

# Verifying Design Modularity, Hierarchy, and Interaction Locality Using Data Clustering Techniques

Liguo Yu  
Computer Science and Informatics  
Indiana University South Bend  
1700 Mishawaka Ave. P.O. Box 7111  
South Bend, IN 46634, USA  
ligyu@iusb.edu

Srini Ramaswamy  
Computer Science Department  
University of Arkansas at Little Rock  
2801 S. University Avenue  
Little Rock, AR 72204, USA  
srini@acm.org

## ABSTRACT

Modularity, hierarchy, and interaction locality are general approaches to reducing the complexity of any large system. A widely used principle in achieving these goals in designing software systems is striving for high cohesion within a module and low coupling between modules. However, this principle has difficulties in practice. Because a hierarchical system structure often consists of several layers, it is difficult to decide at what layer an interaction should be considered as cohesion, and at what layer an interaction should be considered as coupling. In this paper, we do not differentiate cohesion and coupling, but use a general term *interaction* to represent the dependencies between software modules. We propose a method to verify the design modularity, hierarchy, and interaction locality of a software system. This approach is based on the component interactions gathered from certain design level artifacts, such as UML diagrams. Data clustering technique is then used to group software components according to the degree of interactions between them. To show how to use this approach, we apply it to Parna's KWIC object-oriented design example, in which sequence diagram is used to derive the degree of component interactions.

## Categories and Subject Descriptors

D.2.2.d [Modules and interfaces].

## General Terms

Design.

## Keywords

Modularity, hierarchy, interaction locality, clustering.

## 1. INTRODUCTION

It has been observed that most of the complex systems in the world, whether structures of atoms or stellar galaxies, are modular

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACMSE 2007, March 23–24, 2007, Winston-Salem, North Carolina, USA. ©Copyright 2007 ACM 978-1-59593-629-5/07/0003...\$5.00.

and hierarchically structured. A large system may consist of subsystems, which consists of subsystems, and so on, through many layers. The interactions between subsystems tend to decrease as we go upward in the hierarchy. This is called interaction locality [1]. Interaction locality can minimize the energy for the system to operate. Coupled with interaction locality, modularity and hierarchy form the basis for system development and evolution.

In a software system, design modularity and hierarchy means the decomposition of the software system into different layers of components in order to separate concern and reduce system complexity; the interaction locality is expressed via a widely accepted design principle: high cohesion within a component and low coupling between components [2]. For the purposes of this work we define a module as the lowest-level decomposition of the system and define a component as a high order aggregation of modules.

Consider an ideal system that consists of two subsystems Sb1 and Sb2, which contain components C1, C2, and C3, C4 respectively, which in turn contain modules, m1 through m8. Figure 1 depicts the modular structure and Figure 2 depicts the hierarchical structure. The interaction locality is shown in Figure 1 with the arrows between modules, components, and subsystems. The thickness of the arrows indicates the degree of interactions. It matches the general design principle: high interactions exist between lower level modules and low interactions exist between higher level components and subsystems in the hierarchical structure shown in Figure 2.

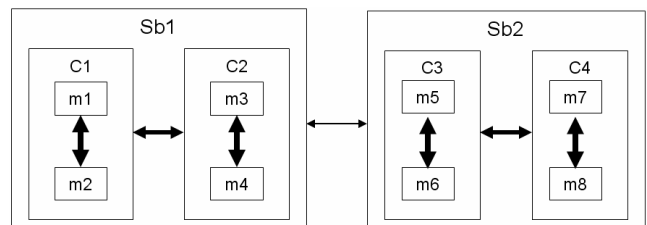
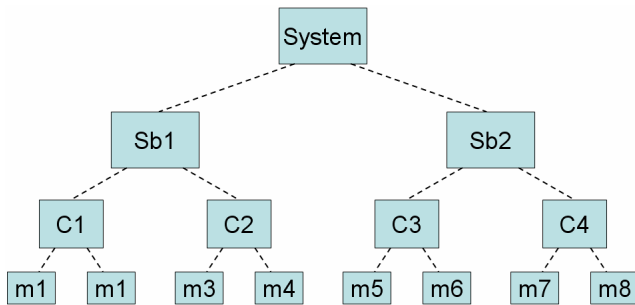


Figure 1. The modular structure and interaction locality of an ideal system.



