. Measuring Information Sharing

To turn the information sharing heuristic into an actual similarity function, Arch profits from research on human similarity judgment, in the field of cognitive science. One particular model, Tversky's Ratio Model [3], corresponds to our intuitive notion of how humans judge that two procedures share design information. In this section we outline that model, and describe how we have adapted it to our problem domain. First, however, we define the software features on which the similarity function is based.

3.1. Features of Software

The information sharing heuristic is based on the non-local names that procedures use. More formally, a non-local name is any name whose scope includes two or more procedure bodies. Arch assigns a unique identifier to each such name, to distinguish multiple declarations of the same identifier (in different scopes). Every non-local name is a potential feature name. Every non-local name appearing in the body of a procedure is a feature of that procedure.

Sometimes, two or more procedures are placed together in the same module because they are called from the same other procedures. Therefore, whenever procedure A calls procedure B, not only does A receive the feature "B", but B receives the feature "called-by-A".

For the C language, we are using a cross-reference extractor based on `cxref` that collects all occurrences of non-local names, including the names of procedures, macros, typedefs, variables, and even the individual field names of structured types and variables.

3.2. Requirements on a Software Similarity Measure

In agreement with Tversky's work, we have identified the following requirements for a software similarity measure:

- *Matching*: Similarity must be a function of the features common to the two procedures, or distinctive to one or the other. It should not be a function of how many possible features are missing from both procedures.
- *Monotonicity*: Adding a common feature to two procedures must increase their similarity. Adding a distinctive feature to one of them must decrease similarity.
- The relative significance of two features must be independent of whether they are common or distinctive. As a whole, common features may be more or less significant than distinctive features, but individual variations are not permitted.
- The similarity between two procedures with no common features must be zero.
- **Exception**: Arch's actual similarity measure has an additional term representing whether or not one of the procedures calls the other. This term is ignored in the requirements above.

The following mathematical development, derived from Tversky's work, is not essential to understanding Arch. The uninterested reader may skip to section 4 without loss of continuity.

3.3. Matching and Monotonicity

Let $A,B,C,...$ be objects described by sets of features $a,b,c,...$, respectively. Each member of a feature set is the name of a characteristic that is true of the corresponding object. Then common and distinctive features are defined as:

$a \cap b$	The set of features that are common to A and B.
$a - b, b - a$	The sets of features that are distinctive to A or B, respectively.

A similarity function, *SIM*, has the *matching* property if there exists functions F and f such that

$$SIM(X,Y) = F(f(x \cap y), f(x - y), f(y - x))$$

This assures that the significance of a set of features occurring in one or both of the compared objects is computed without reference to whether the features are common or distinctive. It also assures that similarity is independent of any other features.

A similarity function, *SIM* has the *monotonicity* property if

$$SIM(A,B) \geq SIM(A,C)$$

whenever

$$a \cap b \supseteq a \cap c,$$
$$a - c \supseteq a - b, \text{ and}$$
$$c - a \supseteq b - a$$

and, furthermore, the inequality is strict whenever at least one of the set inclusions is proper.

Note that monotonicity is based only on the set inclusion ordering, and not on the number or weight of the features. Thus, mononicity does not by itself ensure that the more-similar-than relation is a total ordering.

3.4. Arch's Similarity Function

Tversky proposed two similarity functions that were intuitive, easy to compute, and satisified the matching and monotonicity properties. One of them, the Ratio Model, seems well suited to comparing procedures, because its value is zero in the absence of shared features. Arch's similarity function, although developed independently, has a nearly identical form. First we describe its components:

$w_x > 0$ *weight of feature x*

$W(X) = \sum_{x \in X} w_x$ *Weight of X*

$Linked(S,N) = \begin{cases} 1, & \text{if } A \text{ calls } B \text{ or } B \text{ calls } A \\ 0, & \text{otherwise} \end{cases}$

The weight of a feature is a positive, real number representing its importance, relative to other features. The weight used is the same whether the feature is common or distinctive. Although Tversky's theory permits other aggregate weight functions, we have found the linear sum to be sufficient. The predicate *Linked* is needed because caller-callee relationships must be considered in module formation, in addition to information sharing. Observations of real software confirm that small procedures with few non-local identifiers in them are frequently grouped with their callers.

