The work we report here addresses the fundamental issue of how to measure the quality of a given modularization of the software. Note that modularization quality is *not synonymous* with modularization correctness. Obviously, after software has been modularized and the API of each of the modules published, the correctness can be established by checking function call dependencies at compile time and at run time. If all inter-module function calls are routed through the published API functions, the modularization is correct. As a theoretical extreme, retaining all of the software in a single monolithic module is a correct modularization though it is not an acceptable solution. On the other hand, the quality of modularization has more to do with partitioning software into more maintainable (and more easily extendible) modules on the basis of the cohesiveness of the service provided by each module. Ideally, while containing all of the major functions that directly contribute to a specific service vis-à-vis the other modules, each module would also contain all of the ancillary functions and data if they are only needed in that module. Capturing these "cohesive of service" and "ancillary support" criteria into a metric is an important goal of our research. The work that we report here is a step in that direction.

More specifically, we present in this work a set of metrics that measure in different ways the interactions between the different modules of a software system. It is important to realize that metrics that only analyze inter-module interactions cannot exist in isolation from other metrics that measure the quality of a given partitioning of the code. To explain this point, it is not very useful to partition a software system containing of a couple of million lines of code into two modules, each consisting of a million lines of code, and justify the two large modules purely on the basis of function call routing through the published APIs for the two modules. Each module would still be much too large from the standpoint of code maintenance and code extension. The module interaction metrics must therefore come with a sibling set of metrics that record other desirable properties of the code. The metrics we present in Section 4, while focusing primarily on module interactions, also include other necessary measures of the quality of a given partitioning of code.

In the rest of this paper, we briefly survey in Section 2 the past literature relevant to the goals of our work. Section 3 discusses the relationship between previous work and our current work. We present the metrics in Section 4. In Section 5, we apply the metrics to what is generally considered to be a well-organized body of code - the HTTPD server code from Apache.

2. Previous Work

Research in code modularization and metrics that test the effectiveness of a given modularization date back to the early seminal work of Parnas [17]. Historically speaking, he was one of the first to focus on the notion of a module in software. According to him, a software module should be characterized by a design decision that it hides from all other modules and that module interface should not reveal any inner workings of the module.

A descendent of [17] is the contribution by Schwanke [3]. He sought to create optimized code partitions by characterizing the partitions with a set of numerically measurable features and then trying to find a global best solution in the feature space. One of his primary features is a measure of "information sharing" between the functions that are meant to be in the same module. Information sharing is derived form the commonality of names used in the functions and the functional purpose of those names. For example, if two functions share names that imply that the functions are sharing data objects, then the two functions belong together in the same module. Another feature used by Schwanke is derived from function-call dependencies. If a function A calls function B, then both A and B presumably belong to the same module. That brings us to a brief description of the various efforts that have been undertaken over the years to partition code purely on the basis of function-call dependencies or criteria that are based primarily on such dependencies.

In code modularization driven by function-call dependencies, one first constructs a function-dependency graph or a file-dependency graph (that is derived from function dependencies), the nodes of the graph being the individual functions (or files) and the edges representing either the function-call dependencies or some other inter-function or inter-file attributes. The nodes of the graph may then be clustered together on the basis of certain edge properties or criteria derived therefrom. Each cluster thus discovered becomes a suggestion for a module. The research contributions by Fahme and Holt [11], Mitchell et. al [7][10][12], etc., are representative of this line of work. When graph partitions are optimized with respect to some integrated measure of edge attributes, the optimum partitions would generally correspond to the eigenvectors of a matrix representation of the edge attributes [10]. Optimized partitioning of a graph in this manner can also be achieved by other modern tools such as those based on

genetic algorithms (GA), simulated annealing, etc. The contribution by Zhang and Jacobson [15] is also relevant to our work. Their wok is focused primarily on the development of tools for characterizing aspects and for the discovery of aspects in legacy code. An aspect represents a feature of the software that applies at a more global level to all the units of the software. For example, one can talk about aspects related to security, logging, tracing, etc. Aspects that can be recognized by statement-level "cross cutting" structure may be discovered through lexical pattern matching. Zhang and Jacobson's goal is to modularize code on the basis of the discovered aspects. If successful, this effort would repackage legacy code in the form of an aspect-oriented architecture.