**Figure 2. The hierarchy structure of the system shown in Figure 1.**

Modularity, hierarchy, and interaction locality are widely followed design principles in practice. However, for large software systems that contain many layers, there is still a lack of systemic methods to qualitatively verify whether a design follows these design principles. This is due to the lack of a formal mathematical representation of interaction locality. One commonly raised question is, for a system with many layers, should interactions between components be considered cohesion or coupling? Or in other words, what is the degree of interaction that should exist between two components?

Some research has been carried out in this field. For example, Darcy et al. [3] performed an empirical study and found that both coupling and cohesion should be considered jointly instead of independently when designing, implementing, and maintaining software. However, to our knowledge, there has been no formal approach to combining coupling and cohesion in software design in order to reduce the system complexity.

In this paper, we propose a method to measure the degree of interactions between primitive components (modules, classes) based on design level artifact, such as UML diagram. A data clustering technique is then used to group software components and verify the design modularity and hierarchy.

The following of the paper is organized as follows. Section 2 reviews modularity, hierarchy, and interaction locality. Section 3 presents our approach to verifying design modularity, hierarchy, and interaction locality. Section 4 contains a case study on Parna's KWIC object-oriented design. Our conclusions are presented in Section 5.

## 2. MODULARITY, HIERARCHY, AND INTERACTION LOCALITY

In software systems, modularity is a widely accepted design principle [4]. To achieve high understandability, maintainability, and reusability, a software system should be able to be decomposed to manageable components, which can further be decomposed to modules. For example, structured software development modularizes functions into modules, object-oriented software development modularizes abstract data into classes, and aspect-oriented software development modularizes cross-cutting concerns into aspects [5].

On the other hand, to cope with the complexity of a very large system, it is not sufficient to just divide the system into simple pieces because the pieces themselves will either be too numerous

or too large. A hierarchical modular structure is a common solution [6].

Both modularity and hierarchy are associated with interaction locality. The modular and hierarchical structure should be designed and arranged based on the interaction locality principle: the interactions between components should decrease as we go upward in the design hierarchy. Coupling has been used as the terminology to describe the degree of interaction between two software components (classes, modules, packages, or the like).

There are many different types of coupling [7], [8]. All of them can be shown to fall into one of the following four types: parameter coupling, external coupling, inheritance coupling, and common coupling [2], [9]. This categorization was presented in the context of object-oriented software systems. However, in view of the fact that the constructs of structured software are a proper subset of those of object-oriented software, coupling in structured software systems can be represented using parameter coupling, and common coupling; inheritance coupling is specific to object-oriented software systems. The definitions of these types of coupling are listed in Table 1. In object-oriented software, a class is the basic manageable unit; while in structured software, a module is the basic manageable unit. In Table 1, we refer to both these basic units, in general, as primitive components.

**Table 1. Definitions of various kinds of coupling [2], [9]**

Name	Definition
Parameter Coupling	Two primitive components have parameter coupling if one module invokes method of another module via parameter passing.
External Coupling	Two primitive components have external coupling if they access the same external medium including external files.
Inheritance Coupling	Two primitive components have inheritance coupling if one module is a descendant of another module.
Common Coupling	Two primitive components have common coupling if they access the same global variable.

A good software system should have high cohesion within each component and low coupling between components. Strong coupling means a high degree of dependency between software components. Common coupling is considered to be a strong form of coupling because it induces strong dependencies between software components, making software components difficult to understand, maintain, and reuse. Inheritance coupling is also considered as a strong form of coupling [10], [11] in the context of software maintenance and white-box reuse, because any changes to a base class will affect all its derived classes. Parameter coupling is usually considered as a weak form of coupling.