The similarity function used in Arch is defined as follows:

$$Sim(A,B) = \frac{W(a \cap b) + k \times Linked(S,B)}{n + W(a \cap b) + d \times (W(a-b) + W(b-a))}$$

Notes:

- All coefficients are non-negative.
- Only shared and distinctive features count. The similarity of two procedures is not affected by adding unrelated declarations (features) to the program.
- Similarity increases with shared features and decreases with distinctive features. The constant d controls the relative importance of common and distinctive features.
- If there are no common features, and neither procedure calls the other, similarity is zero.
- The constant n controls normalization. For example, if n is 0, then all similarities are normalized between 0 and 1 (ignoring the *Linked* term). However, if n is large, then similarities are not normalized. The similarity of two objects with identical feature sets X would then be

  $$\frac{W(X) + k \times Linked(X,X)}{n + W(X)}$$

  showing that objects with large numbers of features could be more similar to other objects than could objects with few features.
- $Sim(A,B) = Sim(B,A)$

We are still left with the problem of how to assign weights to the features, and values to k, n and d. Ideally, the heavily-weighted features would be the names corresponding to hidden design decisions. However, we have no direct way of determining which identifiers should be hidden. Our early attempts gave all features the same weight, but found that frequently-occurring features dominated our classifier's performance, and rare features were ignored. More recently, we have been estimating the significance of a feature by its Shannon information content:

$$Weight(f) = -log(Probability(f))$$

where the probability of f is the fraction of all procedures that have feature f. This gives rarely-used indentifiers higher weights than frequently-used identifiers, in keeping with the idea that rare names are more likely to be hidden in modules than frequently-used ones.

To date, we have selected values for k, n, and d by trial and error. In a later section we will describe how we will compute them automatically in the future.

4. Re-Engineering Software Modularity

Re-engineering modularity includes both discovering the latent structure of existing code, and changing that structure to obtain better modularity. Arch supports three different (although overlapping) styles of re-engineering work:

- Incremental change: the software is already organized into high-quality modules. The architect wishes to identify individual weak points in the architecture, and repair them by making small changes.
- Moderate reorganization: although the sofware is already organized into modules, their quality is suspect. The architect wishes to reorganize the code into new modules, but with an eye to preserving whatever is still good from the old modularity.
- Radical (re)organization: Either the software has never been modularized, or the existing modules are useless. The architect wishes to organize the software without reference to any previous organization.

Arch supports these activities with two kinds of intelligent advice: *clustering* and *maverick analysis*. To simplify their description, we first introduce some terminology.

4.1. Clustering and Reclustering

These services organize procedures into a subsystem hierarchy, by hierarchical agglomerative clustering. They can be run in batch or interactively, and can use a pre-existing modularization to reduce the amount of human interaction needed. The architect uses the resulting categories as proposals for new modules.

The basic clustering algorithm is called *hierarchical, agglomerative clustering* [4]. It proceeds as follows:

```
Place each procedure in a group by itself
Repeat
    Identify the two most similar groups
    Combine them
until
    the existing groups are satisfactory
```

The resulting groups are then used to define the memberships of modules.

Similarity between groups is defined by a *group similarity measure*, of which we are experimenting with several. Termination may be based on computed criteria or on the architect's judgment.

Arch supports three variations on this algorithm:

- Batch clustering: the algorithm runs without supervision. Each *combine* operator makes a supergroup out of the two groups it is combining, creating a completely binary tree of groups. Then it heuristically eliminates "useless" groups to create a wider, shallower tree. (Cf. [5]) This variation has the weakness that mistakes early in the clustering process cause the final results to diverge widely from what the architect would like.
- Interactive, radical clustering: this algorithm repeatedly selects pairs of groups to combine but asks for confirmation from the architect before combining them. When accepting a pair to combine, the architect can either merge the two groups, or make a supergroup out of them. When the architect disagrees, the tool notes the pair of groups that was rejected, and never again tries to combine exactly that pair. Instead, it selects the next-most-similar pair of groups.
- Interactive reclustering: this method uses a previous classification to guide the clustering. It starts by noting the original module in which each procedure was located. Then, when it has selected two new groups to merge, it first checks whether their members were all in the same module previously. If so, it combines them without asking the architect. If they originated from different modules, the tool asks the architect, who can accept or reject the proposal, or set aside one or both of the groups for classification later.

