The Structural Complexity of Software: An Experimental Test

David P. Darcy, Chris F. Kemerer, *Member*, *IEEE Computer Society*,
Sandra A. Slaughter, and James E. Tomayko, *Member*, *IEEE Computer Society*

Abstract—This research examines the structural complexity of software and, specifically, the potential interaction of the two dominant dimensions of structural complexity, coupling and cohesion. Analysis based on an information processing view of developer cognition results in a theoretically driven model with cohesion as a moderator for a main effect of coupling on effort. An empirical test of the model was devised in a software maintenance context utilizing both procedural and object-oriented tasks, with professional software engineers as participants. The results support the model in that there was a significant interaction effect between coupling and cohesion on effort, even though there was no main effect for either coupling or cohesion. The implication of this result is that, when designing, implementing, and maintaining software to control complexity, both coupling and cohesion should be considered jointly, instead of independently. By providing guidance on structuring software for software professionals and researchers, these results enable software to continue as the solution of choice for a wider range of richer, more complex problems.

Index Terms—Software complexity, software structure, Wood's model of task complexity, coupling, cohesion, experiment, software maintenance, software metrics, cognition, procedural programming, object-oriented programming.

1 Introduction

Few activities are as complex as the effective design, implementation, and maintenance of software. Since the early criticisms of the difficulties in managing so-called "spaghetti code," software engineering (SE) has attempted to use software measures and models to reduce complexity and, thereby, achieve other goals, such as greater productivity. However, complexity cannot always be reduced. Problems, especially practically important ones, have an inherent level of complexity and it can be argued that it is desirable for organizations to continue to attack problems of increasing complexity. Solving a problem with software tends to add its own complexity beyond that of the problem itself. Unfortunately, increases in problem complexity may lead to supralinear increases in software complexity and increases in software complexity may lead to supralinear impacts on managerial outcomes of interest, such as increasing the effort to design, implement, and maintain software and reducing its quality [7].

Software complexity has multiple facets, including algorithmic complexity [24] and structural complexity [1]. The structural complexity of a program has been defined as "the organization of program elements within a program" [21, p. 191]). This paper focuses on structural complexity because dealing with structural complexity primarily expends intellectual resources, whereas algorithmic complexity primarily consumes machine resources. Thanks to four decades of Moore's law, the availability of machine resources per unit cost continues to grow exponentially. On the other hand, intellectual resources per unit cost are becoming scarcer; this comparison may explain why *Business Week*'s industry review found productivity in the software industry to have slightly declined during the 1990s [22].

The objective of this paper is to understand how software design decisions affect the structural complexity of software. This is important because variations in the structural complexity of software can cause changes in managerial factors of interest, such as effort and quality [12].

Measures of structural complexity are *internal* attributes, i.e., they are specific to the software code artifact. They are distinct from *external* attributes that are measures of more direct managerial interest, such as effort and productivity. Prior research has contributed models and measures of both internal software quality attributes and external attributes such as comprehensibility and maintainability (e.g., [18]). Although the relationships between internal and external attributes can be intuitive, e.g., more complex code will require greater effort to maintain, the precise functional form of those relationships can be less clear and is the subject of intense practical and research concern. The consideration of multiple theoretical perspectives, including human cognition, provides a solid foundation upon which to derive an integrative model relating internal and external attributes of software quality. And, a well-designed empirical study serves to clarify and strengthen the observed relationships. This approach is consistent with Kitchenham et al. [33] who state "Other scientific disciplines

- *D.P. Darcy is with the Robert H. Smith School of Business, University of Maryland, 4351 Van Munching Hall, College Park, MD 20742. E-mail: ddarcy@rhsmith.umd.edu.*
- *C.F. Kemerer is with the Joseph M. Katz Graduate School of Business, University of Pittsburgh, 278A Mervis Hall, Pittsburgh, PA 15260. E-mail: ckemerer@katz.pitt.edu.*
- *S.A. Slaughter is with the David A. Tepper School of Business, Carnegie Mellon University, 354 Posner Hall, Pittsburgh, PA 15213. E-mail: sandras@andrew.cmu.edu.*
- *J.E. Tomayko is with the School of Computer Science, Carnegie Mellon University, 4624 Wean Hall, Pittsburgh, PA 15213. E-mail: jet@cs.cmu.edu.*

Manuscript received 12 Feb. 2005; revised 12 July 2005; accepted 26 Sept. 2005; published online 1 Dec. 2005.
Recommended for acceptance by A. Mili.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0024-0205.

search for a deep hypothesis that arises from an underlying theory, so that testing the hypothesis tests the validity of the theory. For example, rather than merely documenting a relationship between cyclomatic number and faults found, we can use theories of cognition and problem-solving to help us understand the effects of complexity on software" (p. 724). This approach and the consequent results represent a useful approach for SE practice and research.

Any study of past research on the structural complexity of software leads to the conclusion that coupling and cohesion are fundamental underlying dimensions. Coupling and cohesion are understood here in the sense of "measurable concepts" as defined in Annex A of ISO standard 15939, rather than as specific measures. The current research leads to the recognition of the interdependent nature of coupling and cohesion, particularly with regard to their effect on project outcomes such as effort. An experimental study with professional software engineers reinforces the finding that coupling and cohesion are interdependent and should be considered jointly with regard to the structural complexity of software.

This paper makes a number of research and practical contributions. The critical role of the concepts of coupling and cohesion in structuring software is theoretically established. This assists in moving from a general notion of software structure to an understanding of specific factors of structural complexity. Complexity analysis typically proceeds by considering coupling and cohesion independently. Based on theoretical and empirical evidence, this research argues that they must be considered *together* when designing software in order to effectively control its structural complexity. By studying software at higher levels, the effect of design decisions across the entire life cycle can be more easily recognized and rectified. And, it is at the design stage that these results are potentially the most significant, as, the larger the system, the more difficult complexity choices can be. And, in addition, the earlier in the life cycle the intervention is, the more flexible those choices are. Finally, this paper is not about proposing new measures, but, rather, focusing attention more directly on coupling and cohesion as essential and interrelated indicators of the underlying dimensions of structural complexity.

The remainder of this paper is organized as follows: The theoretical threads that underpin this paper are outlined in Section 2. These threads are integrated into a research model and predictions based on the model are outlined in Section 3. Section 4 describes an empirical research design to test the model. Section 5 presents the results of an experiment involving 17 professional software engineers as subjects. Section 6 discusses the results, some limitations, and concludes the paper by summarizing its contributions and suggesting a number of future research directions.

2 Conceptual Background

It is widely believed that software complexity cannot be described using a single dimension [47]. The search for a single, all encompassing dimension has been likened to the "search for the Holy Grail" [17]. To find such a dimension would be like trying to gauge the volume of a box by its length, rather than a combination of length, breadth, and height. Early attempts to determine the key dimensions of software complexity have included identifying factors in single or small numbers based on observing programmers in the field (e.g., [48]) or adapting, refining, and/or improving existing factors (e.g., [11]). However, although useful, neither of these two approaches enables a direct answer to what the important software complexity characteristics are. In fact, the SE literature is replete with calls for theoretical guidance on this issue (e.g., see [3], [9], [23], [27], [42]).

In order to develop a theoretically-based model for the research, two theoretical perspectives are employed. Wood's task complexity model is examined for its insights into task complexity in general, and software complexity in particular [54]. Wood's model is generally representative of task complexity approaches [10], but it is more closely studied in this paper because it has already been found to be useful in a software maintenance context [5]. Wood's model is also valuable to this research as it describes the essence of the structural complexity of a general solution to a given problem; as such, its contribution to unearthing the fundamental dimensions of the structural complexity of software can be a significant asset to progress in the field.