Table 1 provides a general guideline for software design. However, it does not automatically lead to a hierarchy structure that follows interaction locality. The general design principle is to reduce interactions between all components while the interaction locality principle says, for large systems, we can not reduce the number of interactions, but we can rearrange the modular and hierarchical structure to make strong interactions locally and weak interaction at higher layer. Therefore, there is a need for a systematic approach to verifying the design modularity, hierarchy, and interaction locality as shown in Figures 1 and 2.

### 3. VERIFYING MODULARITY, HIERARCHY, and INTERACTION LOCALITY

Our approach to verifying design modularity, hierarchy, and interaction locality consists of two steps. In the first step, we need to measure and represent the degree of interactions between primitive components. In the second step, we will apply a data clustering technique to group software components.

#### 3.1 Step One: Interaction Representation

To represent the degree of interactions between primitive components, we introduce a terminology, *interaction frequency (IF)*.

**Definition 3.1** For two primitive components  $i$  and  $j$ , *interaction frequency* represents the degree of interactions between  $i$  and  $j$  based on one or more types of coupling between them. It is represented as  $IF_{i,j}$ .

There are many representations of the degree of interactions between two primitive components. One approach is to use the definitions in Table 1. Consider two primitive components  $i$  and  $j$ . Suppose the number of parameter coupling, external coupling, inheritance coupling, and common coupling between  $i$  and  $j$  are measured as  $pc$ ,  $ec$ ,  $ic$ , and  $cc$ , the interaction frequency between  $i$  and  $j$  can be represented as:

$$IF_{i,j} = a_1 \bullet pc + a_2 \bullet ec + a_3 \bullet ic + a_4 \bullet cc \quad (1)$$

As discussed in Section 2, different type of coupling indicates different degree of interactions. Therefore, in Equation 1, different coupling is assigned different weight parameters ( $a_1$  through  $a_4$ ) in measuring interaction frequency. Generally speaking, the values of  $a_1$  through  $a_4$  should be in an increasing order. However, the quantitative determination of the values of weight parameters should be based on the empirical evidence of the effects of different coupling on software quality, software maintenance, and software reuse.

Equation 1 is a general representation of primitive component interaction. However, some information in Table 1, such as common coupling, is only available after the system is implemented. To verify design level modularity, hierarchy, and interaction locality, we need to derive interaction frequency as early as possible.

One approach to deriving design level interaction frequency is to use software design artifacts, such as UML diagrams, data-flow diagrams, and entity-relationship diagrams. For example, considering an object-oriented design, the most commonly used

couplings are inheritance coupling and data coupling, which can be derived from class and interaction diagrams, respectively. Therefore, the design level interactions between classes can be refined as

$$IF_{i,j} = a_1 \bullet pc + a_3 \bullet ic \quad (2)$$

Again, we remark here that  $a_1$  and  $a_3$  are relative values to balance the different effects of parameter coupling ( $pc$ ) and inheritance coupling ( $ic$ ). To normalize Equation 2, we assume  $a_1$  to be 1. The object-oriented design level interaction frequency is further refined as follows, where  $\beta = a_3/a_1$ .

$$IF_{i,j} = pc + \beta \bullet ic \quad (3)$$

Definition 3.1 gives the representation of degree of interaction between two primitive components. What we are interested is the large system that contains many primitive components. Therefore, we define *interaction matrix IM*.