4.2. Group Similarity Measures

An earlier version of the Arch clustering algorithm used a similarity measure based on information loss [5]. The same measure was used for individual procedures and for groups. However, it relied on characteristic properties of the group as a whole. This strategy is not always appropriate for combining groups that represent software modules. For example, sometimes only a handful of the procedures in a module share design information with the procedures in another module. Furthermore, the similarity measure we are using cannot be directly generalized to groups without computing group centroids, which would not be appropriate because of wide variation among the features of group members. Therefore, we are using aggregate measures based on pairwise comparisons of all members of two groups. The most promising of these is:

> *Single-link group similarity*: The similarity between two groups is the maximum similarity between any pair of group members (one from each group).

4.3. Good and Bad Neighbors

The following definitions, although not profound, are very useful for discussing comparisons among procedures and across module boundaries:

> *Subject*: A procedure that is being compared to several other procedures, for purposes of clustering or classification.
>
> *Neighbor*: A neighbor of a subject is any procedures with which it has at least one common feature.
>
> *Good Neighbor*: A subject's good neighbors are those neighbors that belong to the same module as it does.
>
> *Bad Neighbor*: A subject's bad neighbors are those that belong to different modules than it does.

4.4. Maverick Analysis

A *maverick* is a misplaced procedure. Arch detects potential mavericks by finding each procedure's most similar neighbors, and noticing which modules they belong to.

More formally,

> *Maverick*: A procedure for which the majority of its k nearest neighbors are bad neighbors.

We found that simply looking at the nearest neighbor was not sufficient, because sometimes it is the neighbor that is the maverick and not the subject itself. In this case, the second and third nearest neighbors will likely be in the same module, so setting k to 3 has proved satisfactory. However, there is nothing magic about the three nearest neighbors; one could also examine a larger neighborhood.

Since a maverick list can potentially be quite large, Arch prioritizes each mavericks by its similarity to its nearest bad neighbor, and presents them "worst first".

4.5. Tuning and User-Defined Features

When the architect disagrees with Arch's findings, she can tune the similarity function to remove the disagreement. One way of doing so is to "bias" some of the feature weights, supplying a multiplier to factor into the Shannon-derived weight. She can also provide biases for feature types, such as macro-names, type-names, "called-by" features, and so on.

Sometimes, the available features simply cannot adequately explain the architect's decision. The architect could, for example, put together two procedures that have no common features, because they use the same algorithm. In this case the architect can define a new feature name representing the abstract property that the procedures share, and assign that feature to all the procedures known to have that property.

5. Practical Results

We have used Arch to critique the modularity of five software systems. These informal experiments have taken place over an 18-month period, and so each used Arch at a somewhat different level of ability. However, together they show that Arch gives valuable advice in real maintenance situations.

5.1. Systems Studied

The systems were all written in C, ranging in size from 64-1100 procedures, spanning 7-75 modules. Types of systems studied included experimental code, rapid prototype, carefully crafted product, and old, heavily abused code. Some of the code was still undergoing maintenance, while other code was abandoned. In every case we were able to consult code experts to assess the value of Arch's analysis.

5.2. Maverick Experiments

Experiments on four systems, without tuning, flagged 10-30% of the procedures as mavericks. Of these, 20-50% were symptoms of real modularization errors in the code. Types of errors encountered included:

- A module that had been split into two without regard to information hiding.
- Modules that were "temporarily" split during development, and never put back together.
- Procedures that combined two very different kinds of functionality, each belonging to a different module. (These procedures were all written by the same rogue programmer!)
- An "unformed module": functionality scattered throughout the system that should have been collected into a single, new module.
- Pairs of procedures, in different modules, that performed exactly the same function on slightly different data structures.
- Programming bugs such as using the wrong global variable, or omitting a required procedure call.
- Code fragments that had been copied many times rather than making a procedure or macro out of them.
- A data abstraction that was violated by outside procedures accessing record fields directly.
- An incomplete data abstraction, missing some the access procedures needed to hide its implementation.
- Mistakes left over from a previous reorganization.
- Three small, closely related modules that should have been merged.
- Unused procedures.

5.3. Clustering

Clustering experiments have been more limited, because clustering systems containing large numbers of undetected mavericks is laborious. However, we have run the batch clustering algorithm on its own code. It contained 64 procedures, in seven modules. The clustering process required 56 choices (to reduce 64 groups to 7). We convened a panel of three architects, all familiar with the code, to review the outcome.

Forty of the choices the algorithm made combined groups whose members had originally been in the same module. Sixteen choices combined procedures orginating in different modules. *However*, the panel agreed that 10 of the 16 decisions were correct! That is, when Arch disagreed with the original modularity, it was correct more often than not.