Once the dimensions of structural complexity are identified, a natural managerial question is what are the relationships among the chosen dimensions to effort? To answer this question, theories of human cognition are used. Understanding cognition in general is difficult and it is clear that a general understanding does not provide an extensive guide to understanding the specifics of software maintenance cognition [10]. As a result, beyond straightforward notions, such as the use of a modular approach that has become the norm in software engineering, there is very little parsimony in modeling software design, programming, and maintenance activities. Despite the difficulty in conceptualizing the cognitive processes for software design, implementation, and maintenance, it is widely believed that cognition must be considered to add credibility to the identification of complexity dimensions, given cognition's role as the likely major source of variation in software design, development, and maintenance [6], [32], [51].

2.1 Wood's Task Complexity Model

Wood's Task Complexity Model considers tasks to have three essential concepts: products, required acts, and information cues [54]. Products are "entities created or produced by behaviors, which can be observed and described independently of the behaviors or acts that produce them" [54, p. 64]. An act is "the pattern of behaviors with some identifiable purpose or direction" [54, p. 65]. Information cues are "pieces of information about the attributes of stimulus objects upon which an individual can base the judgments he or she is required to make during the performance of a task" [54, p. 65]. Using these concepts, three sources of task complexity are defined: *component*, *coordinative*, and *dynamic*.

Component complexity is defined as a "function of the number of distinct acts that need to be executed in the performance of the task and the number of distinct information cues that must be processed in the performance of those acts" [54, p. 66]. *Coordinative complexity* covers the "nature of relationships between task inputs and task

TABLE 1
Coupling and Cohesion in the Procedural Paradigm

Citation	Measures	Empirical Data
[27]	4 coupling measures (data bindings)	2 medium systems
[17]	5 coupling measures	None reported
[1]	Combined measure of coupling and cohesion	1 medium Unix utility
[52]	1 cohesion and 5 coupling measures	Textbook system
[46]	8 measures for coupling and cohesion	1 medium system
[41]	11 measures of coupling	2 small & 3 medium systems
[35]	7 cohesion measures	None reported
[8]	1 measure for functional cohesion	5 sample procedures

products" [54, p. 68]. The form, the strength, and the sequencing of the relationships are all considered to be aspects of coordinative complexity. *Dynamic complexity* refers to the "changes in the states of the world which have an effect on the relationships between tasks and products" [54, p. 71]. Over the task completion time, parameter values are nonstationary. The performance of some act or the input of a particular information cue can cause ripple effects throughout the rest of the task. The predictability of the effects can also play a role in dynamic task complexity. Total task complexity is then a function of the three types of task complexity.

2.2 The Information Processing Perspective on Cognition

The information processing view of cognition is described in terms of a very familiar analogy—the computer. A computer has primary storage (e.g., RAM) and secondary storage (e.g., hard disks, CD-ROM, etc). In order for a computer to "think," it must hold all the (program) instructions and all the relevant data (input and, ultimately, output) in primary storage. This primary-secondary storage mechanism is very similar to the way many cognitive psychologists believe people's minds work [2]. It is believed that people have a primary storage area called short-term memory (STM) and that they must have all of the data and instructions in STM before they can "think." Accessing a process or data in secondary storage (called long term memory or LTM) is generally slower compared to accessing something in STM, although LTM generally has more capacity. What is believed different between the computer and the mind is that, for the mind, an item or, rather, a unit of storage is not as homogenous as a byte for computer storage. The relevant units of mental storage are chunks and what constitutes a chunk seems likely to vary by person and domain [2].

A fundamental approach to improving software development has been to modularize the design by splitting the implementation of the solution into parts [15], [41]. Program parts can sometimes be termed modules. In turn, modules often consist of data structures and one or more procedures/functions. The term *part* can also be used to mean just a single procedure/function. In the object-oriented (OO) programming paradigm, the parts are usually thought of as classes, rather than modules or procedures/functions. This paper will use the term "*program units*" to refer generically to modules, procedures, functions, and classes.

2.3 Coupling and Cohesion

The literature in SE has suggested coupling and cohesion as two essential dimensions of software complexity [18]. In the following sections, the research on coupling and cohesion measurement is briefly reviewed from the procedural and object-oriented paradigms.

2.3.1 Coupling and Cohesion in the Procedural Programming Paradigm

The source for much of the original thinking on coupling and cohesion is Stevens et al. [48]. Some of these authors went on to write books on the subject of structured design [39], [55] and both books devote a chapter each to coupling and to cohesion. Although Myers [39] made no reference to any form of cognition when discussing coupling and cohesion, Yourdon and Constantine [55] included a chapter titled "Human Information Processing and Program Simplicity" (pp. 67-83). The chapter discussed people's limited capacity for information processing, citing Miller's work on the magic number of "7 ± 2." The rest of the chapter built on that notion, laying much of the groundwork for later work on coupling and cohesion measure development and validation. It is commonly cited (and noted in the abstract of [48]) that the ideas expressed in the original paper and the later books came from nearly 10 years of observing programmers by Constantine. Though later criticized for a lack of theory and rigor (e.g., [34], [40]), the work of [48] represents the roots of the first paradigm in coupling and cohesion; many later works on coupling and cohesion start from that point. "Relatedness" of program parts is how coupling is widely understood [26]. Cohesion was originally described as binding—"Binding is the measure of the cohesiveness of a module" [48, p. 121]. Cohesion subsequently replaced binding as the term used for this type of software complexity [8]. The original definition and subsequent treatments of this form of complexity rely on the notion of "togetherness" of processing elements within a module to define cohesion. For both coupling and cohesion, ordinal scales were provided.

It has often proven difficult to distinguish coupling and cohesion results from other software measure results. Nevertheless, some empirical work exists, primarily in the procedural programming paradigm. Some subsequent work on measure development for coupling and cohesion is listed chronologically in Table 1. Most of the work is observational or conceptual in nature, with only a small amount of work done to empirically validate various measures. Progress in

this area has been very gradual, taking a decade of observation to outline the original coupling and cohesion measures and more than another decade to refine the levels and specify measures for each level.

2.3.2 Coupling and Cohesion in the OO Programming Paradigm

The OO programming paradigm has emerged as a widely considered alternative to the procedural programming paradigm. Because the original coupling and cohesion research was couched in procedural terms, the OO paradigm needed to specifically consider the differences between the two paradigms [16]. Early approaches to specification of coupling and cohesion, including the original works, came under significant criticism due to lack of theory [3].

The newer OO programming paradigm and the push for more theoretically-based measures led to the development of the Chidamber and Kemerer suite (hereafter, CK suite) of six software measures for the OO programming paradigm [13]. The suite contained one cohesion measure (LCOM) and two coupling measures (CBO and RFC). The essential characteristics of coupling and cohesion (relatedness and togetherness, respectively) are retained by the CK conceptualization of coupling and cohesion.

A recent review of OO software measurement [43] listed dozens of cohesion and coupling measures for OO. Many of the measures listed are based on the CK suite and the suite has also been extensively empirically validated. For example, empirical work on C++ and Java classes has shown an impact of a subset of the CK suite on defects after controlling for size [49]. Other work has set out to specifically consider the usefulness of coupling and cohesion as structural complexity dimensions and predictors of design quality. For example, in [50], the author used the notion of a concept to achieve lower coupling and higher cohesion to better restructure software modules. As another example, [6] makes it clear that there are only certain instances where the effort of restructuring is worth the cost of performing the restructuring.

3 Conceptual Design Choices and the Structural Complexity of Software

This paper derives the dimensions of structural complexity based on Wood's theory of task complexity. If software activities such as design are considered to be tasks, then software tasks are subject to analysis by the models and methods of task analysis. Wood's model is a general theory that has been applied to analyzing tasks, including software (e.g., [6]).