**Definition 3.2** For a system that contains  $n$  primitive components, the degree of interactions between these  $n$  primitive components is represented as an  $n \times n$  *interaction matrix (IM)*, in which item at position  $(i, j)$  is the interaction frequency between component  $i$  and component  $j$ .

$$IM = \begin{bmatrix} IF_{1,1} & IF_{1,2} & \dots & IF_{1,n-1} & IF_{1,n} \\ IF_{2,1} & IF_{2,2} & \dots & IF_{2,n-1} & IF_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ IF_{n-1,1} & IF_{n-1,2} & \dots & IF_{n-1,n-1} & IF_{n-1,n} \\ IF_{n,1} & IF_{n,2} & \dots & IF_{n,n-1} & IF_{n,n} \end{bmatrix} \quad (4)$$

It worth noting here that (1)  $IF_{k,k}$  represents the degree of interactions within a single primitive component, it should be set to a maximum value, say  $\infty$ ; (2)  $IM$  is a symmetric matrix, in which  $IF_{i,j} = IF_{j,i}$ . In this study, we do not differentiate the directions of coupling. For example, if one primitive component invokes a method of another primitive component, we say there is parameter coupling between them and ignore the dependency directions [12].

#### 3.2 Step Two: Clustering

Clustering [13] is a data mining technique to group items into clusters according to their similarities, differences, or distances. In this research, we use clustering to group software components according to the interaction frequencies between them and form a hierarchical structure so that components at a lower level have larger interaction frequency while components at a higher level have smaller interaction frequency.

There are many different kinds of clustering methods. Here, we briefly review the most commonly used hierarchical clustering method [14].

Given a set of  $n$  primitive components to be clustered, and an  $n \times n$  interaction matrix, the basic procedure of hierarchical clustering is:

1. Start by assigning each component to a cluster. Initially, we have  $n$  clusters, each containing just one component. Let the interaction frequency between the clusters be the same as the interaction frequency between the components they contain.
2. Find the pair of clusters that have largest interaction frequency and merge them into a single cluster.
3. Compute interaction frequency between the new cluster and each of the other (old) clusters.

Repeat steps 2 and 3 until all components are clustered into a single cluster of size  $n$ . Step 3 can be done in different ways: in single-linkage clustering, the interaction frequency between one cluster and another cluster is considered to be equal to the greatest interaction frequency from any member of one cluster to any member of the other cluster; in complete-linkage clustering, the interaction frequency between one cluster and another cluster is considered to be equal to the smallest interaction frequency from any member of one cluster to any member of the other cluster; in average-linkage clustering, the interaction frequency between one cluster and another cluster is considered to be equal to the average interaction frequency from any member of one cluster to any member of the other cluster.

After clustering, we can generate a dendrogram that shows the ideal composition of the system based on the interactions between primitive components. This composition can be used to compare

with the system decomposition (architecture) and verify the three design principles: modularity, hierarchy, and interaction locality.

#### 4. KWIC CASE STUDY

In this section, we perform a case study to show how to use our proposed approach on Parnas' KWIC problem.

“The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be circularly shifted by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.” [15].

One solution to KWIC problem is an object-oriented design that contains six classes (primitive components): KWIC, Input, Output, CircularShifter, Alphabetizer, and LineStorage [16]. Note here, there are no inheritance relations among these classes ( $ic=0$ ). Interaction frequency (Equation 3) is refined as:

$$IF_{i,j} = pc \tag{5}$$

Therefore, in this case study, only parameter coupling will be used to derive interaction frequencies between classes. Figure 3 shows the sequence diagram in the object-oriented design of KWIC, which will be used to measure the parameter coupling ( $pc$ ) between classes.