Next, we tried to determine how much improvement could be obtained by hand tuning. The experimenter made modest adjustment to 8 of the feature weights (there were 80 features), such that rerunning the clustering algorithm achieved complete agreement between the tool and the panel of experts.

5.4. Hand Tuning

For one of the systems we studied, the architect investigated each maverick that Arch reported, and decided whether a code change to eliminate the maverick would be appropriate. He then took his analysis to the system's maintainer, who validated the results. Next, he hand-tuned Arch's similarity measure, by biasing the weights of some features, and adding some user-defined features, so as to minimize the number of mavericks that were not symptoms of harmful information sharing between modules. The results are summarized in the following table. The full study is reported separately [6].

Modules	27
Procedures	300

Mavericks	51	
	Before	After Tuning
Legitimate	18	13
False Alarm	33	10

Hand tuning consisted essentially of biasing 8 features, adding 5 user-defined features, and removing two anomalous procedures from the analysis.

5.5. Discussion

In every case, maverick analysis turned up significant problems with the modularity of existing software. Although one might question the ratio of identified mavericks to real problems, enough real problems were detected to justify further research and development. The clustering experiment, although limited, indicated that clustering is a viable means for proposing a substantial reorganization of a subsystem. Hand-tuning the similarity measure, and adding a few user-defined features, significantly improved agreement between Arch and the architect. However, the tuning process is difficult and unenlightening. The next section describes the automatic tuning method we are developing to replace hand-tuning.

6. Automatic Tuning

The automatic tuning method is based on the expectation that a procedure will be more similar to other procedures in the same module than to procedures in different modules. For each procedure in the system, it identifies the five (more generally, k) nearest good neighbors, and compares each of them to each of the procedure's bad neighbors. Its goal is to minimize the frequency with which a bad neighbor is more similar to a subject procedure than one of the subject's five nearest good neighbors. It achieves this goal by repeatedly examining each of the possible combinations of a subject, a good neighbor, and a bad neighbor, and adjusting the weights (by gradient descent) to bring the good neighbor closer and push the bad neighbor farther away. Implementation details are described in a companion paper [7].

6.1. Learning Performance

Although we eventually hope to solve maintenance problems on the grand scale described earlier, we have been using a rather modest-sized problem for our early experiments. The code is real: it is an early version of Arch's batch clustering tool. It comprises 64 procedures, grouped into seven modules. Membership in the modules is distributed as follows:

#	module
12	outputmgt
14	simwgts
10	attr
12	hac
7	node
4	objects
5	massage

The sample problem has two parts:

1. Identify classification errors in the given data, and remove the offending procedures for reclassification later.
2. Learn a similarity measure, by training on the remaining procedures, that can be used to classify the procedures by the nearest-neighbor rule.

The software is written in C. Extracting cross-references produced 152 distinct feature names. However, many of these features occurred in only one procedure each, and were therefore greatly increasing the size of the problem without ever contributing to the similarity of two procedures. Therefore, we eliminated all such singly-occurring features, leaving 95.

We expected the code to contain modularization errors, being a rapid prototype. However, we wanted to create a "clean" data set for test purposes. Therefore, by a combination of logical and heuristic methods we identified and examined several possible errors. However, we did not remove a procedure from the data set unless we were convinced that it was *both* a true modularization error *and* an object that our method would not be able to adapt to. Twelve procedures were thus removed, leaving 52.

When trained on the remaining 52 procedures, the gradient descent algorithm successfully found weights for which every procedure was in the same module as its nearest neighbor. Therefore, we say that Arch "learned" a similarity measure that was adequate to explain the module membership of every procedure in the training data. The computation took about 10 minutes on a Sun Microsystems SPARCstation 1+.

6.2. Generalization Performance

Learning performance, by itself, is not the primary goal. Instead, the objective is to use the tuned similarity measure to check the module assignment of procedures that were not in the training data.

To test the network's generalization, we constructed a jacknife test, in which the 52 procedures were divided into a training set and a test set, to determine how well the tuned similarity measure would predict the module membership of procedures that were not in the training data. The test consisted of 13 experi-

ments, each using 48 procedures for training and 4 for testing, such that each procedure was used for testing exactly once. Each procedure was tested by using the similarity function to identify its nearest neighbor, and predicting that the tested procedure belonged to that neighbor's module.

