COUPLING AND COHESION

Shikha Gautam
SITM, Lucknow

Rekha Singh
SITM, Lucknow

S. B. Singh Yadav
BBDNITM, Lucknow

Abstract

This research examines the structural complexity of software, and specifically the potential interaction of the two most important structural complexities: coupling and cohesion. Coupling and Cohesion are the two terms which very frequently occur together. Together they talk about the quality a module should have. Coupling talks about the interdependencies between the various modules while cohesion describes how related functions within a module are. Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large. Thus, Cohesion represents how tightly bound the internal elements of a module are to one another. Coupling is the nature of degree of independence between modules. Its measures by the no. of interconnection between modules. High cohesion and low coupling is a criterion for good software design.

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 software complexity management.

Keywords - Software complexity, software structure, task complexity, coupling, cohesion.

1. Introduction

Few activities are as complex as the effective design, implementation, and maintenance of software. Since 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 supra-linear increases in software complexity, and increases in software complexity may lead to supra-linear impacts on managerial outcomes of interest, such as increasing the effort to design, implement, and maintain software and reducing its quality.

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.

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, the more flexible are those choices. 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.

2. 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.

3. Conceptual Background

It is widely believed that software complexity cannot be described using a single dimension. The search for a single, all encompassing dimension has been likened to the “search for the Holy Grail”. 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 or adapting, refining, and/or improving existing factors. However, although useful, neither of these two approaches enables a direct answer to what are the important software complexity characteristics. In fact, the SE literature is replete with calls for theoretical guidance on this issue.

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. Wood’s model is generally representative of task complexity approaches, but it is more closely studied in this paper because it has already been found to be useful in a software maintenance context. 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.

A. Wood’s Task Complexity Model

Wood’s Task Complexity Model considers tasks to have three essential concepts: 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”. An act is “the pattern of behaviors with some identifiable purpose or direction”. 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”. 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”. Coordinative complexity covers the “nature of relationships between task inputs and task products”. The form, the strength and the sequencing of the relationships are all considered 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”. 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. Total task complexity is then a function of the three types of task complexity.

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 (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. 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.

A fundamental approach to improving software development has been to modularize the design by splitting the implementation of the solution 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 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.

4. Coupling and Cohesion

This subsection is separated into three 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
2) Measures for the OO Programming Paradigm
3) Empirical Research on Coupling and Cohesion

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

Figure 1: Coupling levels

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

Figure 2: Cohesion levels
A. Measures for the Procedural Programming Paradigm

Coupling was defined in the original paper as “the measure of the strength of association established by a connection from one module to another”. ‘Relatedness’ of program parts is how the concept is widely understood. A categorical scale for the type of association or binding was outlined. Figure 1 lists the coupling categories, from the least desirable first to the most desirable, using an ordinal scale.

Cohesion was originally described as binding - “Binding is the measure of the cohesiveness of a module”. Cohesion subsequently replaced binding as the term used for this type of software complexity. 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 2 lists the cohesion types.

Table 1: Coupling and Cohesion in the Procedural Paradigm.

<table>
<thead>
<tr>
<th>Measures</th>
<th>Empirical Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 coupling measures (data bindings)</td>
<td>2 medium systems</td>
</tr>
<tr>
<td>5 coupling measures</td>
<td>None reported</td>
</tr>
<tr>
<td>Combined measure of coupling and cohesion</td>
<td>1 medium Unix utility</td>
</tr>
<tr>
<td>1 cohesion and 5 coupling measures</td>
<td>Textbook system</td>
</tr>
<tr>
<td>8 measures for coupling and cohesion</td>
<td>1 medium system</td>
</tr>
<tr>
<td>11 measures of coupling</td>
<td>2 small &amp; 3 medium systems</td>
</tr>
<tr>
<td>7 cohesion measures</td>
<td>None reported</td>
</tr>
<tr>
<td>1 measure for functional cohesion</td>
<td>5 sample procedures</td>
</tr>
</tbody>
</table>

B. 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. 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. Early approaches to software measure specification, including the original works, came under significant criticism due to lack of theory. At around the same time, several techniques for more rigorous software measure validation were also published. 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.

C. 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. Subsequent measure development work is chronologically listed in Table 1.

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.

5. 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. To use Halstead’s terminology, information cues are operands. 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.

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.

6. 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 tradeoffs 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. 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.

Complexity grows at a much faster rate for code that increases coupling as compared to code that reduces cohesion. Consider the cases illustrated by Figure 4.

![Figure 4: Interaction of coupling and cohesion](image)

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.

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.

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. 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. Proposition 3:

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.

7. 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 2 levels, high and low, producing a fully crossed 4-cell design as illustrated in Figure 4.

<table>
<thead>
<tr>
<th>Coupling (Cpl)</th>
<th>Cohesion (Coh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>HiCpl-LoCoh</td>
<td>HiCpl-HiCoh</td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>LoCpl-LoCoh</td>
<td>LoCpl-HiCoh</td>
</tr>
</tbody>
</table>

Figure 4: Experimental Design

8. 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. 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.

REFERENCES