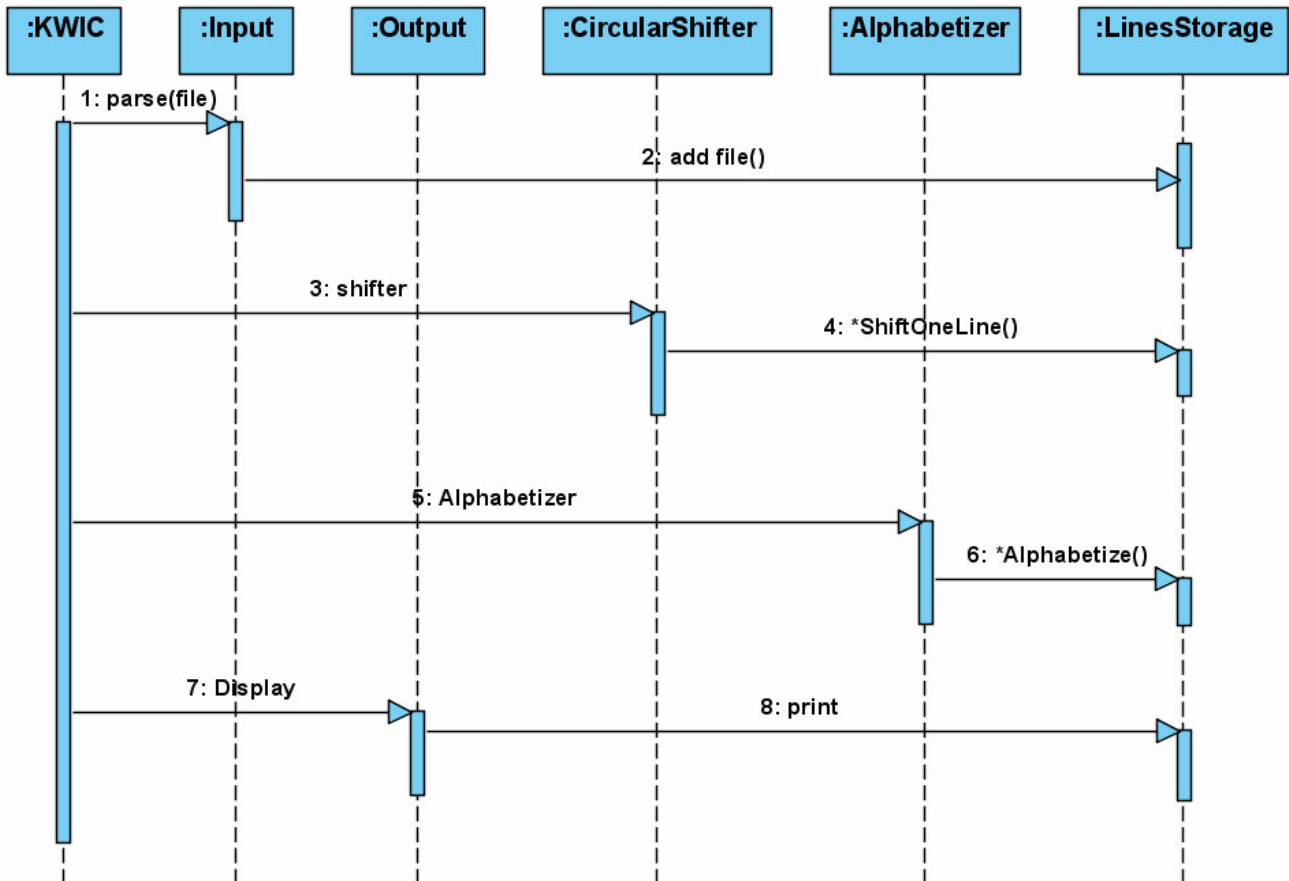
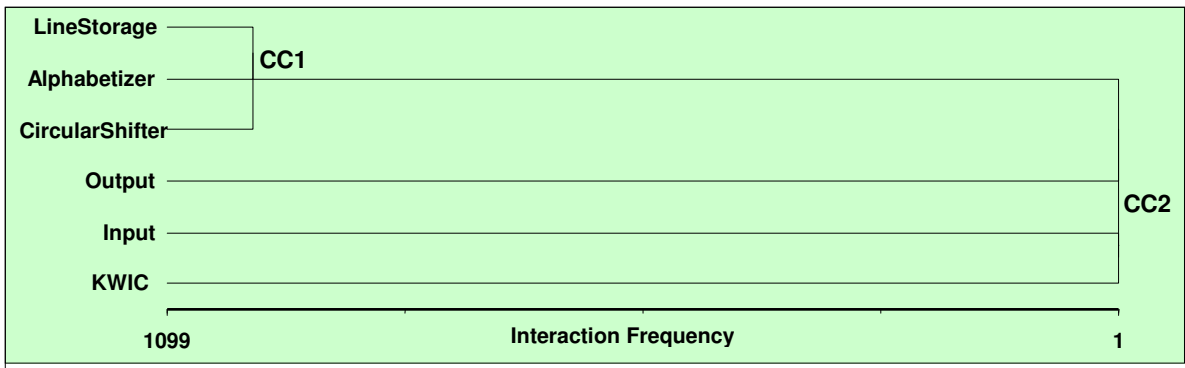