Structural complexity was defined earlier as "the organization of program elements within a program" [21, p. 191]. The definition highlights that structural complexity is a function of the solution (i.e., the software), rather than an aspect of the problem. Building software provides many opportunities to make design, implementation, and maintenance decisions that lead to different structures for the same problem. The definition also highlights that complexity must be evaluated for the whole program. While particular program units can be the subject of focus during complexity analyses, improving the complexity of certain program units may increase the complexity of other program units, leading to a neutral impact across the entire program.

While the definition cited above provides some guidance, it is a relatively crude representation of the intuition on structural complexity and includes very little material upon which to select units of analysis. For example, it is unclear as to what elements the definition is referring. Thus, to articulate the notion of the structural complexity of software, it is necessary to look further. Section 3.1 explores the impacts of structural complexity in the context of software maintenance, the focus of this research. Section 3.2 examines how the general notions of structural complexity from Wood's model translate into the structural complexity of software and illustrates the significance of coupling and cohesion. Section 3.3 addresses the interaction effect of coupling and cohesion on effort.

3.1 Structural Complexity and Maintenance Effort

It is emphasized that effort is meant in the context of operating "on" the software rather than "with" the software, i.e., effort is an issue for software designers rather than for users of the software. The structure of a program is an accumulation of all of the design, implementation, and maintenance choices made to that point in the program's evolution. Although there is no "best" structure for a solution to a given problem, there are definitely better and worse structures—the distinction is typically made largely as a matter of intuition combined with experience. The reason for examining structural complexity is to more clearly outline what might constitute better and worse structures, thereby aiding in creating structurally optimal programs that are most easily comprehended.

In this study, effort is considered in the context of software maintenance tasks. Software maintenance is the third and last major stage in the life of software, following design and implementation. Maintenance activities are generally classified as corrective, adaptive, preventative, and perfective [30], [46]. In economic terms, maintenance is recognized as far more important than design or implementation, representing as much as 80 percent of resources consumed [5], [28], [51]. The problem of how to optimally implement a given design has not yet been satisfactorily solved [52], [53]. Therefore, it should not be surprising that the trend is for comprehension models to be generally found in the domain of design and coding, rather than maintenance. Though not explicitly done, some of the programming comprehension models are sufficiently generalizable so that they can also be used to understand and explain maintenance cognition (e.g., [52]). There is also research into cognitive processes during maintenance. For example, there exist controlled experimental studies (e.g., [20], [45], [51]), although some of these studies have been criticized for using inexperienced programmers as subjects [27]. Finally, there also exists evidence of the link between complexity and maintenance in the commercial arena (e.g., [4], [19]).

3.2 Coupling and Cohesion as Dimensions of the Structural Complexity of Software

It is necessary to draw parallels between Wood's model to the specific domain of this paper, software. Consider the three atomistic and essential components of Wood's model: acts, products, and information cues. In software terms, a distinct act is a program unit (e.g., a function or an object). Each program unit is unique and can be accessed or called from other points in the program (overall task). Products are equivalent to the outputs of a program unit. Information cues could be, to use Bieman and Ott's terminology, data tokens [8]. The three sources of task complexity, component, coordinative, and dynamic, also have parallels in the software domain.

Component complexity is a function of each of the distinct acts and the information cues that must be attended to and processed for each act. It is valuable to consider how this mechanism was conceived. Wood's first approach on component complexity was to simply calculate the number of distinct acts [54]. But, as each act is different, this was not a sufficiently encompassing method of calculating component complexity. Each act needed to be considered individually. Hence, each act was weighted by the number of information cues processed by that act before that act was added to the total for component complexity [54].

In the software domain, an act is a program unit and information cues are the data tokens specific to that program unit. Cohesion is the concept in software that considers the complexity of a program unit as a function of the elements directly encompassed by that program unit and is a measure of the "togetherness" of a single program unit or "act." In other words, at the level of a single program unit, cohesion is effectively equivalent to component complexity. Cohesion is an inverse indicator of complexity in that the more cohesive a program unit is, the less complex that program unit is considered to be and the lower the effort required to maintain it. In addition, Wood's model points toward a mechanism to "scale up" from considering cohesion at the level of a single program unit to consideration at the level of the whole program. Thus, Hypothesis 1:

Hypothesis 1. *For the more highly cohesive programs, maintenance effort is lower.*

Coordinative complexity is a function of the precedence relations for an act, summed across all of the acts required to complete the task. Coupling measures the "relatedness" of a program unit to other program units. That same relatedness is effectively equivalent to precedence at the level of a single program unit as completion of an act requires the resolution of all links for that act. For a given program unit, a greater relatedness to other program units increases the complexity of that program unit and, consequently, the effort required to maintain it. Again, Wood's analysis allows generalizing from consideration of a single program unit to the whole program. Thus, Hypothesis 2:

Hypothesis 2. *For the more highly coupled programs, maintenance effort is higher.*

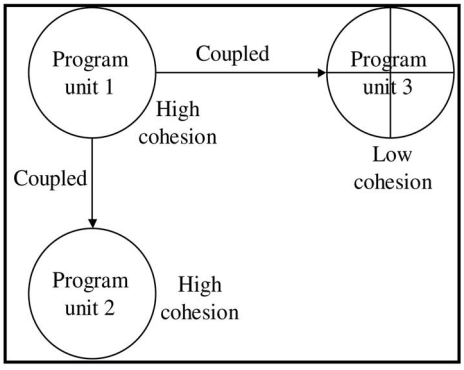

Fig. 1. Interaction of coupling and cohesion.

Drawing on Wood's model, it is apparent that part of a task should be able to stand by itself as much as possible; the lower the degree of coupling of the part, the better. The reverse is true for cohesion; for a given entity, all of its components should be as related as possible—it should not be possible to divide up the entity much further. Lower coupling and higher cohesion are taken as design and implementation goals [8]. For both concepts, it is recognized that it is not desirable or feasible to completely remove all coupling or to have complete cohesion. Therefore, the practical challenge is in selecting an acceptable level for either measure.

Implementation will add complexity above and beyond that of the complexity inherent in the design. At implementation, the decisions made impact both coupling and cohesion. And, for a given piece of software, its structural complexity can be significantly altered based on the placement of a code segment, but the size of that software, in terms of the number of lines of code, is hardly affected. In other words, coupling and cohesion tap different complexity factors other than size, a point that is further developed in the next section.

3.3 The Relationship between Coupling and Cohesion

Given that Wood's task complexity model is applied to identify the two complexity dimensions of coupling and cohesion, the second research question focuses on how to resolve conflicts and to make trade-offs between these two dimensions. It is at this point that insights from the cognition literature are drawn, particularly with regard to the information processing perspective on cognition and its fundamental concepts of STM, LTM, and chunks. Little empirical evidence and few theoretical propositions exist to guide resolution of the issue. However, the notion of limited cognitive resources clearly implies a joint effect for coupling and cohesion on effort. In Wood's model, the three components of task complexity are presented as independent from one another and as having an additive effect on total task complexity. However, Wood later recognizes possible interdependencies between the components in his model [54].

Complexity grows at a much faster rate for code that increases coupling as compared to code that reduces cohesion. Consider the cases illustrated by Fig. 1.

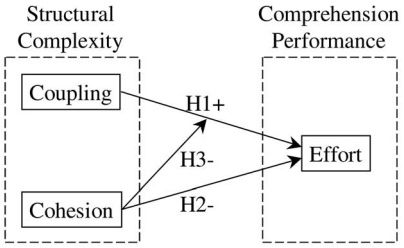

Fig. 2. Research model for the structural complexity of software.