Yet another previous contribution relevant to our work is the work of Rysselberghe and Demeyer [16] on how to gain evolutionary insights into large software by analyzing the change histories. It seems that history logs could aid in the remodularization of code that started out being well-partitioned but then gradually devolved into an unstructured body of code.

Specifically with regard to the issue of metrics, we must mention the work of Oman and Hagemeister[1][2] They have applied themselves to the formulation of metrics that assess the maintainability of code In particular, this work presents an empirical formula for calculating a maintainability index (MI) that is based on Halstead and Cyclomatic complexity measures derived from source code analysis. The utility of these metrics has been evaluated by comparing the MI with human perception of maintainability. This work is important to us since a well-modularized body of code will also be easy to maintain. So, one would think that we could use the same metrics to assess the goodness of our modularization at least from the standpoint of the effort required for code maintenance.

3. Relationship of Previous Work to Our Present Goals

We believe that many of these previous approaches suffer from shortcomings with regard to the goals we have in mind. The approaches that carry out software partitioning purely on the basis of function call dependencies (or file-dependencies that are derived from function-call dependencies) are obviously not suitable for meeting our goals. Function call dependencies are semantically orthogonal to the groupings on the basis of cohesiveness of service. To elaborate, in code partitioning on the basis of function-call dependencies, if a function A calls a function B, then both A and B must belong together in the same module. But using function call dependencies as the sole basis for modularization runs counter to the very spirit of what is meant by modules in modern code writing. Modules pull together functions not because they call one another, but because they serve similar purposes with respect to the rest of the software. For example, the number of intra-module function calls in the java.util module (referred to as a package in the Java parlance) is minimal. The main reason for why the functions in the java.util module belong together is because they provide very similar services to the rest of the software.

The approaches that try to cluster together functions based on what [3] refers to as information sharing are also deficient because they do not scale well to large software.

Our mention of prior work would be incomplete without mentioning the system modernization effort of the Object Management Group. OMG has started an initiative to define a standard [16] for application metadata description that is meant to facilitate interoperability of modernization tools. To the extent the modernization tools under consideration by OMG include those intended for legacy code, the OMG effort is relevant to the work being reported here.

4. Module Interaction and Other Related Metrics

Modern software engineering dictates that large software be organized along the following lines:

1. The software system should consist of a set of modules where each module is a collection of data structures and functions that together offer a well-defined service. In other words, the structures used for representing knowledge and any associated functions in the same module should cohere on the basis of similarity-of-service as opposed to on the basis of function call dependencies.
2. The modules should interact with one another only through the exposed API functions. With regard to code maintenance, this is desirable for isolating faults and rectifying them quickly.
3. Whenever feasible, the modules should be organized in a hierarchical manner in a set of layers. A layer should only be aware of the layers below it (that is, function calls are only made to the lower layers) and should not be aware of the layers above it. A layer

can be thought of as horizontal partitioning of the system.

4. Modules should be independently testable and releasable. The impact of a single change should typically stay confined to a module and should minimally impact other modules.

These considerations have led us to formulate the following metrics for measuring the quality of module interactions. Of the four considerations listed above, the testability related consideration is more complex and depends on what tools are brought to bear and what protocols are used for testing code. Therefore, for now, we will ignore this consideration. Given the importance of this issue, we certainly plan to take up testability related issues in a future research contribution. Additionally, as was stated in the introduction, metrics that focus solely on the interactions between the modules cannot exist in isolation from the metrics that measure other qualities of code modularization. Therefore, the set of metrics shown below includes those that are needed to simultaneously report these other attributes.

