Software Complexity: Toward a Unified Theory of Coupling and Cohesion

David P. DARCY, and Chris F. KEMERER

Abstract-- **Knowledge work is generally regarded as involving complex cognition, and few types of knowledge work are as important in the modern economy as software engineering (SE). A large number of measures have been developed to analyze software and its concomitant processes with the goals of evaluating, predicting and controlling its complexity. While many effective measures can be used to achieve these goals, there is no firm theoretical basis for choosing among measures. The first research question for this paper is: how to theoretically determine a parsimonious subset of software measures to use in software complexity analysis? To answer this question, task complexity is studied; specifically Wood's model of task complexity is examined for relevant insights. The result is that coupling and cohesion stand out as comprising one such parsimonious subset. The second research question asks: how to resolve potential conflicts between coupling and cohesion? Analysis of the information processing view of cognition results in a model of cohesion as a moderator on a main relationship between coupling and complexity. The theory-driven approach taken in this research considers both the task complexity model and cognition and lends significant support to the developed model for software complexity. Furthermore, examination of the task complexity model steers this paper towards considering complexity in the holistic sense of an entire program, rather than of a single program unit, as is conventionally done. Finally, it is intended that by focusing software measurement on coupling and cohesion, research can more fruitfully aid both the practice and pedagogy of software complexity management.**

Index Terms—**Software, Complexity, Coupling, Cohesion.**

I. Introduction

Few activities are as complex as the effective design, programming and maintenance of software. Campbell's framework of *task complexity* is widely regarded as representative of research into task complexity. Within Campbell's framework, the design, implementation and maintenance of software falls into the most complex category (fuzzy tasks) with multiple paths to end states, multiple end states, conflicting interdependence between paths and the presence of uncertainty between paths and outcomes [1].

Manuscript received ???. (Write the date on which you submitted your paper for review.) (sponsor acknowledgment goes here).

D. P. Darcy is with the Department of Decision and Information Technologies, Robert H. Smith School of Business, University of Maryland, College Park, MD 20742, USA (telephone: 301-405-4900, e-mail: ddarcy@rhsmith.umd.edu).

According to BusinessWeek's industry review, over the decade of the 1990's, software productivity actually declined slightly [2]. Any new initiatives that are targeted at significantly improving the problems in designing, programming and maintaining software is headline news, as evidenced by a recent front page story in the Wall Street Journal [3].

Ever since Dijkstra's criticism of the 'GOTO' statement [4] and Parnas' information hiding concept [5, 6], software engineering (SE) has attempted to use software measures and models to reduce complexity and thereby achieve other goals such as better productivity. Over a decade ago, reviews of research into software measurement yielded over 500 articles on the topic [7] and over 100 complexity measures [8]. In the time since those evaluations, software measurement has produced an even more extensive and diverse set, despite calls to the contrary [9]. Furthermore, the relatively new object-oriented (OO) programming paradigm has necessitated reevaluations of measures that were originally developed for the procedural programming paradigm, thereby producing even more measures (e.g. [10]).

Despite an ever-increasing set of recommended program methodologies and structures intended to reduce software complexity, empirical results on their efficacy have been mixed [11-13]. It is well beyond time to rationalize research directions in the field of software complexity measurement. Instead of further increasing the set of software complexity measures, the first research question addressed in this paper is to determine whether some subset of software complexity measures can be effective in measuring software complexity. Few programmers or software project managers can simultaneously or even usefully interpret very many of the software measures in existence. If a subset could be shown to represent the relevant dimensions of complexity, then that set can be used to aid programmers and managers in the evaluation, prediction and control of software complexity. As part of this first research question, it is necessary to determine whether it is feasible to select a subset of measures small enough to enable parsimonious evaluation, prediction and control of software complexity, yet large enough to include all the salient dimensions of software complexity.

The approach taken in this paper to answer the first

C. F. Kemerer is with the Joseph M. Katz Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA. (telephone 412-648-1572, e-mail: ckemerer@katz.pitt.edu).

research question is to choose a subset of measures based on theoretical grounds. Early attempts to choose a subset of measures have included developing measures, singly or in small numbers based on observing programmers in the field (e.g. [14]) or adapting, refining and/or improving existing measures (e.g. [10, 15, 16]). However, neither of these two approaches allows the direct answer of the first research question, that is, what subset might be most effectively chosen and applied. Software measurement research is replete with calls for theoretical guidance on this issue, as is clear from Figure 1.

Cite	Quote
[17]	"Successful software-complexity-measure development must be motivated by a theory of programming behavior", p. 1044
[18]	"For research results to be meaningful, software measurement must be well grounded in theory", p. 277
[19]	"For cohesion measures to provide meaningful measurements, they must be rigorously defined, accurately reflect well understood software attributes, and be based on models that capture those attributes", p. 644
[20]	"As software engineers, we tend to neglect models. In other scientific disciplines, models act to unify and explain, placing apparently disjoint events in a larger, more understandable framework", p. 37
[21]	"However, widespread adoption of object-oriented metrics in numerous application domains should only take place if the metrics can be shown to be theoretically valid", p. 491
[22]	"The history of software metrics is almost as old as the history of software engineering. Yet, the extensive research and literature on the subject has had little impact on industrial practice. … we feel that it is important to explicitly model: (a) cause and effect relationships", p. 149
[23]	"It is not uncommon to find software organizations that are: 1) collecting redundant data; 2) collecting data that people who do not even know it exists inside their organization. For these reasons, improving ongoing measurement is an important problem for many software organizations. We believe the solution for this problem needs to address two key issues: 1) to better understand and structure this ongoing measurement; 2) to better explore the data that the organization has already collected", p. 484
[24]	"Yet, the mechanism underlying the software development process is not understood sufficiently well or is too complicated to allow an exact model to be postulated from theory", p. 567

Figure 1: Calls for theoretically based software measurement research

To some extent, the SE research community has already weighed in on the first research question. Specifically, their response has been that software complexity, in all its incarnations, is not likely to be captured by a single measure [25-27]. The search for a single all encompassing measure has been likened to the "search for the Holy Grail" [28]. To 'find' such a measure would be like trying to gauge the volume of a box by its length, rather than a combination of length, breadth and height.

To deal with the difficulty inherent in the concept of software complexity, the SE research community has rallied by proposing and validating a myriad of indirect measures of software complexity [9]. This response would seem to suggest that there might be no upper limit to the ways in which software complexity can and should be measured. In order to develop a theoretically based answer to the subset question, Wood's task complexity model is examined for its insights into task complexity in general and software complexity in particular [29]. Wood's model is generally representative of task complexity approaches [1]. It is more closely studied in this paper because it is considered influential and because it has already been found to be useful in a software maintenance context [30].