If a programmer needs to comprehend program unit 1, then the programmer must also have some understanding of the program units to which program unit 1 is coupled. In the simplest case, program unit 1 would not be coupled to any of the other program units. In that case, the programmer need only comprehend a single chunk (given that program unit 1 is highly cohesive). In the second case, if program unit 1 is coupled to program unit 2, then just one more chunk needs to be comprehended (given that program unit 2 also shows high cohesion). If program unit 1 is also coupled to program unit 3, then it can be expected that STM may fill up much more quickly because program unit 3 shows low cohesion and thus represents several chunks. But, the primary driver of what needs to be comprehended is the extent to which program unit 1 is coupled to other units. If coupling is evident, then the extent of cohesion becomes a comprehension issue.

In the SE literature, coupling and cohesion are generally examined separately and are often tested independently. Although it is possible to analytically examine coupling and cohesion independently, their theoretical effect on complexity is expected to be interactive. The main effect for cohesion prevalent in the SE literature could be seen as a joint effect with some level of coupling, rather than a main effect by itself. This suggests Hypothesis 3:

Hypothesis 3. *For the more highly coupled programs, higher levels of cohesion reduce maintenance effort.*

The proposed model is summarized in Fig. 2.

4 Research Design

To test the model proposed in the previous section, a controlled lab experimental design was chosen in order to maximize the causal inferences that can be made from the results. To increase generalizability, professional software engineers were selected as subjects. The participants attempted two perfective software maintenance tasks: One task was performed under the procedural programming paradigm using the C programming language and the other task was performed under the OO paradigm using the C++ programming language. Effort was measured in terms of the total length of time to complete both tasks without errors (no time limit was specified). The experimental design manipulated the structural complexity of the code that was maintained. Both coupling and cohesion are independently manipulated across two levels, high and low, producing a fully crossed four-cell design, as illustrated in Fig. 3.

4.1 Tasks

For each of the two tasks, there are four variants where each variant corresponds to a cell in Fig. 3. However, apart from the variation in the structure of the code to be maintained, the essence of the two tasks is preserved across each variant. Specifically, both tasks are perfective maintenance tasks and the specification in each case is precisely the same across all four variants. For both tasks, the original specification and the perfective task were described on a single printed page and the existing code was provided in a single file.

The specification of the procedural programming task —"Analyst"—is contained in Appendix A. The task is based on the specifications for a well-documented and widely used exercise (the "B series" of exercises found in Appendix D of Humphrey [25]. Analyst is capable of reading in data (either from the keyboard or a file), displaying, and writing out data to a file. The data is in pairs of one x value and one y value (one observation). The participants were asked to add functionality to the existing code so that the program will calculate and display some basic ordinary least squares regression parameters. The specification was the same regardless of which variant the participant receives. However, given that there are four different structures in existence, the path to task completion was significantly different, depending on which variant the participant actually received.

The specification of the OO task—"UNIDB"—is contained in Appendix B. It is also based on a previously documented maintenance task [14]. UNIDB is a database for a university environment. It currently has lecturer and staff classes as well as corresponding lecturer node and staff node classes that facilitate the implementation of a linked list to store lecturer and staff objects. Participants added a professor class that operates in a fashion similar to that of the existing classes. As with the Analyst task, the UNIDB task is the same for all participants, regardless of the treatment. Only the structure of the code varied with the cell.

4.2 Participants

Lab experiments in SE are sometimes criticized for using novice undergraduate students as participants. In order to

		Cohesion (Coh)	
		Low	High
Coupling (Cpl)	High	HiCpl-LoCoh	HiCpl-HiCoh
	Low	LoCpl-LoCoh	LoCpl-HiCoh

Fig. 3. Experimental design.

```
Main
    show_menu();
    enter_data(data, numobs);
        keyin_data(data, count);
            true_data(test, xy);
    savefile_data(data, numobs);
        check_file(mode);
    getfile_data(data, numobs);
        check_file(mode);
    modify_data(data, numobs);
        display_data(data, numobs);
        change_obs(data, numobs, finished);
            true_data(test, xy);
    display_data(data, numobs);
```

Fig. 4. Calling structure of analyst code.

provide a more rigorous test of the proposed model, only software engineers with multiple years of professional experience were selected as subjects. All of the invited participants were engaged in a Master's in Software Engineering program, a professional degree program at a private university in the Eastern half of the US. To be admitted to the program, the students must have a minimum of two years of professional SE experience. In actuality, those in the program typically have professional SE careers of over four years. Each experimental participant was asked to complete one task set (i.e., one of the four variants of the Analyst and one of the four variants of the UNIDB task). For their participation, subjects were paid $20 and the top one-third performers received another $20. The participants were randomly distributed to one of the four cells. The order of task presentation was counterbalanced to minimize order effects. A variety of data was collected from each participant in order to be able to test for possible confounding correlates.

4.3 Coupling and Cohesion

Software measures such as specific instantiations of the concepts of coupling and cohesion typically focus around a specific program unit such as a function or a class. Coupling and cohesion are examined at a program level rather than at the level of one program unit. The shift in unit of analysis is achieved by averaging the program unit measure across all program units. It can be shown that taking the average of the measures across program units instead of the sum produces a more accurate program level measurement.

For the Analyst task, a guide to the structure is presented in Fig. 4. Each further level of indentation implies that the previous procedure is calling the procedure that is indented. For example, `enter_data` (called from `main`) calls `keyin_data` that in turn calls `true_data`. The table is based on the C code for the variant in the HiCpl-HiCoh cell (see Fig. 4).

For the Analyst task, coupling was manipulated through the presence or absence of the `keyin_data` and `change_obs` procedures. In the higher coupling variant, procedure `enter_data` calls procedure `keyin_data`. In the lower coupling variant, the content of procedure `key_in` (including the call to `true_data`) was placed in the body of procedure `enter_data`. Removing part of the code and placing it in an additional procedure increases the average amount of coupling for the program.

Cohesion was manipulated through the placement of the `dataexist` variable. In the higher cohesion variant, the `dataexist` variable appears (and is changed) only in the main procedure. In the lower cohesion variant, the `dataexist` is sent to and manipulated in more procedures, including `enter_data`, `savefile_data`, `getfile_data`, `modify_data`, and `display_data`. Sending the `dataexist` variable (as a reference parameter) deeper into the calling structure lowers the average cohesion for the program.

For the UNIDB task, the class diagram abstracted from the C++ code is presented in Fig. 5. The table lists the instance variables and the methods for each of the four classes. For the class diagram, the constructor method, the destructor method, and single statement access methods have all been omitted for clarity, though they were present in the code. Fig. 5 is based on code for the variant in cell two.

For the UNIDB task, coupling was manipulated by the presence of method calls from one class to another. In the higher coupling variant, the `staff::calc_tax` method called the `lecturer::calc_tax` method. In the lower coupling variant, the `staff::calc_tax` performed the complete calculation without calling the `lecturer::calc_tax` method. Cohesion was manipulated through the presence or absence of the `lect_node::tax_report` method as part of the `lect_node` class. In the higher cohesion variant, the `tax_report` method was part of the `lect_node` class. In the lower cohesion variant, the code necessary to complete the tax report was placed in the `main` procedure of the program. For the OO paradigm, cohesion is a function of the number of instance variables accessed by each method in a given class. The more instance variables accessed by each method, the higher the level of cohesion for that class. Given that the `tax_report` method accesses all of the instance variables of the `lect_node` class, the existence of this method in the `lect_node` class increases cohesion for the `lect_node` class, thereby increasing average cohesion for the program.

4.4 Total Task Time (Effort)

The participants utilized a task timer running on their PCs that recorded how long they took to complete the tasks. The total effort was a combination of the two task times. Each task time represented the effort to produce defect free code. Though the two times were for different tasks, the total effort taken for both tasks is considered to be the most representative of the maintenance effort given that a particular participant was in the same cell for both tasks. This is analogous to combining the verbal and mathematical scores in the GMAT to achieve an overall test score.