Figure 3. The sequence diagram of KWIC object-oriented design.

**Table 2. The Interaction Matrix of KWIC classes**

	KWIC	Input	Output	Alphabetizer	CircularShifter	LineStorage
KWIC	$\infty$	1	1	1	1	0
Input	1	$\infty$	0	0	0	1
Output	1	0	$\infty$	0	0	1
Alphabetizer	1	0	0	$\infty$	0	999
CircularShifter	1	0	0	0	$\infty$	999
LineStorage	0	1	1	999	999	$\infty$



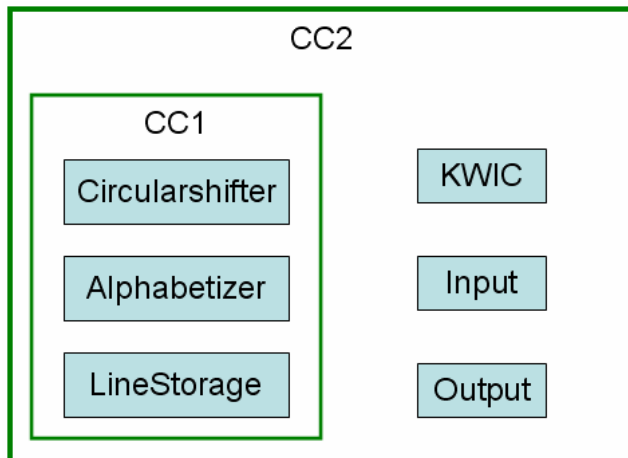
**Figure 4. The dendrogram of clustering KWIC classes.**

In deriving parameter coupling (pc), we applied the following rules: (1) a method call from one class (Ci) to another class (Cj) is counted as one parameter coupling between classes Ci and Cj; (2) multiple iterative invocations marked with \* is counted with a special value 999. Table 2 shows the Interaction Matrix of these six classes.

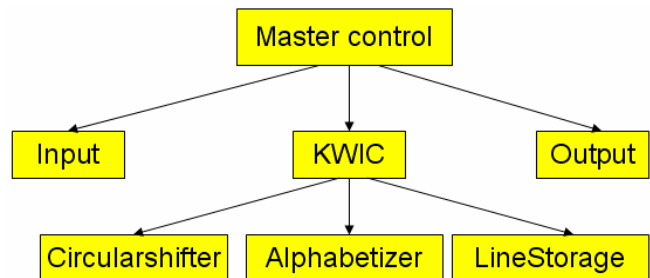
Next, we apply single-linkage hierarchical clustering on the interaction matrix shown in Table 2. Figure 4 shows the dendrogram.

Figure 4 shows that three primitive components, CircularShifter, Alphabetizer, and LineStorage form the first level composite component CC1, because these classes have the largest interaction frequency. CC1, Input, Output, and KWIC form the second level composite component CC2.

Figure 5 is the system decomposition of KWIC program based on the dendrogram shown in Figure 4. This architecture follows the three design principles: modularity, hierarchy, and interaction locality.



**Figure 5. The Modular structure of KWIC object-oriented design based on the clustering of interaction matrix.**



**Figure 6. The hierarchical structure of KWIC provided in [16].**

After deriving Figure 5, we can use it to compare with the system decomposition to verify the architecture designs. Figure 6 shows the object-oriented architecture provided in [16]. It can be seen that Figure 5 and Figure 6 have many similarities: composite component CC2 in Figure 5 is the Master control in Figure 6;

component Input, KWIC, and output are in the same level; component Circularshifter, Alphabetizer, and LineStorage are in the same level. The only difference between Figure 5 and Figure 6 is there is a composite component CC1 in Figure 5, while in Figure 6, this component is not shown.