Examination of Wood's model leads to the conclusion that the software measurement categories of coupling and cohesion are the logical subset of choice from the extant list of software measures. The underlying premise for this choice lies in the fact that the basic methodology for programming is to apply a *divide and conquer* approach. Programming proceeds by dividing the overall task into parts and then designing, implementing or maintaining each individual part. Wood's model focuses on the complexity of the individual parts, the complexity of the relationships between the parts and the change in these two complexities over time. Cohesion is a measure of the complexity of a single program unit and coupling is a measure of the complexity of the relationships or linkages among program units. Wood's third component of task complexity, changes in the first two components over time, is not directly considered in this paper, though its implications for future research are considered.

Coupling and cohesion stand out from other software measurement categories as having what appears to be a "considerable longevity" [31]. Coupling and cohesion are also interesting because they have been applied to procedural programming languages as well as OO languages. In turn, any future programming paradigm based on a divide and conquer approach would likely readily yield to analysis using coupling and cohesion measures. Furthermore, coupling and cohesion measures have proven ubiquitous across a wide variety of measurement situations, including evaluating 4GLs [32] and even in significantly different fields (for an example in business systems planning, see [33]). There is also empirical evidence that coupling and cohesion predict effort, even when size and programmer variables are controlled [34].

In answering the first research question, coupling and cohesion have been proposed as sufficient to encompass much

of the complexity of software. Nevertheless, in answering the first question (i.e. by proposing coupling and cohesion as constituting the chosen set), the second question naturally crystallizes: how is it possible to reconcile tradeoffs or conflict in simultaneously improving coupling and cohesion? To answer this second question, cognition is explored. Understanding cognition in general is difficult [1] and general understanding does not provide an extensive guide to understanding the specifics of programming cognition. As a result, beyond straightforward notions, such as the use of a divide and conquer approach (i.e. modularization) that is the norm throughout SE, there is very little parsimony in modeling software design, programming and maintenance activities.

The difficulty in conceptualizing the cognitive processes for software design, implementation and maintenance [35-37] may represent one explanation for why software measure validation is so focused on measurement theory to the point of excluding any consideration of cognition [38]. Perhaps, cognitive models are not yet sufficiently mature or detailed to support the software measurement validation process.

Despite the difficulty, however, cognition must be considered to add credibility to the choice of measures, given cognition's role as the likely major source of variation in software design, development and maintenance [12, 40, 41]. The second research question more specifically asks how tradeoffs between coupling and cohesion can be made. It is through examination of cognition issues that such tradeoffs can be decided. Ultimately, a model of software complexity is derived in this paper where it is argued that cohesion is a moderator on the coupling relationship. In a sense, answering the first research question allows the paper to draw the appropriate boxes in the nomological network, while answering the second research question allows the paper to draw the appropriate arrows connecting the boxes in the model.

The remainder of this paper is organized as follows. The theoretical threads that underpin this paper are outlined in the next section. The threads are integrated into a research model and predictions based on the model are outlined in section III. Section IV concludes the paper by reiterating its contributions and suggests some future research directions.

II. Theoretical Background

As Wood's model of task complexity is the primary theoretical foundation for this paper, it is detailed first. To form the basis for the research model, Wood's model is integrated with the information processing perspective on cognition. As such, cognition research, both general information processing concepts and those relevant to the domain of SE, are covered second. The third section examines coupling and cohesion in three subsections. The first subsection reviews coupling and cohesion measures developed for the procedural programming paradigm. The second subsection provides a similar discussion for the measures developed in the OO programming paradigm. The third subsection examines the surprisingly limited array of empirical work using coupling and cohesion measures. The background section is completed by a brief review.

A. Wood's Task Complexity Model

Construct confounding and measurement problems motivated Wood's model of task complexity [29]. He considered four theoretical frameworks upon which to base his definition: *task qua task*, *task as behavior requirements*, *task as behavior description* and *task as ability requirements*. A combination of the first two approaches was selected as having "the greatest potential for the general theoretical analyses of tasks" ([29], p. 64). These approaches were chosen specifically to avoid confounding the construct with individual differences. The model purports to represent task complexity independently of the level of cognition required to perform the task.

Wood considered tasks to have three essential components: products, required acts and information cues. Products are "entities created or produced by behaviors which can be observed and described independently of the behaviors or acts that produce them" ([29], p. 64). An act is "the pattern of behaviors with some identifiable purpose or direction" ([29], 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" ([29], p. 65). The relationships are graphically illustrated in Figure 2. Using the three atoms of products, acts and information cues, three sources of task complexity were defined: component, coordinative and dynamic.

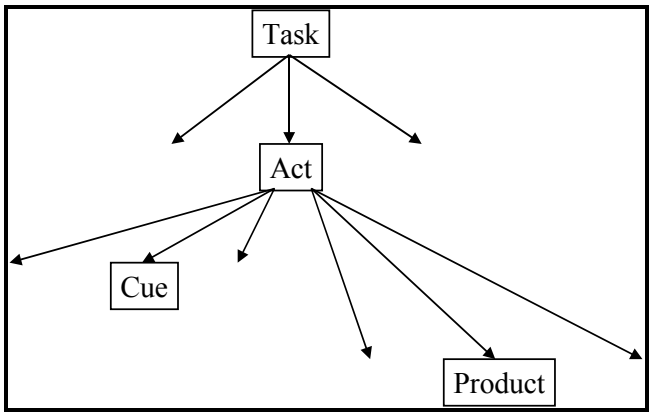

Figure 2: Task Components

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" ([29], p. 66). The emphasis on distinct is because component complexity is not changed just because a single act is repeated twice, three times or a hundred times. On the other hand, component complexity is increased as the number of distinct acts is increased. Equation **(1)** shows the general

Component complexity	$TC_1 = \sum_{j=0}^{p} \sum_{i=1}^{l} W_{ij}$ (1)
Coordinative complexity	$TC_2 = \sum_{i=1}^{n} r_i$ (2)
Dynamic complexity 1	$TC_3 = \sum_{t=1}^{m} \left\lvert TC_{1(t+1)} - TC_{1(t)} \right\rvert + \left\lvert TC_{2(t+1)} - TC_{2(t)} \right\rvert$ (3)
Dynamic complexity 2	$TC_3 = \sum_{t=1}^{m} \left\lvert TC_{1(t+1)}{}^{s} - TC_{1(t)}{}^{s} \right\rvert \left(1 - \rho_{TC_1}\right) + \left\lvert TC_{2(t+1)}{}^{s} - TC_{2(t)}{}^{s} \right\rvert \left(1 - \rho_{TC_2}\right)$ (4)
Total task complexity	$TC_t = \alpha TC_1{}^{s} + \beta TC_2{}^{s} + \gamma TC_3{}^{s}$ (5)
Where:	TC_1 is component complexity, p is the number of subtasks in the task, l is the number of acts in a subtask, W_{ij} is the number of information cues to be processed for act i of subtask j, TC_2 is coordinative complexity, n is the number of acts in the task, r_i is the number of precedence relations between act i and all other acts in the task, TC_3 is dynamic complexity, m is the number of time periods over which the task is measured, $TC_1{}^{s}$ is component complexity measured in standardized units, $TC_2{}^{s}$ is coordinative complexity measured in standardized units, $TC_3{}^{s}$ is dynamic complexity measured in standardized units, ρ_{TC1} is the autocorrelation coefficient for TC_1, ρ_{TC2} is the autocorrelation coefficient for TC_2, TC_t is total task complexity, $\alpha < \beta < \gamma$

Figure 3: The analytics for Wood's model of task complexity [29]

formula for calculating component complexity. All of the equations referenced in this subsection can be found in Figure 3.