M1: Module Interaction Index:

Suppose that a module *M* has n functions f_1 ,.., f_n and suppose that there are N_{int} number of calls made to a function *f* from other functions internal to module *M* and N_{ext} number of calls made to *f* from other functions external to module *M*. Also suppose that there are *m* modules in the system. Then the following ratio measures the utility of a function with regard to its usefulness to other modules:

$$MII(f) = \frac{Next}{Next + N\,\text{int}}$$

An average measure of this utility over all the functions in a given module is:

$$MII(M) = \frac{1}{n}\sum_{i=1}^{n} MII(fi).$$

This measure is averaged over all the modules to yield the following metric that applies to the entire software:

$$MII(System) = = \frac{1}{m}\sum_{i=1}^{m} MII(M_i)$$

A well designed module often exposes a limited set of API functions through which other modules interact. These API functions are generally a small percentage of all the functions that constitutes the module. They represent the service that the module has to offer. Since these API functions are meant to be used by other modules, the internal functions of a module typically do not call API functions of the module. Therefore, for an API function *f* of a well designed module, MII(f) → 1. By the same argument, a non-API function should not have any external calls at all, i.e. its MII should be 0. Since the majority of the functions in a module should ideally be non-API, an average of all MIIs at the module level would be low for a well designed module.

M2: Non-API Function Closedness Index:

As one is experimenting with different ways of partitioning code, one is bound to go through stages when the code may be considered to be semi-modularized. That is, the code partitions may be in a state in which most inter-module function-call traffic is routed through the published API's for each of the modules, but there remain some residual inter-module function calls outside the API's. For all those functions in a module that have not yet been declared to be API functions, we calculate a metric that we call the "Non-API Function Closedness Index". This metric represented by C(f) is measured by the following formula. *C(f)*=

$$\frac{\#\text{times } f \text{ is called by other functions in the same module}}{\text{Total number of times } f \text{ is called}}$$

Assuming that a module has p non-API functions, we average *C(f)* as follows over a module:

$$C(M) = \frac{1}{p}\sum_{i=1}^{p} C(fi)$$

= 0 if there are no non-API functions

Averaging *C(M)* over all the modules *m1* having non-zero *C(M)* yields:

$$C(\text{System}) = \frac{1}{m1}\sum_{i=1}^{m1} C(M_i)$$

Since a well designed module does not expose the non-API functions to the external world, $C(f_{na})$ → 1 for a non-API function f_{na}. Since the average is taken over non-API functions only, *C(M)* for a well designed module should also be close to 1.

M3: API Function Usage Index:

This index determines what fraction of the API functions exposed by a module is being used by the other modules. When a big monolithic module M presents a large and versatile collection of API functions offering many different services, any one of the other modules may not need all of its services. That is, any single other module may end up using only a small part of the API. The intent of this index is

to *discourage* the formation of such large monolithic modules offering services of disparate nature and *encourage* modules that offer specific functionalities. Currently, we have formulated this index as:

APIU(M) =

$$\frac{\text{Max (Number of API functions called by any other module Mj)}}{\text{Total number of API functions of M}}$$

= 0 if M has no API functions

The System level *APIU(System)* can be computed as

$$APIU(System) = \frac{1}{m2}\sum_{i=1}^{m2} C(M_i)$$

assuming that there are *m2* modules with non-zero *APIU* values.

M4: Module Size Uniformity Index:
Within reason, of course, all modules should be roughly equal in size. A strong deviation from this constraint of uniformity in size would generally be indicative of poor modularization. Therefore, a modularization algorithm needs to possess a bias towards making the modules as equal in size as possible, subject to the fulfillment of other modularization constraints. This constraint can be expressed in terms of the average value μ and the standard deviation σ associated with the module sizes. We can define a Module Size Uniformity Index (MSUI) as the ratio;

$$MSUI(System) = \frac{|\mu - \sigma|}{\mu}$$

Obviously, the closer this index is to 1 the greater the uniformity of the module sizes. The formula for this metric assumes that the standard deviation is not comparable to the average size value.