The results of the jacknife test are shown in the table below. Each row gives the number of procedures that were in that module, and how many of them were classified into each module during the jacknife test.

actual	predicted					
module	A	B	C	D	E	F
A 11	11					
B 11		10	1			
C 9			9			
D 8				8		
E 7					7	
F 2						2

Out of the 52 procedures in the data set, only one was misclassified!

Naturally, we will conduct more experiments, on other data sets, to better evaluate the usefulness of the tuned similarity measure If further experiments confirm these results, we will conclude that Arch's similarity function adequately measures similarity between procedures for the purpose of grouping procedures into modules.

6.3. Incremental Adaptation To The Architect

Next, we need a way to incorporate the automatic tuning method into the maverick analysis and clustering services. The difficulty lies in fitting the data too well. If Arch tunes the measure to precisely fit an existing system, the services will not suggest any changes!

Arch will overcome this problem by using only qualified data to tune the similarity function. The complete process will proceed something like this:

1. Arch identifies procedures that have no good neighbors at all, and presents them as mavericks. The architect must either agree that they are mavericks, set them aside for later analysis, or supply user-defined features that they share with other procedures in the same module.
2. Arch estimates the feature weights, without training, based on Shannon information content.
3. Arch creates an initial maverick list. Any procedure not appearing on the list is marked "qualified". Mavericks are marked "deferred".
4. Arch presents the mavericks to the architect, who may then either move a maverick, defer deciding about it, or mark it as qualified to remain in its present module.
5. Each time the architect qualifies a maverick to remain where it is, Arch retrains its similarity measure, as described below.

Retraining:

1. Construct the training set using only qualified procedures as subjects and neighbors. Deferred procedures are excluded.
2. Bias the training toward subjects that are architect-approved as opposed to machine-approved, reflecting the higher level of confidence placed on them.
3. Use the Shannon-based feature weights as the starting point for training.
4. Stop the training when all the procedures are correctly classified.
5. Update the maverick list, rechecking all the deferred mavericks, and highlighting any additions.
6. If, after a reasonable time, training does not achieve completely correct classification, Arch will report the failure to the architect and request a user-defined feature to resolve the problem.

The net effect of this incremental learning process will be that Arch starts with a naive view of similarity based on the information hiding principle and Shannon information content, then gradually modifies this view to fit the architect's judgements, bending "just enough" to agree with the architect. The architect will not have to manually approve the procedures that the tool already agrees are classified correctly; she only needs to examine those that seem to be mavericks. Whenever she rejects a maverick, Arch revises its own maverick criteria, by tuning weights, and removes from the list any procedures that are no longer mavericks by the revised criteria.

7. Conclusions

From experiments with the basic advisory services and with the training system, we conclude that Arch's similarity measure is a useful model for the way that programmers judge similarity between procedures during modularization, and that the advisory services are promising tools for re-engineering software modularity.

8. Acknowledgements

Monica Hutchins designed and implemented the current Arch prototype, including inventing some graphical browsing capabilities that are very helpful for investigating mavericks. Ronald Lange contributed many implementation ideas, as well as thoroughly testing Arch and using it for a thorough case study. H. G. Tempel and Shuyuan Chen also explored significant design and implementation issues.

References

1. David L. Parnas, "Information Distribution Aspects of Design Methodology", *Information Processing 71*, North-Holland Publishing Company, 1972.

2. David L. Parnas, "On the Criteria To Be Used In Decomposing Systems Into Modules", Tech. report, Computer Science Department, Carnegie-Mellon University, August 1971.

3. Amos Tversky, "Features of Similarity", *Psychological Review*, Vol. 84, No. 4, July 1977.

4. Yoelle S. Maarek, *Using Structural Information for Managing Very Large Software Systems,* PhD dissertation, Technion -- Israel Institute of Technology, January 1989.

5. Robert W. Schwanke and Michael A. Platoff, "Cross References Are Features", *Second International Workshop on Software Configuration Management*, ACM Press, November 1989, also available as SigPlan Notices, November 1989

6. Ronald Lange and Robert W. Schwanke, "Software Architeture Analysis: A Case Study", *Third International Workshop on Software Configuration Management*, ACM Press, June 1991.

7. Robert W. Schwanke and Stephen José Hanson, "Using Neural Networks to Modularize Software", Tech. report, Siemens Corporate Research, Inc., September 1990, Submitted to *Machine Learning*