Coordinative complexity covers the "nature of relationships between task inputs and task products" ([29], p. 68). The form, the strength and the sequencing of the relationships are all considered aspects of coordinative complexity. It is also important to consider only non-redundant or distinct relationships, just as was the case for component complexity. Equation **(2)** provides a formula for calculating 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" ([29], p. 71). Over the task completion time, parameter values are non-stationary. 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. Equation **(3)** shows one formula for calculating dynamic complexity. Wood went on to consider the effect that the predictability of change from one time period to the next would have on the calculation of dynamic complexity. Specifically, Wood considers the serial or autocorrelation between component and coordinative complexity from one time period to the next. Autocorrelation and dynamic complexity are negatively related; if autocorrelation is high, then dynamic complexity will not be high. To include this effect, autocorrelation terms are added to the calculation as shown in **(4)**.

Total task complexity is a function of the three types of task complexity. At its simplest, the function is a linear combination of the three types. Each of the three components is expressed in standardized units. A unit of dynamic complexity contributes more than a unit of coordinative complexity that, in turn, contributes more than a unit of component complexity. Algebraically, we have **(5)**.

B. 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 (read only memory, RAM) and secondary storage (hard disks, floppy disks, CD-ROM, DVD-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 our minds work [42-49]. We have a primary storage area called short-term memory (STM). We must have all of our data and instructions in STM before we can 'think' [42, 50]. Accessing a process or data in secondary storage (called long

term memory or LTM) is slow compared to accessing something in STM [51]. Our STM is also small relative to our LTM [51]. In early cognitive psychology research, Miller determined our STM capacity to be limited to about 7 (plus or minus two) items [52].

What is very different between the computer and the way the mind works is that in the mind, an item or, rather, a unit of storage, is not as homogenous as a byte is for computer storage. The relevant units of mental storage are *chunks* [53, 54]. What constitutes a chunk varies by person and domain [55, 56]. A single chunk for a novice chess player might represent the way a particular piece moves. The intermediate chess player might store a certain strategy for a position covering seven or eight pieces as one chunk. For a chess expert, a single chunk might be a chessboard, complete with placements for all of the remaining pieces in a famous game, for a particular move [53]. As an extension of this model, it follows that if we store chunks in our memory, then it is conceivable that we also comprehend things in chunks. Experiments on this proposition have been supportive (for general cognition, see [46, 57, 58]; for cognition specific to programming, see [59]).

The basic concepts of the information processing perspective - chunks, STM and LTM - are pervasive throughout programming cognition models [13, 60-63]. This commonality is not by accident: all of these models use the information processing approach, either implicitly or explicitly, at their core.

Many believe that the essence of programming is comprised of two major tasks: abstracting a design and coding that design [64-67]. Inherent in a given design is a certain level of complexity [68]. Since the dawn of programming, the fundamental approach has been to divide and conquer the design [4, 5] by splitting the implementation into parts.

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 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 generically refer to modules, procedures, functions, and classes.

C. Coupling and Cohesion

This subsection is separated into four subsections corresponding to coupling and cohesion measure development in each of the two programming paradigms: procedural and OO, a subsection detailing examples of measure calculations and a subsection on coupling and cohesion empirical research.

1) Measures for the Procedural Programming Paradigm

The source for much of the original thinking on coupling and cohesion can be traced to [14]. Besides the first author, each of the other authors went on to write a textbook on the subject of structured design [69, 70]. For the relationships between these three works, see Figure 4 (the three works cited in Figure 4 are hereafter referred to as the original works). Both books contained a chapter on coupling and a chapter on cohesion. [69] made no reference to any form of cognition; [70] contained 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' [52]. 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 [14]) that the ideas expressed in the original paper and the later books come from nearly ten years of observing programmers by Constantine. Though later criticized for a lack of theory and rigor (e.g. [71, 72]), the original works set out in Figure 4 represent the roots of the first paradigm in coupling and cohesion; later works on coupling and cohesion start from one or more of these three original works.

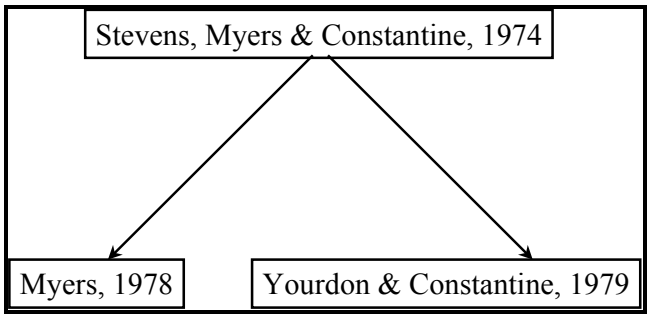

Figure 4: The seminal works on coupling and cohesion

Coupling was defined in the original paper as "the measure of the strength of association established by a connection from one module to another" ([14] p. 117). 'Relatedness' of program parts is how the concept is widely understood [73]. A categorical scale for the type of association or binding was outlined. Figure 5 lists the coupling categories, from the least desirable first to the most desirable, using an ordinal scale (i.e., levels)

Level	Description
Content	Where one module branches into, changes data or alters a statement in another module.
Common	Where two modules access the same global variable.
External	Where two modules access heterogeneous global data.
Control	Where one module passes a parameter to a second module to control behavior in the second module.
Stamp	Where two modules accept the same record as a parameter.
Data	Where two modules communicate via parameters.
None	Where none of the above events occur between two modules.

Figure 5: Coupling levels (based on [70])

Cohesion was originally described as binding - "Binding is the measure of the cohesiveness of a module" ([14], p. 121). Cohesion subsequently replaced binding as the term used for this type of software complexity [19]. The original definition and subsequent treatments of this form of software complexity rely on the notion of 'togetherness' of processing elements within a module to define cohesion. Figure 6 lists the cohesion types. Once again, an ordinal scale is provided, with the least desirable binding type listed first. The list is the same as that included in the original paper, except that both of the later books added the procedural category to the first list.