It is to be noted here that we are not using Figure 5 directly as the design architecture of the target system because (1) we have only used the sequence diagram to derive the interaction matrix. A complete and accurate interaction matrix should be based on all related design artifacts, such as entity-relationship diagram, class diagram, and so on; (2) there are many other related factors (for example, component reuse) more than just basic component interactions that need to be considered to design the complete architecture. Therefore, our proposed approach in this paper is for verifying architecture design.

## 5. CONCLUSIONS

In this paper, we presented a formalized approach to verifying the modularity, hierarchy, and interaction locality of a software design. This approach is based on the measurement of interaction frequencies between primitive components and the hierarchical clustering method. We used KWIC as a case study to show how this approach can be applied.

One important issue in using this approach is to determine the interaction frequency between primitive components. Our future research will study how to derive this information from design level artifacts, such as class diagrams. We will also study how to quantitatively represent the interaction frequency of different types of coupling.

## 6. ACKNOWLEDGEMENTS

This work is supported partly by the Indiana University South Bend Faculty Research Grant (2007).

## 7. REFERENCES

- [1] Simon, H. A., "The Architecture of Complexity", The Sciences of the Artificial, Cambridge, MA, MIT Press, 1969: 192–229.
- [2] Offutt, J., Harrold, M. J., and Kolte, P., "A Software Metric System for Module Coupling," Journal of System and Software, 1993, 20(3): 295–308.
- [3] Darcy, D. P., Kemerer, C. F., Slaughter, S. A., and Tomayko, J. E., "The Structural Complexity of Software: An Experimental Test", IEEE Transactions on Software Engineering, 2005, 31(11): 982–995.
- [4] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B., "The Structure and Value of Modularity in Software Design", Proceedings of the 8<sup>th</sup> European Software Engineering Conference, Vienna, Austria, 2001: 99–108.
- [5] Griswold, W. G., Sullivan, K. J., Song, Y., Shonle, M. M., Tewari, N., Cai, Y., and Rajan, H., "Modular Software Design with Crosscutting Interfaces", IEEE Software, 2006, 23(1): 51–60.
- [6] Blume, M. and Appel, A. W., "Hierarchical Modularity", ACM Transactions on Programming Language and System, 1999, 21(4): 813–847.
- [7] Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, 1974, 13(2): 115–139.
- [8] Page-Jones, M., The Practical Guide to Structured Systems Design. New York: Yourdon Press, 1980.
- [9] Offutt, J., Abdurazik, A., and Schach, S. R., "Coupling-Based Maintenance Metrics for Object-Oriented Software," submitted for publication, 2006.
- [10] Bruegge, B. and Dutoit, A. H., Object-Oriented Software Engineering Using UML, Patterns, and Java, Pearson Prentice Hall, Upper Saddle River, NJ, 2004.
- [11] Hassoun, Y., Johnson, R., and Counsell, S., "A Dynamic Runtime Coupling Metric for Meta-Level Architectures," Proceedings of the 8<sup>th</sup> European Conference on Software Maintenance and Reengineering, Tampere, Finland, March 24–26, 2004: 339–346.
- [12] L. Yu, "Understanding Component Co-evolution with a Study on Linux", *Empirical Software Engineering*, 2006, to appear.
- [13] Jain, A. K., Murty, M. N., and Flynn, P. J. "Data Clustering: a Review", *ACM Computing Surveys*, 1999, 31(3): 264–323.
- [14] Johnson, S. C., "Hierarchical Clustering Schemes", *Psychometrika*, 1967, 2: 241–254.
- [15] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 1972, 15: 1053–1058.
- [16] Garlan, D. and Shaw, M., "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Co., 1993.