M5: Module Size Boundedness Index:
While the previous metric will "push" a modularization algorithm to make the modules as nearly uniform in size as possible, it could still result in modules that are too large. Many practitioners of the art and science of code modularization recommend that, ideally, no module should significantly exceed some recommended particular size that is expressed in terms of the number of lines of code. Assuming that this "magic number" is N, if we want a modularization algorithm to try to honor this number, we can do so by first defining an average deviation in module length from the magic length by

$$\partial_{av} = \frac{1}{m}\sum |N_i - \quad N|$$

where N_i is the number of lines in the i^{th} module and *m* is the number of modules. As a fraction of the largest of either the magic length, N, or any of the module lengths, this deviation is re-expressed as

$\delta_{fav} = \frac{\delta_{av}}{\max(N, N_i)}$. The metric may now be defined as :

$$MSBI(System) = 1 - \delta_{fav}$$

5. Experiments with Well-Organized Code

In order to validate the efficacy of the metrics, we applied them to the roughly 350,000 lines of the Apache web server code written in C. This software is highly regarded in industry and academia for its quality and robustness. The software is well-organized into a directory structure on the basis of the services provided by the various sub-directories and files. Functionally distinct sub-directories carry mnemonic names that correspond to the corresponding services. That makes it relatively easy to identify the various modules comprising the system..

Before the metrics are applied to the Apache software, the code was first analyzed by the open-source tool Sourcenav [18] that yielded a database containing the associations between the function definitions and the corresponding file names and also the function-call dependency information. Subsequently we ran a set of tools written in Perl and Java to extract the above mentioned metrics. The approach has been depicted in Figure 1.

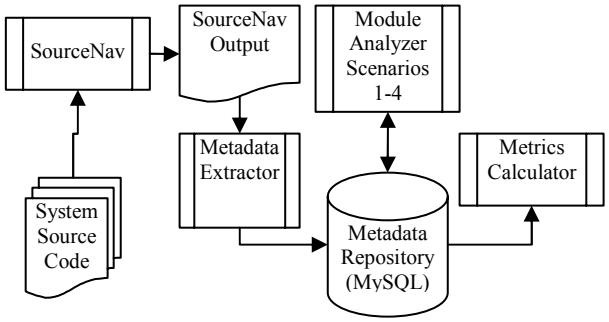

Figure 1 Schematic diagram of Metric Extraction Process

Shown below is a sample of metadata collected by the Metadata Extractor:

System Metadata	Value
Total files	689
Total LOC	370,000
LOC without Comments	336000
# Function call cross reference obtained from static analysis	7213
Total leaf directories	110

To establish the usefulness of our metrics, we not only need to show that the numbers look good for well-written code; we also need to demonstrate that the numbers yielded by the metrics become progressively worse as the code becomes increasingly disorganized. In order to make such a demonstration, we created four different modularized versions of the software. These versions of the software correspond to the following scenerios:

Scenario 1 (Human): We considered the directory leaf nodes of the directory hierarchy of the original source code to be the most fine-grained functional modules. All the files (and functions within) inside a leaf level directory were considered to belong to a single module --- the module corresponding to the directory itself. In this manner, all leaf level directories formed the module set for the software. We call this module set the Developer Generated Module Set.

Scenario 2 (Random): Functions were assigned randomly to modules in such a manner that we ended up with the same number of modules as in the developer generated module set. We call this the Randomly Generated Module Set.

Scenario 3 (Semi-Random): Instead of functions, we now assign files randomly to modules. All the functions in the same file are considered to belong to the module to which the file was assigned. Note that in this case whereas a file-to-module assignment is completely random, the same cannot be said of the function-to-module assignment. We call this as Semi-Random Module Set.

Scenario 4 (Concept Driven): We used a heuristic approach to assign functions to modules. Recall that our long-term goal is to be able to examine millions of lines of code and cluster together data structures and functions on the basis of similarity of purpose. It will obviously be the case that we will have a set of heuristics that will "measure" the purpose of a data structure or a function. The heuristics would provide us with a first cut at the modularization of the code; the metrics would measure the quality of the modularization thus obtained; and iterative algorithms (whose discussion is beyond the scope of this paper) would then refine the modularization until certain criteria are met with regard to the quality of modularization. As a step in that direction, the work reported in this paper used a simple heuristic for function clustering — clustering driven by keywords denoting the service performed by the function. We may think of these keywords as domain concepts that provide a clue as to the service performed by the function. The domain concepts are matched with the full directory pathnames to the files containing the functions and the function names to provide a weight value as to whether the function corresponds to the purpose corresponding to the domain concepts. Here is a pseudo-code description of how the domain concepts are matched with directory pathnames and the filenames:

GetSimilarity(filename `f`, concept string `c`) {
1. `PathArray` = get the directory path of `f`.
2. for each directory name `d` ε `PathArray` {
   i. Find out how close `c` matches with `d`. Let this number be υ [0..1].
   ii. Calculate the weight ω of `d`.
   /** We have used a function that assigns high ω to `d` if `d` appears in the middle of the path. The idea is that the leftmost and the rightmost directories are too coarse-grained and too fine-grained, respectively, and should play no role in matching d with c. The directories that appear in the middle are good candidates for matching.
   **/
   iii. The similarity S_d is = ω *υ;
   }
3. Return `S` = Max (S_d);
}

AssignModule(filename `f`, concept-list `C`, threshold τ) {
Let S_{max} = Max(S_c) where S_c= *GetSimilarity*(`f`, `c`) and c_{max} be the concept for which the similarity is maximum;
If $S_{max} > \tau$ {
Assign c_{max} to be module for `f`;
Return c_{max} ;
}
}

In each of the four scenarios, a key challenge was as to how to identify the API functions vis-à-vis the non-API functions in a given module.. For the purpose of this particular experiment with the Apache code, we identify API functions with the heuristic that an ideal API

function is mostly called by functions belonging to other modules. Similarly, an ideal non-API function is never exposed, i.e. it is never called by functions from other modules. According to this heuristic, in each modularized version, a module function was declared to be an API function or a non-API function depending on whether or not it was called by other modules. In several cases a function is called externally as well as internally. In such cases we have considered the function as API as well as non-API depending on the metric we are calculating.

Experimental Results

The following table shows the metrics calculated from the application metadata by the Metrics Calculator tool in Figure 1 for each of the four scenarios mentioned previously. The abbreviations used for the names of the metrics in the top row are the same as in the formulas of the previous section.

Experi-ment #	#Mod-ules	MII	C	APIU	MSUI	MSBI
Developer Generated	110	0.228	0.874	0.8392	0.5978	0.6914
Randomly Generated	110	0.985	0.445	0.1586	0.752	0.2235
Semi-randomly generated	110	0.333	0.833	0.7269	0.1362	0.3352
Concept Driven	40	0.196	0.915	0.8109	0.1276	0.7129

In the rest of this section, we will briefly discuss the nature of the values calculated for the metrics in each of the four scenarios. The abbreviations used for the metric names are the same as in the column headings of the table.

MII: The values obtained for this metric are in agreement with our designation of functions as API functions and non-API functions. For the developer generated and concept driven scenarios, the number of functions declared as API is 224 and 185, respectively, out of a total 1684 functions as per our heuristic for designating API functions.. In the random scenario almost all of 1684 functions (1674 to be exact) became API functions due to the random allocation of functions to modules. Thus, the MII values for human and concept driven scenarios become small when we take an average over all the functions of a module (since most of the functions are non-API and their MIIs are 0).

C: As evident from the table, this metric is consistently high for the human and concept-driven scenarios and low for random and semi-random scenarios. This once again validates our observation on the characteristics of C as described in Section 4.

APIU: This metric shows high values for the human and concept-driven scenarios and low values for random & semi-random scenarios. The random assignment of functions to modules, causing arbitrary declaration of API functions, results in disparate-natured services offered by a module. Clearly such a scenario is discouraged (low value).

MSUI: The MSUI metric shows high values for random and semi-random scenarios. Since the random number generation follows a uniform distribution, the functions are randomly but uniformly distributed across several generated modules in random and semi-random scenarios. This in turn results in smaller standard deviations for these two scenerios, causing high values for MSUI.