Level	Description
Coincidental	No association between two processing elements.
Logical	At each invocation of the module, one of the elements is executed.
Classical	Weak association among sequential elements.
Procedural	Both elements are sequentially part of the same iteration or operation.
Communicational	Both elements use the same input and/or output data set.
Sequential	The output of one element provides the input to another.
Functional	The elements perform a single specific function.

Figure 6: Cohesion levels (based on [70])

2) Measures for the OO Programming Paradigm

The OO programming paradigm has recently emerged as a possible replacement for the procedural programming paradigm, even if its adoption has been slower than expected [74]. Because the original coupling and cohesion works were couched in procedural terms, the OO paradigm needed measures that specifically considered the differences between the two paradigms [75]. Early approaches to software measure specification, including the original works, came under significant criticism due to lack of theory [18, 76]. At around the same time, several techniques for more rigorous software measure validation were also published (e.g. [8, 36, 77]). As is healthy at this stage of a discipline's growth, even some of these rigorous approaches to measure development have themselves been the subject of some criticism [78-80].

The combination of 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 [81, 82]. The suite contained one cohesion measure (LCOM) and two coupling measures (CBO & RFC). Since their publication, the CK suite in general, and the cohesion and coupling measures in particular, have been the subject of extensive discussion and refinement (e.g. [15, 83-85]). The essential characteristics of coupling and cohesion - relatedness and togetherness, respectively - are retained by the CK measures of coupling and cohesion.

Two recent OO software measurement reviews [86, 87] listed 13 OO cohesion and 30 OO coupling measures. The majority of the measures listed are based on the CK suite. As such, the CK suite represents the first major departure from work based on the categories presented in Tables 1 and 2. The CK suite represents a new coupling and cohesion paradigm that has arisen not simply as a result of a shift towards the OO programming paradigm. It is a development that has arisen from increasing the rigor in software measure specification and validation, a development to be welcomed. Moreover, the suite has also been extensively empirically validated. For example, recent empirical work on a commercial system revealed that among all of the listed measures, the original CK CBO and LCOM measures had some of the highest correlations to the fault proneness of a class [88].

3) Empirical Research on Coupling and Cohesion

Without clear theoretical guidance on which software complexity measures to apply, it has proven difficult to distinguish coupling and cohesion results from other software measurement results. Nor has much of the research exclusively pursued just coupling and cohesion results. Nevertheless, some empirical work exists. The three works cited in Figure 4 form the basis for most coupling and cohesion measurement research centered in the procedural programming paradigm. Subsequent measure development work is chronologically listed in Figure 7.

Cite	Measures	Empirical Data
[73]	4 coupling measures (data bindings)	2 medium commercial systems
[90]	5 coupling measures	None reported
[91]	Combined measure including 1 coupling and 1 cohesion measure	1 medium Unix utility
[92]	1 cohesion and 5 coupling measures	Textbook system
[93]	8 measures for both coupling and cohesion	1 medium system
[72]	11 measures of coupling	2 small and 3 medium systems
[71]	7 cohesion measures (for Table 2 levels)	None reported
[19]	1 measure for functional cohesion	5 sample procedures were analyzed

Figure 7: Coupling and Cohesion Measure Development for the Procedural Paradigm

Figure 7 represents the majority of published work using measures of coupling and cohesion for the procedural programming paradigm to be found in major SE journals. As can be seen from the table, despite the relevance of the problem, most of the work is observational or conceptual in nature, with only a small amount of work done to empirically validate various measures. This is an unfortunate outcome given the 10 years of observation that contributed to the

original coupling and cohesion levels outlined in Figure 5 and Figure 6 and another 10 to 15 years refining the levels and providing measures for each level. In total, the table shows two papers with no empirical work and the other 6 papers analyze a grand total of 7 medium commercial systems and some smaller examples. The table lists more measures than systems analyzed using those measures. Nevertheless, there is still more than enough procedural work (e.g. COBOL) for these measures to play an important role.

In another paper that specifically uses coupling and cohesion to restructure code to improve its quality, [89] also determined that such work yielded better code at the end of the analysis. It should be noted that both of these works are based in the procedural programming paradigm. So far, such work has not yet significantly impacted the structuring or restructuring of classes and objects, methods and instance variables in the OO programming paradigm.

The OO programming paradigm, given its newer claim to fame than the procedural programming paradigm, has received more recent research attention. In [34], it was shown that coupling and cohesion were consistently useful predictors across three different situations (design, coding and rework). This result was particularly striking in that the development of the prediction models was done using stepwise regression, allowing size and 4 out of the 6 CK measures to enter the solution. Though the three different systems analyzed were of a medium size, such industrial data is quite difficult to come by.

Other work has set out to more exclusively use coupling and cohesion and consider their usefulness as structural complexity measures and predictors of design quality. For example, in [94], the author used the notion of a concept to achieve lower coupling and higher cohesion to better restructure software modules. 20 applications were tested (10 public domain and 10 commercial); a measure of 'distance' from the original structure to the new structure was calculated as well as the cost of the restructuring. Ultimately, concept analysis proved worth the cost. This point is important; [41] makes it clear that there are only certain instances where the effort of restructuring is worth the cost of performing the restructuring.

D. Summary

As a first step in developing more rigorous theory underlying the importance of coupling and cohesion, task complexity was analyzed using Wood's model. His model tells us that there are three parts to task complexity: component, coordinative and dynamic. The information processing approach that dominates the program comprehension literature proposes that we comprehend in chunks that are of variable size, but that the total number of chunks that individuals can simultaneously utilize is quite small. There are two main activities to programming: designing an abstract solution and coding that abstract design. In both cases, the general approach taken is to divide and conquer the overall problem and solution into more manageable parts. There are many ways in which the problem can be divided into parts; different ways will require a different set of parts and different linkages among the parts. Examining these complexities leads us to the concepts of coupling and cohesion. The historical introduction and development of these measures goes all the way back to very early SE research. Their first domain was in the procedural programming paradigm. Several levels of each concept were proposed in 1974 and that proposal is the seminal work for the procedural programming paradigm. The proposal was based on a decade of observation of real programmers. The second coupling and cohesion measurement paradigm was more rigorous in its development and can be found in the OO programming paradigm. It should be noted that although the two measurement paradigms are approximately based in the two programming paradigms, the overlap of each measurement paradigm and its corresponding programming paradigm was achieved more by accident than any conscious design. Also, development of coupling and cohesion measures is a more dominant research activity than empirically validating them. The issues introduced in this section are considered further in the next section, ultimately leading to the development of an integrated model.

III. Research Model

The first subsection describes how coupling and cohesion are measures of software complexity analogous to Wood's coordinative and component measures of general task complexity, respectively. Some suggestions on how to deal with Wood's third source – dynamic complexity – are also made in this section. The first section is completed using an example to demonstrate how coupling and cohesion tap complexity above and beyond that of size and given problem complexity. The second section proceeds to take what was outlined in the theoretical background and create propositions that could, ultimately, be tested. The section culminates in the presentation of a research model based on the propositions.