Effort was singled out of the multidimensional phenomenon that is comprehension performance. The code submitted by the participants was run and checked by the researchers to ensure it conformed to the specification and delivered the specified results before the subjects left the experimental setting. Thus, the expectation was that the primary analyses would be able to focus on the effort required without having to make potentially arbitrary

```
class lecturer
//  Instance variables
    last_name, first_name, age, department, employee_id, annual_salary
//  Methods
    print()
    calc_tax(annual_sal)
```

```
class lect_node
//  Instance variables
    lecturer, lect_node
//  Methods
    insert(lect_node)
    new_node(lect_node, lname, fname, age, dept, eid, asal)
    print()
    list_subords(staff_node)
```

```
class staff
//  Instance variables
    last_name, first_name, age, department, employee_id, hourly_wage, boss
//  Methods
    print()
    calc_tax(annual_sal)
```

```
class staff_node
//  Instance variables
    staff, staff_node
//  Methods
    insert(staff_node)
    new_node(staff_node, lname, fname, age, dept, eid, hw, boss)
    print()
```

Fig. 5. Class diagram for the UNIDB code.

comparisons between quicker, but perhaps lower quality, results [44].

4.5 Covariates

A number of individual factors can potentially impact effort. These include previous programming experience in terms of the length of experience (measured as length of experience with the programming languages used in the experiment, length of programming career, and general age), the breadth of experience (number of computer languages known), and general ability (undergraduate GPA). The data for each participant were captured on a questionnaire and assessed as potential covariates for the model.

4.6 Procedure

At the beginning of the first task session, the experimental procedure was explained in a seminar room to the participants. A packet of materials was randomly distributed to each of the participants. The packet contained a consent form, a task specification, and a floppy disk (containing the code). Half of the participants received the procedural task and half of the subjects received the OO task. The participants were asked to read and sign the consent form.

For the task performance itself, the participants moved to cubicles in a computer lab. Each of these cubicles had a similarly equipped PC. The participants used Microsoft's Visual C++ to complete both tasks. Once participants indicated task completion, their task time was noted and their work was checked for errors. If no errors were found, they completed the postexperiment questionnaire. If errors were found, the participants would be asked to continue work on the task until they could produce defect-free code and the time required to do this would be tracked. Finally, the experimental materials were collected, including the changed code. In the second task session, another packet of materials was distributed. The participants that received the procedural task in the first session received the OO task in the second session and vice versa. If a participant received a task in the HiCpl-LoCoh cell (see Fig. 3) during the first task session, then that participant received a task in the same cell during the second session. Once the participant completed the second task, their task time was noted. Their work was checked for errors and, if no errors were found, their participation was complete. If errors were found, the participants would be asked to continue work on the task until they could produce defect-free code and the time required to do this would be tracked.

5 Results

The decision to use only experienced software engineers as subjects resulted in a total of 17 participants. Given the relatively small sample size, an outlier analysis was performed to identify potential outliers that could be anticipated to have a substantial effect on the results. One of the participants was observed to have an outlying (large) value for the total length of time to perform both tasks. Where relevant, the analysis below is performed both with and without this subject. All of the tests in the paper were evaluated using version 12 of the SPSS statistical package.

As a preliminary test of potential learning effects from session one to session two, a paired t-test was performed to compare the time taken in the first session and the time taken in the second session. No statistically significant difference was found between the two sessions, leading to a conclusion that order and learning effects were minimized.

Five factors were assessed as potential covariates for the model (see Section 4.5; the descriptive statistics are reported in Table 2). Correlations of total effort and the individual task times with the potential covariates are shown in Table 3;

TABLE 2
Descriptive Statistics for Potential Covariates

Covariate	Units	n	Min	Max	Mean	Std. Dev
Language experience	Hours	14	30	5500	1863	1943
Programming career	Months	17	6	96	45.29	27.6
Age	Categorical	16	20-24	30-34	25-29	
Number of languages	Absolute number	17	1	5	2.65	1.3
Undergraduate GPA	Absolute number	15	3.00	4.00	3.5	.31

TABLE 3
Correlations for Time and Potential Covariates

	Language Experience	Programming career	Age category	Number of Languages	Undergraduate GPA
Procedural task time	-.11 (.71)	-.31 (.22)	.07 (.80)	-.22 (.40)	-.41 (.13)
OO task time	-.04 (.89)	.42 (.10)	.41 (.11)	-.29 (.25)	-.01 (.98)
Total task time	-.02 (.94)	.18 (.50)	.31 (.25)	-.39 (.12)	-.35 (.20)

TABLE 4
ANOVA for Coupling and Cohesion (Adj. $R^2 = 0.203$)

Source	Type III Sum of Squares	df	Mean Square	F	Sig.
Intercept	1418018473.274	1	1418018473.274	178.387	.00
COUPLING	204624.011	1	204624.011	.026	.88
COHESION	16746242.063	1	16746242.063	2.107	.17
COUPLING * COHESION	36736566.063	1	36736566.063	4.621	.05
Error	103338427.700	13	7949109.823		
Total	1634621406.000	17			
Corrected Total	159600470.471	16			

none of the covariates was significantly correlated with task time (at the p = 0.05 level). In a series of two-way ANOVAs, with each of the potential covariates as the dependent variables and coupling and cohesion as the independent variables, none of the covariates yielded a significant model, implying that the null hypothesis of a homogeneous distribution of the covariates across the four cells could not be rejected.

The experiment utilized a two treatment (with coupling and cohesion as the two treatments) fully crossed design with a single dependent variable (effort). Such a design naturally lends itself to analysis via two-way univariate ANOVA. Table 4 shows the results of such an analysis. Testing the hypotheses in this fashion enables a test for an interaction of the two treatments in addition to the conventional direct effects for each of the individual treatment effects.

The research model (Fig. 2) predicts main effects for coupling and cohesion, as well as an interactive effect of coupling and cohesion on effort, i.e., a nonlinear effect (as predicted in Hypothesis 3). The main effects for coupling and cohesion were in the hypothesized direction, although neither was statistically significant at usual levels. Thus, neither Hypothesis 1 nor 2 can be accepted based on the results reported from this experiment. However, the results reveal a significant interaction effect at the p = 0.05 level. Thus, support is provided for Hypothesis 3. Graphically, the interaction can be seen in Fig. 6. Though the analysis is reported for total effort, the interaction pattern is also observed for the individual effort in each of the two tasks.

Such a combination of effects, no main effects and a significant interaction, is sometimes referred to as a *complete interaction* and its interpretation can be that "it shows the effect of an independent variable depends completely on the level of the other" [31, p. 201]. In addition, the statistically significant interaction effect alone does not indicate which concept, coupling or cohesion, is the main relationship and which represents the moderator. It is only theory that allows such an assertion and, from the discussion in Section 3, it is reiterated that coupling is believed to be the main relationship and cohesion the moderator.

6 Discussion and Conclusions

This paper has endeavored to theoretically define cohesion and coupling drawing on theories of task complexity and human cognition. It has validated experimentally that these two concepts do affect structural complexity (in terms of program comprehension and time to perform a maintenance task). The theoretical foundation for cohesion and coupling and the experimental results showing that cohesion and coupling jointly affect comprehension are new and represent contributions to the SE research and practice.