6. Conclusion

Despite all the attention that code modularization has received over the last couple of decades, one has yet to see a set of metrics that can be used for the partitioning of legacy code on the basis of the services provided by the functions and the data structures in the modules. The metrics that have been proposed in the past, while useful for generating measures of software quality with regard to maintainability, extendibility, portability, modularity on the basis of function-call and data dependencies, etc., appear to be unsuitable for modularization on the basis of the similarity of services rendered. The metrics proposed in this paper are a step in the direction of rectifying this deficiency.

The experiments described in this paper seek to establish whether our proposed metrics can distinguish a well modularized system from a randomly modularized one. In order to simulate the condition of gradual degradation of modularity, we started with a well modularized open source system as packaged by the developers of the software and created a semi-randomly modularized version (intermediate state of disorganization) and a randomly modularized version (state of complete disarray) of the software. The metrics when calculated for each version confirmed the degradation trend. We also wanted to determine whether the values yielded by the metrics for a services based modularization are close to the values for the developer-generated modularization. As an initial step

towards this direction, we simulated a services-based modularization by clustering functions on the basis of keywords, with each keyword denoting a service. The metric values in this case were close to the ones obtained for developer-generated modularization. Although still preliminary, our experimental results do indicate that the proposed metrics are of the sort that will be needed by the code modularization algorithms of the future.

7. References

[1] Oman, P. & Hagemeister, J. "Metrics for Assessing a Software System's Maintainability," 337-344. *Conference on Software Maintenance 1992*. Orlando, FL, November 9-12, 1992.

[2] Oman, P. & Hagemeister, J. "Constructing and Testing of Polynomials Predicting Software Maintainability." Journal of Systems and Software 24, 3 (March 1994): 251-266

[3] Schwanke, Robert. W. "An intelligent tool for re-engineering software modularity", in *Proceedings of the lSth International Conference on Software Engineering* (May 1991), pp. 83-92

[4] Keith Bennett, "Legacy Systems: Coping with Success", *IEEE Software*, Jan 1995, pp 19-23.

[5] M. P. Ward and Keith Bennett, "Formal Methods for Legacy Systems", Journal of Software Maintenance: Research and Practice,Vol 7, no 3, May-June 1995, pp 203-219.

[6] T.A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization", Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97)

[7] S Mancoridis, Brian S Mitchell, C Rorres, Y Chen, ER, "Using automatic clustering to produce high-level system organizations of source code", Proc. 6th Intl. Workshop on Program Comprehension, 1998

[8] C Lindig, G Snelting, A Softwaretechnologie, "Assessing modular structure of legacy code based on mathematical concept analysis",

[9] C. J. Alpert, A. B. Kahng, and S. Yao, "Spectral Partitioning with Multiple Eigenvectors," Discrete Appl. Math., Vol. 90, pp. 3-26, 1999

[10] D Doval, S Mancoridis, B Mitchell, "Automatic clustering of software systems using a genetic algorithm", *Proceedings of Software Technology and Engineering Practice*, 1999.

[11] H. Fahmy and R.C. Holt. "Software Architecture Transformations", Proceedings of the International Conference on Software Ma intenance, San Jose, Oct. 2000 pp 88-96

[12] Brian S. Mitchell, Spiros Mancoridis and Martin Traverso, "Search Based Reverse Engineering", SEKE 2002, July 15-19 2002. Italy

[13] D Vecchio, "Legacy Modernization Provides Applications for Tomorrow", ID Number: M-19-3671, 5th March 2003.

[14] C Zhang, HA Jacobsen, "A Prism for Research in S oftware Modularization through Aspect Mining", Technical report, Middleware Systems Research Group, Sept 2003

[15] F.V. Rysselberghe and S Demeyer, "Studying Software Evolution Information by Visualizing the Change History", 20th IEEE International Conference on Software Maintenance, Sep 2004, pp 328-337

[16] White paper from OMG: "Modernization Scenarios: Mapping the KDM to Modernization Initiatives", Draft #3-9/9/2004

[17] Parnas, D. L. "On the Criteria to be used in Decomposing Systems into Modules", Communications of the ACM, 15(12) pp 1053-1058, 1972.

[18] Source-Navigator™ 5.4.1 Source Code Analysis Tool , 2003, URL:http://sourcenav.sourceforge.net