A. Coupling and Cohesion as Measures of Complexity

It is necessary at this point to make parallels from the general terminology of Wood's model to the specific domain of this paper: software complexity. 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 a class). 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 are, to use Bieman's terminology, data tokens [19]. To use Halstead's terminology, information cues are operands [95]. A case could be made that operators could also be considered information cues; however, a closer examination reveals that operators are to be considered part of the act itself as they contribute to the outputs of a program unit but are not themselves outputs.

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 thought on component complexity was to simply calculate the number of distinct acts. 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.

In the software domain, an act is a program unit and information cues are the data tokens specific to that program unit. Component complexity is a function of the information cues in each act, summed over all acts required to complete the task. Cohesion is a measure of the "togetherness" of a single program unit or act as it is in Wood's terms. In other words, at the level of a single program unit, cohesion is equivalent to component complexity. In addition, Wood's model gives us the means to scale up from considering cohesion at the level of a single program unit to consideration at the level of the whole program.

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 program unit. Again, Wood's analysis allows us to generalize from consideration of a single program unit to the whole program.

Dynamic complexity is a function of the change in component and coordinative complexity over the span of completion for the task. In software terms, dynamic complexity can be evaluated in the light of the changes in cohesion and coupling and cohesion over time. Because measurement of coupling and cohesion is generally performed on static code that has been excerpted at some point during the implementation, usually the end, the dynamic complexity source serves little utility for the current discussion. Following the recommendations in Wood's model, one way to analyze dynamic complexity would be to follow an implementation over its lifecycle, thus providing insight into the correlation of coupling and cohesion measures over time. Although longitudinal studies on software over its lifecycle are rare, research such as [9], [96] and [97] provide some basic building blocks for carrying out future research into the dynamic source of software complexity.

In more general terms, examination of dynamic complexity could yield results from another path. True task dynamism would address the dynamics of program execution (and consequent program unit interaction) as a program was being run. However, such an assessment could only be made in a subjective manner, as every run of a program is likely to be different from previous and future runs even across the same user. As such, each individual run could be considered a unique task, separate in identity from other runs. If we consider the purpose of analyzing program structure, it is straightforward to conclude that comprehension only needs to consider the static code at the time of the comprehension task.

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 any or much further. Lower coupling and higher cohesion are taken as design and implementation goals [19]. For both concepts, it is recognized that it is not desirable or feasible to completely remove all coupling or have complete cohesion. Therefore, difficulties arise in deriving or selecting an acceptable level for either measure.

Implementation will add complexity, above and beyond that of the complexity inherent in the design. At implementation time, a decision of where to place a code segment impacts both coupling and cohesion. Structural complexity can be significantly altered based on the placement of a code segment but size, in terms of the number of lines of code, is hardly affected. In other words, coupling and cohesion tap different structural complexity factors other than size, a point that is further developed in the next section.

B. Software Complexity: A Research Model

At this point, it is useful to review the two research questions this paper set out to explore. The first question speaks to choosing a subset of software complexity measures. There are several issues implicit in the question. Three of the most salient issues are discussed in this section. The first issue deals with the motivation for the question. As stated in the introduction, no individual programmer or software project manager can usefully interpret the many messages that could be taken from the diverse set of existing complexity measures. SE researchers and practitioners need a way to focus on some particular subset of measures so that we can usefully proceed in evaluating, predicting and controlling software complexity.

The second issue inherent in the first research question is how to determine membership of the chosen subset of measures. SE literature has strongly spoken out against the selection of a single measure, but so far, that is largely their only proclamation on the matter. On the one hand, it is clear that we need more than one measure. On the other hand, we also know that we need a set that is considerably smaller than the existing broad and multitudinous set of measures.

The third issue is tied to how to select the "hallowed" subset of measures. This paper makes the choice on theoretical grounds. If we consider software activities such as design, implementation and maintenance to be tasks, then like any task, software tasks are subject to analysis by the models and methods of task analysis. Wood's model is a general mechanism that has been applied to analyzing tasks, including software tasks. Based on the analysis presented in the

previous subsection, it should be clear to the reader that in the domain of software complexity, the software measurement categories of coupling and cohesion are equivalent to the coordinative and component sources of task complexity. As such, these two concepts are sufficient to encompass much of the extent in variability for software structure. Other issues, particularly those in the environment, will also impact software tasks. However, the focus of this paper is on how to evaluate and control the structural complexity of software within a given environment.

This brings us to the second research question. Given that we have used Wood's task complexity model to choose the two software measures of coupling and cohesion, the second research question focuses on how to resolve conflicts and make tradeoffs between these two measures. It is at this point, that we draw once again from cognition literature, particularly with regard to the information processing perspective on cognition and its fundamental concepts of STM, LTM and chunks.

One dilemma that can now be resolved is in making tradeoffs between reducing coupling and increasing cohesion. Further examination of cognition enables resolution of the tradeoff. Little empirical evidence and few theoretical propositions exist to guide resolution of the issue.

This work posits that complexity grows at a much faster rate for code that increases coupling compared to code that reduces cohesion. If a module is highly coupled, then the understanding of all of the other coupled modules quickly eats up available chunks in STM. Looking at it in another fashion, as coupling increases, a typical unit of program analysis (e.g. module, function, procedure, object, etc.) is added to the chunk load to be comprehended by the programmer. If the added part is low in cohesion, the comprehension load is increased by a larger than average unit of program analysis. This is because if the added part is low in cohesion then it may represent several chunks rather than the single chunk that a part that has high cohesion. However, the rate of growth for chunk use is much greater as coupling increases (adding chunks) than as cohesion increases (increasing the size of a chunk). Not all chunks are equal and the 'quality' (as captured by cohesion) of a chunk added to the comprehension stack matters. Wood's model values coordinative (coupling) complexity higher than component (cohesive) complexity.

Consider the cases illustrated by Figure 8. If a programmer needs to comprehend program unit 1, then the programmer must also have some understanding of the program units that program unit 1 is coupled to. 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 1 more chunk is added to the comprehension stack (given that program unit 2 also shows high cohesion). If program unit 1 is also coupled to program unit 3, because program unit 3 shows low cohesion and thus represents several chunks, the comprehension stacks gets filled up much quicker. But the primary driver of the size of the comprehension stack is the extent to which program unit 1 is coupled to other units. If coupling is evident, it is only then that the extent of cohesion becomes a comprehension issue. This leads to Proposition 1

P1: For more highly coupled programs, comprehension performance decreases.