One way to illustrate the value of the results is to look at the insights that would be obtained without considering the interactive effects of coupling and cohesion. Testing the model without the interaction term for the data set reveals nonsignificant effects for both coupling and cohesion. This is because such a test is dependent on averages for the main

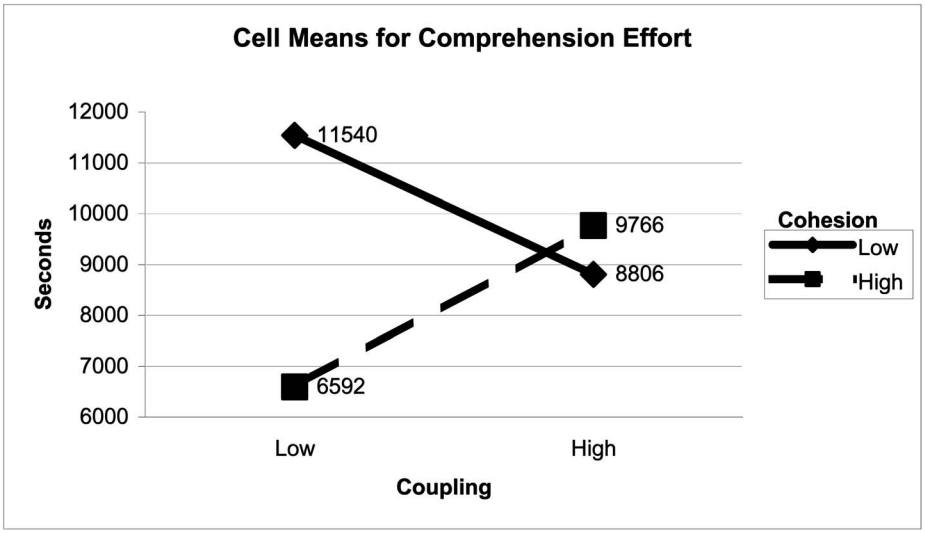

Fig. 6. Interaction effect of coupling and cohesion on effort (p = 0.05).

effects and they tend to cancel one another out as the best model (as suggested by theory and tested with the data) of the underlying situation indicates that it is the interaction that is important, rather than the individual effects in isolation. Without testing for the interaction, a researcher could conclude that neither design dimension, coupling, nor cohesion was significant, a result that would be incorrect and, if acted upon by practitioners, would fail to take advantage of the effect found in this research.

6.1 Implications for Research on Structural Complexity

Coupling and cohesion stand out from other internal software attributes as having what has been noted elsewhere to be a "considerable longevity" [29]. Coupling and cohesion are also interesting because they have been applied to both procedural and OO programming languages. And, a new programming paradigm based on a modular approach would likely readily yield to analysis using coupling and cohesion measures. Furthermore, coupling and cohesion dimensions have proven ubiquitous across a wide variety of structural complexity contexts, including evaluating 4GLs [36] and even in significantly different fields (for an example in business systems planning, see [35] and, in product architecture, [37]). A major contribution of this work lies in the premise that coupling and cohesion have merit above other concepts in the complexity of software structure. Although the idea that coupling and cohesion are important indicators of structural complexity has long been suggested (e.g., [48]), the theoretical approach taken in this paper explicates and solidifies the theoretical basis for coupling and cohesion's essential importance.

6.2 Implications for Structuring Software

The practical value of this research lies in recognizing that the effect of coupling and cohesion on effort is joint and that levels of coupling and cohesion are unlikely to be able to be changed without some impact on the other. A focus on improving one concept to the exclusion of the other may not necessarily reduce the overall structural complexity of the software. For example, a focus on cohesion will produce many program units, with each individual program unit demonstrating a high level of cohesion. However, a high number of program units implies a high level of coupling among the program units. A focus on coupling will produce a small number of program units, with each program unit demonstrating a low level of cohesion. Though coupling among the program units would be low, understanding individual program units would be difficult. In structuring programs, both coupling and cohesion should be jointly considered to arrive at the most desirable structure for the program.

This research shows that a higher level of structural complexity (combining coupling and cohesion) is associated with more effort. The logical conclusion is that if some effort were expended in reducing structural complexity, such effort would be repaid in terms of reducing later effort. Structural complexity can be considered at the program, class, or system level. The greater the level of complexity, the greater the utility in considering coupling and cohesion. And, it is at the system level, where the largest amount of complexity is likely to be present, that the application of coupling and cohesion decisions will have the most return in analyzing, and ultimately controlling, structural complexity. In terms of the software lifecycle, as software evolves, it grows in complexity, all else being equal. Thus, a change made early in the life of a piece of software is easier to accomplish than that same change made later. Structural complexity can be assessed once a piece of software has achieved some level of specification, even early in its design before much code exists. Assessing and reducing the structural complexity of software in the early stages of development (e.g., at the end of the design stage) is likely to have substantial advantageous impacts on effort during later stages. It is at the design stage where the insights from this paper can be most influential, when the structure is the most flexible and changing it is the least troublesome. Doing so would have the largest positive benefit for

TABLE 5
Test Data

Program	New and changed lines of code (x)	Development hours (y)	x^2	xy
1	186	15	34596	2790
2	699	69.9	488601	48860.1
3	132	6.5	17424	858
4	272	22.4	73984	6092.8
5	291	28.4	84681	8264.4
6	331	65.9	109561	21812.9
7	199	19.4	39601	3860.6
8	1890	198.7	3572100	375543
9	788	38.8	620944	30574.4
10	1601	138.2	2563201	221258.2
Total	6389	603.2	7604693	719914.4

software through its impact on effort. And, the key insight is to consider structural complexity in terms of the interaction of coupling and cohesion, not simply each independently.

6.3 Limitations

Readers are generally cautioned against making too broad a generalization from these results as experimental designs can have limitations based on their setting. Moreover, the relatively small scale of problem could lead to conclusions different from those that might be divined in field settings. However, there is a high degree of confidence in the results given the opportunity for experimental control.

Every effort was made to maximize the ability to generalize results. Experienced software engineers were recruited to participate. The experimental design also manipulated the independent construct of interest (structural complexity) rather than observing the correlation of its variance with dependent variables (such as effort), enabling a stronger case to be made for causation between the independent and dependent variables. Finally, each participant completed a task in each of the two major programming paradigms rather than a single one, increasing both the robustness of the results and generalization across both of the programming paradigms.

Nevertheless, certain research design decisions indubitably bound the generalizability of these results. For example, only a single type of task (perfective maintenance) was studied. It may be that other types of maintenance will best suit a different model. However, though there is some variance in reported maintenance task distributions [46], most would argue that perfective maintenance is a dominant type of maintenance (e.g., [30]), hence its selection for the experimental type of task. Finally, a limited set of internal software attributes have been studied in this paper. Though the approach and results are of significance to the field, they can also be used as stepping stones to open up new ways to consider a wider set of internal attributes, such as control flowgraphs and cyclomatic complexity, their interrelationships, and their independent and interdependent effect relationship on comprehension performance.

6.4 Conclusions and Future Research

Comprehension performance is a rich and varied phenomenon under intense scrutiny. Though this paper focused on effort, the literature has discussed other dimensions, including quality (e.g., [44]). It would be valuable to extend this work to other aspects of comprehension performance and under a wider range of contexts, such as corrective maintenance. It may be that the nonlinear effect of coupling and cohesion on effort would be repeated for other dimensions of comprehension performance (for such a result, see [38]). Though the particular research design for testing the model is experimental, the model could be applied in a much larger context, including large-scale systems. The concepts of coupling and cohesion can be operationalized and applied throughout the software life cycle, including during design and across different levels of software, such as small code segments all the way to large systems. This is also why there is such a value in examining the deep structure of problems and deriving a model from blending multiple theoretical abstractions. Furthermore, it is software designers who, it is believed, would most benefit from the insights in the paper as it is at the design stage that issues of structure tend to be most relevant and flexible, whereas, at later stages in the life of software, the structure tends to be more rigid and changing it is much more expensive.

It is a fundamental premise of this work that the concepts of coupling and cohesion have merit above and beyond many other software complexity concepts. Coupling and cohesion have historically been described and discussed; their role as essential indicators of the structural complexity

TABLE 6
Attributes for New Professor Class

Attribute	Data type
last_name	char[40]
first_name	char[40]
Age	unsigned int
Department	char[40]
employee_id	unsigned int
monthly_salary	float

TABLE 7
New Professors Data

last_name	first_name	age	department	employee_id	monthly_salary
Jones	Fred	48	CS	78078439	5000
Hilper	Harry	49	UMD	75980987	5200
Moran	Linda	44	Katz	81875987	5100

of software has been further explicated and reinforced by this experiment. Acceptance of this premise opens up new horizons for software complexity measurement. Future research efforts can then focus on confirming and refining coupling and cohesion measures and models. Practice can be aided by the development of better tools to facilitate the use of such measurement. Pedagogically, progress can be made in elevating programming courses and classes beyond lessons in syntax to consider the implications of design decisions. In addition, coupling and cohesion have been conventionally considered as independent concepts, something that these results begin to question, although further studies would clearly be desirable.

In closing, this paper considers the concepts of coupling and cohesion well developed and explored. In addition, the concepts are exceptionally well endowed with operationalizations and measures. By the time the next modular programming paradigm is widely adopted, coupling and cohesion are expected to continue to facilitate complexity evaluation and prediction. What is far less clear is the direction that should be taken in further exploring measurement of these concepts. Researchers need to take stock of the existing measures and concentrate on building models based on theory and rigorous testing of those models. This paper represents a first step in fulfilling such a goal.

Appendix A

Procedural Task

A.1 Analyst

The Analyst application is being prepared to perform regression analysis. A user can enter new data or retrieve, display, and modify existing data. Error checking is performed on data entry, file input, and menu choice entry.

You have been asked to enhance the Analyst so that it can perform basic regression calculations, i.e., to calculate the linear regression coefficients using the ordinary least squares method. You have been provided with the test data shown in Table 5. The first three columns represent the test data (which can also be found in the file test.txt), while the last two columns are provided to facilitate the example calculation shown below the table.

$$\beta_1 = \frac{\left(\sum_{i=1}^{n} x_i y_i\right) - \left(n x_{avg} y_{avg}\right)}{\left(\sum_{i=1}^{n} x_i^2\right) - \left(n\left(x_{avg}^2\right)\right)}$$

$$= \frac{(719914.4) - (10^{*}638.9^{*}60.32)}{(7604693) - (10^{*}(638.9^2))} = 0.095,$$

$$\beta_0 = y_{avg} - \beta_1 x_{avg} = 60.32 - 0.095^{*}638.9 = -0.351.$$

The enhanced version of Analyst will add a single menu item that will first calculate and then display all four of the following terms: x_{avg}, y_{avg}, β_0, and β_1.

Add your code to the existing code file.

Appendix B

OOP Task

B.1 UNIDB

The UNIDB application is being prepared to perform database functions in a university environment. The main entities already in existence are lecturers and staff.

You have been asked to enhance the UNIDB by adding a new professor class and a new professor list class (in the same way that the lecturer and staff classes have corresponding list classes). The new professor class will have the attributes listed in Table 6.

The Professor class should be able to construct an object using parameters and, like any of the existing classes, be able to set all of the attributes when required.

In addition, your professor node class must create a method that lists all of the professors followed by their annual salary and their annual taxes. Also, as each professor is a head of department, you must create a method to list the lecturers in the department of each professor (this would be done in a manner similar to the way the lect_node class lists all of the subordinates for each of the lecturers in method lect_node::list_subords).

To test your work, create three new professors with the data in Table 7.

Add your code to the existing code file.

Acknowledgments

This work was funded in part by the Center for Electronic Markets and Enterprises at the University of Maryland, the Software Industry Center at Carnegie Mellon University, and the Institute for Industrial Competitiveness at the University of Pittsburgh.

References

[1] R. Adamov and L. Richter, "A Proposal for Measuring the Structural Complexity of Programs," *J. Systems and Software*, vol. 12, pp. 55-70, 1990.

[2] J.R. Anderson, L.M. Reder, and C. Lebriere, "Working Memory: Activation Limitations on Retrieval," *Cognitive Psychology*, vol. 30, pp. 221-256, 1996.

[3] A.L. Baker et al., "A Philosophy of Software Measurement," *J. Systems and Software*, vol. 12, pp. 277-281, 1990.

[4] R.D. Banker et al., "Software Complexity and Maintenance Costs," *Comm. ACM*, vol. 36, no. 11, pp. 81-94, 1993.

[5] R.D. Banker, G.B. Davis, and S.A. Slaughter, "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study," *Management Science,* vol. 44, no. 4, pp. 433-450, 1998.

[6] R.D. Banker and S.A. Slaughter, "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement," *Information Systems Research,* vol. 11, no. 3, pp. 219-240, 2000.

[7] R. Berry, "Trends, Challenges and Opportunities for Performance Engineering with Modern Business Software," *IEE Proc. Conf. Software Eng.,* vol. 150, no. 4, pp. 223-229, 2003.

[8] J.M. Bieman and L.M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Eng.,* vol. 20, no. 8, pp. 644-657, Aug. 1994.

[9] F.P. Brooks, "Three Great Challenges for Half-Century-Old Computer Science," *J. ACM,* vol. 50, no. 1, pp. 25-26, 2003.

[10] D.J. Campbell, "Task Complexity: A Review and Analysis," *Academy of Management Rev.,* vol. 13, no. 1, pp. 40-52, 1998.

[11] H.S. Chae, Y.R. Kwon, and D.H. Bae, "A Cohesion Measure for Classes in Object-Oriented Classes," *Software—Practice and Experience,* vol. 30, no. 12, pp. 1405-1431, 2000.

[12] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 629-639, Aug. 1998.

[13] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, June 1994.

[14] J. Daly et al., "Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software," *Empirical Software Eng.,* vol. 1, pp. 109-132, 1996.

[15] E.W. Dijkstra, "GOTO Statement Considered Harmful," *Comm. ACM,* vol. 11, no. 8, p. 147, 1968.

[16] F.B. Abreu and R. Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *J. Systems and Software,* vol. 26, pp. 87-96, 1994.

[17] N.E. Fenton and S.L. Pfleeger, "Science and Substance: A Challenge to Software Engineers," *IEEE Software,* pp. 86-95, 1994.

[18] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach, Revised.* Boston, Mass.: Int'l Thomson Publishing, 1997.

[19] R.K. Fjeldstad and W.T. Hamlen, "Application Program Maintenance Study: Report to Our Respondents," *Tutorial on Software Maintenance,* pp. 11-27, 1983.

[20] V.R. Gibson and J.A. Senn, "System Structure and Software Maintenance Performance," *Comm. ACM,* vol. 32, no. 3, pp. 347-358, 1989.

[21] N. Gorla and R. Ramakrishnan, "Effect of Software Structure Attributes Software Development Productivity," *J. Systems and Software,* vol. 36, no. 2, pp. 191-199, 1997.

[22] N. Gross and S. Hamm, "Industry Outlook: Software," *BusinessWeek,* 11 Jan. 1999.

[23] R. Harrison, S.J. Counsell, and R.V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 491-496, 1998.

[24] J. Hartmanis, "On Computational Complexity and the Nature of Computer Science," *Comm. ACM,* vol. 37, no. 10, pp. 37-43, 1994.

[25] W.S. Humphrey, "A Discipline for Software Engineering," *The SEI Series in Software Eng.,* p. 790, 1995.

[26] D.H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Eng.,* vol. 11, no. 8, pp. 749-757, Aug. 1985.

[27] J.K. Kearney et al., "Software Complexity Measurement," *Comm. ACM,* vol. 29, no. 11, pp. 1044-1050, 1986.