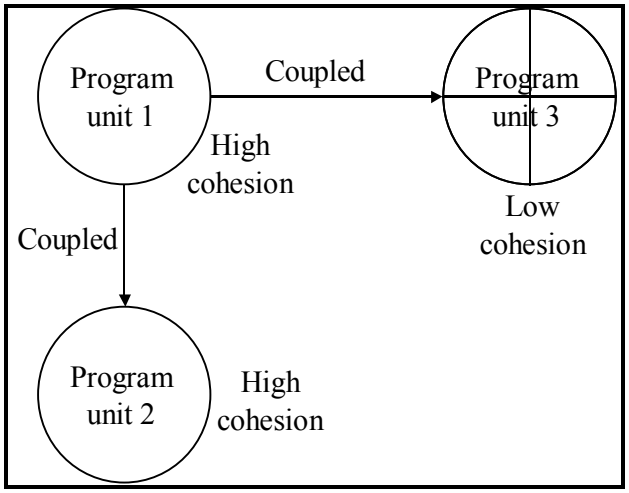

Figure 8: Interaction of coupling and cohesion

At first glance, cognition and task complexity appear to diverge on the effect of coupling and cohesion. Yet the notion of limited cognitive resources clearly implies a joint effect for coupling and cohesion on comprehension. In Wood's model, the three components of task complexity are presented as independent from one another and having an additive effect on total task complexity. However, Wood later recognizes possible interdependencies between the components in his model.

It is the central theme of this work that we cannot ignore the joint effect of coupling and cohesion on cognition. In the SE literature, coupling and cohesion measures are generally examined separately. Even in confirmatory work, coupling and cohesion measures are usually tested independently. Although we can analytically examine coupling and cohesion independently, their effect on complexity is actually interactive. The main effect for cohesion prevalent in the SE literature is actually a joint effect with coupling rather than a main effect by itself. This brings us to Proposition 2:

P2: For more highly coupled programs, higher levels of cohesion increase comprehension performance.

Building on the preliminary analysis presented in the previous section, the concept of structural complexity is proposed. Structural complexity includes coupling and cohesion. The relationship between coupling and cohesion is shown as an interactive one with coupling being the driver of comprehension performance. The model is shown in Figure 9.

Answering the first research question allowed us to use the coupling, cohesion and complexity conceptual boxes. Answering the second question allowed us to draw the arrows or relationships between the concepts, leading to the model proposed in Figure 9.

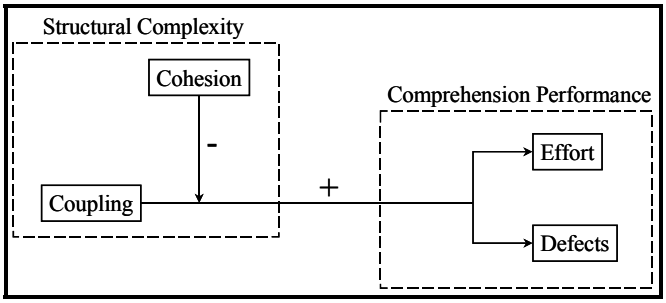

Figure 9: Research Model

It should be noted that the model is presented without naming a specific programming paradigm or language as the underlying context. Given that the model is drawn from the hypotheses and the hypotheses are also presented independently of any program paradigm or language, proposition 3 should come as no surprise to the reader. This proposition is controversial. From section 2.1, it is clear that such a stance - assuming the same effort for coupling and cohesion over procedural and OO programming paradigms - runs counter to current thinking.

P3: Any programming paradigm or language will show the same impact of changing coupling and cohesion on comprehension performance.

In other words, P1 and P2 will apply regardless of whether the programming paradigm is procedural in nature and the language is C or COBOL, etc. or the paradigm is OO in nature and the language is C++ or Java, etc. Similarly, P1 and P2 will also apply to future programming paradigms and languages so long as the general approach of the future programming paradigms and languages is to 'divide and conquer' the overall problem into smaller parts. Given that the tendency is to solve problems by dividing them into smaller parts long before the solution is programmed, P1 and P2 should have a considerable ubiquity for some time to come.

C. *Alternative Models*

There are several ways in which the model presented in Figure 9 could be reconfigured. For example, coupling has been hypothesized as the main effect with cohesion moderating that relationship. A test of this hypothesis (such as through ANOVA) will only reveal that coupling and cohesion are statistically interrelated. The test will not reveal which of coupling or cohesion is the main relationship and which is the moderator. The distinction between the main relationship and the moderator comes from the theoretical exposition of the model. Examining the relative statistical effect of both coupling and cohesion can also bolster the distinction.

A more radical departure from the research model shown in Figure 9 would be a model where cohesion's effect on comprehension is mediated by coupling. The problem with such a model is that, by definition, a mediated effect of a variable can only happen in the presence of the mediating variable. In that case, the implication is that the effect of cohesion can only happen through coupling. If no coupling were present, then there would be no effect of cohesion either. However, in such a case, for example where a program was entirely contained in only a single procedure, cohesion would still make a significant difference on comprehension performance. It is more likely that in this alternative conception, cohesion will have its own main effect on comprehension performance in addition to its mediated effect through coupling.

Such a conception would be similar to that of the ease of use construct that can be found in the technology acceptance model (TAM). In TAM, usefulness has a main effect on intention to use and is the largest single effect in the model. However, the model also includes a main effect for ease of use and a mediated effect for ease of use through usefulness (Davis, 1989). In the case of the mediated model, coupling would be equivalent to usefulness, representing the main and strongest effect with cohesion having a small but direct effect as well as a mediated effect through coupling.

IV. Conclusions

It is a fundamental premise of this work that coupling and cohesion have merit above and beyond many other software measures. Acceptance of this premise opens up new horizons for software complexity measurement. Future research efforts can then be focused 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 bringing programming classes beyond lessons in syntax [98, 99]. Specifically, coupling and cohesion have historically been described and discussed; however, it has never been quite this clear just how important they are as essential indicators of software complexity. In addition, coupling and cohesion have been conventionally considered as independent concepts, something that is under challenge in the analysis and proposed model.

In closing, it should be clear to the reader that the concepts of coupling and cohesion are well developed and explored. In addition, the concepts are exceptionally well endowed with operationalizations and measures. By the time the next divide and conquer programming paradigm is started, coupling and cohesion will be well prepared to lend a hand in evaluation and prediction. What is far less clear is the direction that should be taken in further exploring measurement of these concepts. We need to take stock of all 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.

ACKNOWLEDGMENTS

The preferred spelling of the word "acknowledgment" in American English is without an "e" after the "g." Use the singular heading even if you have many acknowledgments. Avoid expressions such as "One of us (S.B.A.) would like to thank ... ." Instead, write "S.B.A. thanks ... ." Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

[1] D. J. Campbell, "Task Complexity: A Review and Analysis," *AMR*, vol. 13, pp. 40-52, 1988.

[2] N. Gross and S. Hamm, "Industry Outlook: Software," in *BusinessWeek*, 1999.

[3] D. Wessel, "NASA explores the future of software," in *Wall Street Journal*, 2001, pp. 1.

[4] E. W. Dijkstra, "[Letter] GOTO Statement Considered Harmful," *CACM*, vol. 11, pp. 147, 1968.

[5] Parnas, "A Technique for Software Module Specification with Examples," *CACM*, vol. 15, pp. 330-336, 1972.

[6] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *CACM*, vol. 15, pp. 1053-1058, 1972.

[7] L. J. Waguespack and S. Badlani, "Software Complexity Measurement: An Introduction and Annotated Bibliography," *ACM SIGSOFT*, vol. 12, pp. 52-71, 1987.

[8] H. Zuse and P. Bollman, "Software Metrics: Using Measurement Theory To Describe the Properties and Scales of Static Software Complexity Measures," *SIGPLAN Notices*, vol. 24, pp. 23-33, 1990.

[9] C. F. Kemerer, "Software Complexity and Software Maintenance: A Survey of Empirical Research," *Annals of Software Engineering*, vol. 1, pp. 1-22, 1995.

[10] J. Eder, G. Kappel, and M. Schreff, "Coupling and Cohesion in Object-Oriented Systems," University of Linz, Linz, Austria 0293, 1994.

[11] T. DeMarco, *Controlling Software Projects*. Englewood Cliffs, NJ: Prentice Hall, 1982.

[12] I. Vessey and R. Weber, "Some Factors Affecting Program Maintenance: An Empirical Study," *CACM*, vol. 26, pp. 128-134, 1983.

[13] I. Vessey and R. Weber, "Research on Structured Programming: An Empiricist's Evaluation," *IEEE TSE*, vol. 4, 1984.

[14] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, pp. 115-139, 1974.

[15] M. Hitz and B. Montazeri, "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE TSE*, vol. 22, pp. 267-271, 1996.

[16] 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, pp. 1405-1431, 2000.

[17] J. K. Kearney, R. L. Sedimeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software Complexity Measurement," *CACM*, vol. 29, pp. 1044-1050, 1986.

[18] A. L. Baker, J. M. Beiman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty, "A Philosophy of Software Measurement," *JSS*, vol. 12, pp. 277-281, 1990.

[19] J. M. Bieman and L. M. Ott, "Measuring Functional Cohesion," *IEEE TSE*, vol. 20, pp. 644-657, 1994.

[20] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham, "Status Report on Software Measurement," *IEEE Software*, pp. 33-43, 1997.

[21] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE TSE*, vol. 24, pp. 491-496, 1998.

[22] N. E. Fenton and M. Neil, "Software Metrics: Successes, Failures and New Directions," *The Journal of Systems and Software*, vol. 47, pp. 149-157, 1999.

[23] M. G. Mendonca and V. R. Basili, "Validation of an approach for improving existing measurement frameworks," *TSE*, vol. 26, pp. 484-499, 2000.

[24] M. Shin and A. L. Goel, "Empirical data modeling in software sngineering using radial basis functions," *TSE*, vol. 26, pp. 567-576, 2000.

[25] N. E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE TSE*, vol. 25, pp. 675-689, 1999.

[26] J. E. Smith, "Characterizing Computer Performance with a Single Number," *CACM*, vol. 31, pp. 1202-1206, 1988.

[27] B. A. Kitchenham, S. L. Pfleeger, and N. E. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE TSE*, vol. 21, pp. 929-943, 1995.

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

[29] R. E. Wood, "Task Complexity: Definition of the Construct," *OBHDP*, vol. 37, pp. 60-82, 1986.

[30] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study," *MS*, vol. 44, pp. 433-450, 1998.

[31] C. F. Kemerer, "Progress, Obstacles, and Opportunities in Software Engineering Economics," *CACM*, vol. 41, pp. 63-66, 1998.

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

[33] H. Lee, "Automatic Clustering of Business Processes in Business Systems Planning," *European Journal of Operational Research*, vol. 114, pp. 354-362, 1999.

[34] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object oriented software: an exploratory analysis," *IEEE TSE*, vol. 24, 1998.

[35] H. A. Simon, "The Structure of Ill-Structured Problems," *Artificial Intelligence*, vol. 4, pp. 181-201, 1973.

[36] A. C. Melton, D. A. Gustafson, J. M. Bieman, and A. L. Baker, "A Mathematical Perspective for Software Measures Research," *SEJ*, pp. 246-254, 1990.

[37] M. B. Rosson, "Human Factors in Programming and Software Development," *ACM Computing Surveys*, vol. 28, pp. 193-195, 1996.

[38] S. A. Whitmire, *Object-Oriented Design Measurement*. New York, NY: John Wiley & Sons, Inc., 1997.

[39] C. J. Hemingway, "Towards a Socio-Cognitive Theory of Information Systems: An Analysis of Key Philosophical and Conceptual Issues," presented at IFIP Working Group 8.2 and 8.6, Helsinki, 1998.

[40] J. Kim and F. Lerch, "Cognitive Process in Logical Design: Comparing Object Oriented Design and Traditional Functional Decomposition Methodologies," presented at CHI 92 Human Factors in Computing Systems, 1992.

[41] R. D. Banker and S. A. Slaughter, "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement," *ISR*, vol. 11, pp. 219-240, 2000.

[42] H. A. Simon, *Models of Thought: Volume I*. New Haven, Connecticut: Yale University Press, 1979.

[43] M. C. MacDonald, M. A. Just, and P. A. Carpenter, "Working Memory Constraints on the Processing of Syntactic Ambiguity," *Cognitive Psychology*, vol. 24, pp. 56-98, 1992.

[44] M. Shaw, "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, vol. 1, pp. 10-26, 1984.

[45] M. B. Rosson and S. R. Alpert, "The Cognitive Consequences of Object-Oriented Design," *Human Computer Interaction*, vol. 5, pp. 345-379, 1990.

[46] K. R. Koedinger and J. A. Anderson, "Abstract Planning and Perceptual Chunks: Elements of Expertise in Geometry," *Cognitive Science*, vol. 14, pp. 511-550, 1990.

[47] A. G. Sutcliffe and N. A. M. Maiden, "Analyzing the Novice Analyst: Cognitive Models in Software Engineering," *IJMMS*, vol. 36, pp. 719-740, 1992.

[48] S. N. Cant, D. R. Jeffery, and B. Henderson-Sellers, "A Conceptual Model of Cognitive Complexity of Elements of the Programming Process," *Information and Software Technology*, vol. 37, pp. 351-362, 1995.

[49] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc, 1983.

[50] H. A. Simon, *Models of Thought: Volume II*. New Haven, Connecticut: Yale University Press, 1989.