[28] C.F. Kemerer, "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Annals of Software Eng.,* vol. 1, no. 1, pp. 1-22, 1995.

[29] C.F. Kemerer, "Progress, Obstacles, and Opportunities in Software Engineering Economics," *Comm. ACM,* vol. 41, no. 8, pp. 63-66, 1998.

[30] C.F. Kemerer and S.A. Slaughter, "An Empirical Approach to Studying Software Evolution," *IEEE Trans. Software Eng.,* vol. 25, no. 4, pp. 493-509, 1999.

[31] G. Keppel, *Design and Analysis: A Researchers Handbook,* third ed. Prentice Hall, 1991.

[32] J. Kim and F. Lerch, "Cognitive Process in Logical Design: Comparing Object Oriented Design and Traditional Functional Decomposition Methodologies," *Proc. Conf. Human Factors in Computing Systems (CHI '92),* 1992.

[33] B.A. Kitchenham et al., "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Trans. Software Eng.,* vol. 28, no. 8, pp. 721-734, Aug. 2002.

[34] A. Lakhotia, "Rule-Based Approach to Computing Module Cohesion," *Proc. 15th Int'l Conf. Software Eng.,* 1993.

[35] H. Lee, "Automatic Clustering of Business Processes in Business Systems Planning," *European J. Operational Research,* vol. 114, no. 2, pp. 354-362, 1999.

[36] S.G. MacDonell, M.J. Shepperd, and P.J. Sallis, *Metrics for Database Systems: An Empirical Study,* 1997.

[37] J.H. Mikkola and O. Gassmann, "Managing Modularity of Product Architectures: Toward an Integrated Theory," *IEEE Trans. Eng. Management,* vol. 50, no. 2, pp. 204-218, 2003.

[38] K.-H. Moller and D.J. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. First Int'l Software Metrics Symp.,* 1993.

[39] G.J. Myers, *Composite/Structured Design.* New York: Van Nostrand Reinhold, 1978.

[40] A.J. Offutt, M.J. Harrold, and P. Kolte, "A Software Metric System for Module Coupling," *J. Systems and Software,* vol. 20, pp. 295-308, 1993.

[41] D.L. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[42] S.L. Pfleeger et al., "Status Report on Software Measurement," *IEEE Software,* pp. 33-43, Mar./Apr. 1997.

[43] S. Purao and V. Vaishnavi, "Product Metrics for Object-Oriented Systems," *ACM Computing Surveys,* vol. 35, no. 2, pp. 191-221, 2003.

[44] V. Rajlich and G.S. Cowan, "Towards Standard Experiments in Program Comprehension," *Proc. Int'l Workshop Program Comprehension,* 1997.

[45] H.D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Trans. Software Eng.,* vol. 13, no. 3, pp. 344-354, Mar. 1987.

[46] S.R. Schach et al. "Determining the Distribution of Maintenance Categories: Survey versus Measurement," *Empirical Software Eng.,* vol. 8, no. 4, pp. 351-365, 2003.

[47] J.E. Smith, "Characterizing Computer Performance with a Single Number," *Comm. ACM,* vol. 31, no. 10, pp. 1202-1206, 1988.

[48] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems J.,* vol. 13, no. 2, pp. 115-139, 1974.

[49] R. Subramanyam and M.S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexty: Implications for Software Defects," *IEEE Trans. Software Eng.,* vol. 29, no. 4, pp. 297-310, 2003.

[50] P. Tonella, "Concept Analysis for Module Restructuring," *IEEE Trans. Software Eng.,* vol. 27, no. 4, pp. 351-363, Apr. 2001.

[51] I. Vessey and R. Weber, "Some Factors Affecting Program Maintenance: An Empirical Study," *Comm. ACM,* vol. 26, no. 2, pp. 128-134, 1983.

[52] A. von Mayrhauser and M.A. Vans, "Program Comprehension during Software Maintenance and Evolution," *Computer,* vol. 12, pp. 44-55, 1995.

[53] A. von Mayrhauser and M.A. Vans, "Identification of Dynamic Comprehension Processes during Large Scale Maintenance," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 424-437, June 1996.

[54] R.E. Wood, "Task Complexity: Definition of the Construct," *Organizational Behavior and Human Decision Processes,* vol. 37, pp. 60-82, 1986.

[55] E. Yourdon and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice Hall, 1979.

David P. Darcy received the BComm and MMangt.Sc degrees from University College Dublin and the PhD degree from the University of Pittsburgh. He is currently an assistant professor of information systems in the Decision and Information Technologies Department, Robert H. Smith School of Business, University of Maryland. Previously, he was a member of the Faculty of Commerce, University College Dublin, Ireland. His research interests include software complexity, software metrics, software development processes, and software project management, and he has previously published in the *IEEE Transactions on Software Engineering*. He is a member of the Association for Information Systems.

Chris F. Kemerer received the BS degree from the Wharton School at the University of Pennsylvania and the PhD degree from Carnegie Mellon University. He is the David M. Roderick Professor of Information Systems at the Katz Graduate School of Business, University of Pittsburgh, and an adjunct professor of computer science at Carnegie Mellon University. Previously, he was an associate professor at MIT's Sloan School of Management. His research interests include management and measurement issues in information systems and software engineering and he has published numerous articles on these topics as well as editing two books. He is currently conducting research on software evolution under a grant from the US National Science Foundation. He has served on a number of editorial boards and is the immediate past editor-in-chief of *Information Systems Research*. He is a member of the IEEE Computer Society and is a past associate editor of the *IEEE Transactions on Software Engineering*.

Sandra A. Slaughter received the PhD degree in management information systems from the University of Minnesota in 1995. She is currently an associate professor in the Tepper School of Business at Carnegie Mellon University. Prior to joining the faculty at Carnegie Mellon University, Dr. Slaughter worked in industry as a project leader and systems analyst at Hewlett-Packard, Rockwell International, and Square D Corporation. She has consulted with the Information Technology Management Association and with several companies on software development-related issues. Her research is motivated by her experience in software development and focuses on software productivity and quality. Currently, she is conducting research funded by the US National Science Foundation on software evolution. Her thesis on software development practices and software maintenance performance won first place in the doctoral dissertation competition held by the International Conference on Information Systems (ICIS) in 1995. Since then, she has gone on to publish more than 70 articles on software development issues in leading research journals, conference proceedings, and edited books. Her work has received seven best paper awards at major conferences, including a recent award for a paper on open source software development. She currently serves or has served on a number of editorial boards for major journals in information systems and management science. She is a member of the ACM and the Association for Information Systems.

James E. Tomayko is a teaching professor at the School of Computer Science at Carnegie Mellon and a part-time senior member of the technical staff of the Software Engineering Institute (SEI). He is the director emeritus of the Master Software Engineering Program in the School of Computer Science at Carnegie Mellon University. Previously, he was the leader of the Academic Education Project at SEI. Prior to that, he founded the software engineering graduate program at Wichita State University. He has worked in industry through employee, contract, or consulting relationships with NCR, NASA, Boeing Defense and Space Group, CarnegieWorks, Xerox, the Westinghouse Energy Center, Keithley Instruments, and Mycro-Tek. He has given seminars and lectures on software fault tolerance, software development management, fly-by-wire, and software process improvement in the United States, Canada, Mexico, Argentina, Spain, Great Britain, South Africa, Germany, China, and Columbia. His courses on managing software development and overviews of software engineering are among the most widely distributed courses in the SEI Academic Series. He has had a parallel career in the history of technology, specializing in the history of computing in aerospace, and has written five books and several articles on spacecraft computer systems and software, concentrating primarily on NASA's systems. He is a member of the editorial board of the *IEEE Annals of the History of Computing*. He is a member of the IEEE Computer Society.