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

[52] G. A. Miller, "The magic number seven plus or minus two: some limits of our capacity for information-processing," *Psychological Review*, vol. 63, pp. 81-87, 1956.
[53] W. G. Chase and H. A. Simon, "Perception in Chess," *Cognitive Psychology*, vol. 4, pp. 55-81, 1973.
[54] S. P. Davies, "The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior," *Cognitive Science*, vol. 15, pp. 547-572, 1991.
[55] H. A. Simon, "How Big is a Chunk?," *Science*, vol. 183, pp. 482-488, 1974.
[56] J. S. Davis, "Chunks: A Basis for Complexity Measurement," *IPM*, vol. 20, pp. 119-127, 1984.
[57] H. Buschke, "Learning is Organized by Chunking," *Journal of Verbal Learning and Verbal Behavior*, vol. 15, pp. 313-324, 1976.
[58] J. S. Reitman and H. H. Reuter, "Organization Revealed by Recall Orders and Confirmed by Pauses," *Cognitive Psychology*, vol. 12, pp. 554-581, 1980.
[59] K. B. McKeithen, J. S. Reitman, H. H. Reuter, and S. C. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology*, vol. 13, pp. 307-325, 1981.
[60] B. Curtis, "Fifteen Years of Psychology in Software Engineering,"., 1985, pp. 438-453.
[61] B. Curtis, "Five Paradigms in the Psychology of Programming," in *Handbook of Human-Computer Interaction*, M. Helander, Ed. Amsterdam: North Holland, 1988.
[62] A. von Mayrhauser and M. A. Vans, "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer*, vol. 12, pp. 44-55, 1995.
[63] M.-A. D. Storey, F. D. Fracchia, and H. A. Muller, "Cognitive Design Elements to Support the Construction of a Mental Model During the Software Exploration," *JSS*, vol. 44, pp. 171-185, 1999.
[64] W. J. Tracz, "Computer Programming and the Human Thought Process," *Software Practice and Experience*, vol. 9, pp. 127-137, 1979.
[65] C. Pair, "Programming, Programming Languages and Programming Methods," in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay, and F. J. Gilmore, Eds. New York: Academic Press, 1993, pp. 9-19.
[66] T. R. G. Green, "The Nature of Programming," in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay, and F. J. Gilmore, Eds. New York: Academic Press, 1993, pp. 21-44.
[67] N. Pennington and B. Grabowski, "The Tasks of Programming," in *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay, and F. J. Gilmore, Eds. New York: Academic Press, 1993, pp. 45-62.
[68] F. P. Brooks, "No silver bullet: essence and accidents of software engineering," *Computer*, vol. 20, pp. 10-19, 1987.
[69] G. J. Myers, *Composite/Structured Design*. New York, NY: Van Nostrand Reinhold, 1978.
[70] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1979.
[71] A. Lakhotia, "Rule-Based Approach to Computing Module Cohesion," presented at 15th International Conference in Software Engineering, 1993.
[72] A. J. Offutt, M. J. Harrold, and P. Kolte, "A Software Metric System for Module Coupling," *JSS*, vol. 20, pp. 295-308, 1993.
[73] D. H. Hutchens and V. R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE TSE*, vol. 11, pp. 749-757, 1985.
[74] R. G. Fichman and C. F. Kemerer, "Adoption of software engineering process innovations: the case of object orientation," *SMR*, vol. 34, pp. 7-22, 1993.
[75] F. B. e Abreu and R. Carapuca, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *JSS*, vol. 26, pp. 87-96, 1994.
[76] N. Fenton, *Software Metrics-A Rigorous Approach*, 1st ed. London: Chapman Hall, 1991.
[77] E. J. Weyuker, "Evaluating Software Complexity Measures," *TSE*, vol. 14, pp. 1357-1365, 1988.
[78] J. C. Cherniavsky and C. H. Smith, "On Weyuker's Axioms for Software Complexity Measures," *TSE*, vol. 17, pp. 636-638, 1991.
[79] B. A. Kitchenham and J. G. Stell, "The Danger of Using Axioms in Software Metrics," *IEEE Proceedings*, 1997.
[80] F. Xia, "Look Before You Leap: On Some Fundamental Issues in Software Engineering Research," *Information and Software Technology*, vol. 41, pp. 661-672, 1999.
[81] S. R. Chidamber and C. F. Kemerer, "Towards a Metrics Suite for Object Oriented Design," presented at Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Pheonix, AZ, 1991.
[82] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476-493, 1994.
[83] N. I. Churcher and M. J. Shepperd, "Comments on "A Metrics Suite for Object Oriented Design"," *IEEE TSE*, vol. 21, pp. 263-638, 1995.
[84] S. R. Chidamber and C. F. Kemerer, "Authors' Reply [to Churcher and Sheppard, 1995]," *IEEE Transactions on Software Engineering*, vol. 21, pp. 265, 1995.
[85] L. Etzkorn, C. Davis, and W. Li, "A Practical Look at the Lack of Cohesion in Methods Metric," *JOOP*, pp. 27-34, 1998.
[86] L. C. Briand, J. W. Daly, and J. K. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *ESE*, vol. 3, pp. 65-117, 1998.
[87] L. C. Briand, J. W. Daly, and J. K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE TSE*, vol. 25, pp. 91-121, 1999.
[88] L. C. Briand, J. K. Wust, S. V. Ikonomovski, and H. Lounis, "Investigating Quality Factors in Object Oriented Designs: an Industrial Case Study," presented at International Conference on Software Engineering, Los Angeles, CA, 1999.
[89] B.-K. Kang and J. M. Bieman, "A Quantitative Framework for Software Restructuring," *Journal of Software Maintenance*, vol. 11, pp. 245-284, 1999.
[90] N. E. Fenton and A. Melton, "Deriving Structurally Based Software Measures," *JSS*, vol. 12, pp. 177-187, 1990.
[91] R. Adamov and L. Richter, "A Proposal for Measuring the Structural Complexity of Programs," *JSS*, vol. 12, pp. 55-70, 1990.
[92] D. Tesch and G. Klein, "Optimal Module Clustering in Program Organization," *IEEE*, pp. 238-245, 1991.
[93] L. Rising and F. W. Calliss, "Problems with Determining Package Cohesion and Coupling," *SPE*, vol. 22, pp. 553-571, 1992.
[94] P. Tonella, "Concept analysis for module restructuring," *TSE*, vol. 27, pp. 351-363, 2001.
[95] M. H. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.
[96] V. Rajlich, "Modeling software evolution by evolving interoperation graphs," *ASE*, vol. 9, pp. 235-248, 2000.
[97] B. A. Nejmeh, "NPATH: A Measure of Execution Path Complexity and Its Applications," *CACM*, vol. 31, pp. 188-200, 1988.
[98] M. Russell, "International Survey of Software Measurement Education and Training," *JSS*, vol. 12, pp. 233-241, 1990.
[99] M. J. Bowman and W. A. Newman, "Software Metrics as a Programming Training Tool," *JSS*, vol. 13, pp. 139-147, 1990.

David P. Darcy Other usual biography information includes birth date and place, education, employments, and memberships in other professional societies.

Chris F. Kemerer