Journal of Software Engineering and Applications, 2021, 14, 44-66
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Cyclomatic Complexity-Based Encapsulation, Data Hiding, and Separation of Concerns

Charles W. Butler, Thomas J. McCabe

Colorado State University, Fort Collins, USA
Email: Charles.Butler@colostate.edu

How to cite this paper: Butler, C.W. and McCabe, T.J. (2021) Cyclomatic Complexity-Based Encapsulation, Data Hiding, and Separation of Concerns. *Journal of Software Engineering and Applications*, **14**, 44-66.
https://doi.org/10.4236/jsea.2021.141004

Received: November 25, 2020
Accepted: January 18, 2021
Published: January 21, 2021

Abstract

Three design principles are prominent in software development-encapsulation, data hiding, and separation of concerns. These principles are used as subjective quality criteria for both procedural and object-oriented applications. The purpose of research is to quantify encapsulation, data hiding, and separation of concerns is quantified using cyclomatic-based metrics. As a result of this research, the derived design metrics, coefficient of encapsulation, coefficient of data hiding, and coefficient of separation of concerns, are defined and applied to production software indicating whether the software has low or high encapsulation, data hiding, and separation of concerns.

Keywords

Object-Oriented Design Methods, Reliability, Complexity Measures, Software Design, Encapsulation, Information Hiding, Separation of Concerns, McCabe Metrics, Coefficient of Encapsulation, Coefficient of Data Hiding, Coefficient of Separation of Concerns

1. Introduction

The importance of software development has never been more critical. Software is a driving force in product and information technology as it provides functionality for a broad range of computer platforms. Today, the sheer volume of software is immeasurable. Existing application portfolios grow annually through the introduction of new applications and maintenance of existing applications. An overriding force in this growth is software quality. When developing software, a software developer is aware of how critical high quality is to every software solution. Sustaining quality in today's large software inventories is an ongoing challenge.

Three proven design principles, encapsulation, data hiding and separation of concern, have a long and successful history in the software development discipline. They have been used to assist software developers when designing software and assessing the quality of the software design. When packaging functionality, software developers seek to encapsulate process with data so that the functionality is constructed logically correct and highly maintainable. When process and data are constructed to hide data, the design properties promote protected access and reduced risks to data. Further, when multiple software components are developed with separation of concerns, the influence across software components is less, and these components standalone without creating ripple effects that are often unknown or unpredictable. Software with good encapsulation, data hiding and separation of concerns are more desirable than the alternative—poor encapsulation, data hiding and separation of concern.

The problem with these design principles is their inheritance subjectivity. While software engineers seek to achieve good encapsulation, data hiding, and separation of concern, the assessment of these design principles is subjective. What is good? What is poor? Multiple software engineers can look at the same software target and assess the design quality to be poor, medium, or high. Currently, these design principles lack a fact basis for assessing the quality level. In this research, McCabe metrics are utilized to derive objective measures for the level of encapsulation, data hiding and separation of concern in a software application. Using these software metrics, a software engineer can achieve the following:

1) An objective measurement for encapsulation, data hiding, and separation;

2) A repeatable measurement for a targeted software application;

3) A measurement understandable by software engineers and management;

4) A measurement derived from a classical set of McCabe metrics that have recognition and creditability in the software engineering discipline.

2. Objectives

Software developers and engineers should consider proven design principles when designing and constructing software. Software design principles serve as the foundation for analytical reasoning regarding the quality of a software solution. A good software design facilitates the creation of software code and reduces the time to maintain the software while achieving execution performance. The objectives of this article include two elements. The first is to define three quantitative design metrics: encapsulation, data hiding, and separation of concerns coefficients. This quantification increases the value of these principles when creating and maintaining software. The second element is to illustrate these metrics using software examples and demonstrate how they provide valuable objective insight into software quality. To accomplish the stated objectives, McCabe metrics are applied to three proven design principles: encapsulation, data hiding, and separation of concern. Cyclomatic complexity, module design complexity, global data complexity, and specified data complexity for local data are used to quantify these design principles.

3. Object Oriented Software Metrics

Important research has been performed on software metrics. Metrics have been shown to predict faults [1] [2] [3] [4]. Other research has measured modularity [5]. Still other research has measured maintainability [6]. Three groups of metrics have become mainstream including those developed by Chidamber and Kemerer [7], Bansiya and Davis [8], and Tang, Kao, and Chen [9]. Table 1 contains a summary of the popular metrics authored by these researchers.

These software metrics are powerful, and the Chidamber and Kemerer metric suite has become the gold standard for object oriented metrics. Intuitively, they demonstrate measurable characteristics of an object oriented program. For the general design environment, they describe measurable attributes about complexity,

Table 1. Software metric descriptions.

Metric	Acronym	Description	Author
Number of public methods	NPM	A count of public methods in a class	Bansiya & Davis
Data access metric	DAM	The ratio of private attributes to the total attributes declared in a class	Bansiya & Davis
Measure of aggregation	MOA	A count of the number of class fields who types are user defined classes; measures the part-whole relationship	Bansiya & Davis
Measure of functional abstraction	MFA	The ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods on the class	Bansiya & Davis
Cohesion among methods of class	CAM	The summation of the number of different types of method parameters in every method divided by a multiplication of the number of different method parameter types in whole class and number of methods; the relatedness of methods of a class based on the parameter list of the methods	Bansiya & Davis
Weighted methods per class	WMC	The number of methods in the class	Chidamber & Kemerer
Depth of inheritance tree	WIT	The inheritance levels form the top object hierarchy top	Chidamber & Kemerer
Number of children	NOC	The number of immediate descendants of the class	Chidamber & Kemerer
Coupling between object classes	CBO	The number of classes coupled to a given class as a result of method calls, field accesses, inheritance, method arguments, return types, and exceptions	Chidamber & Kemerer
Response for a class	RFC	The number of different methods that can be executed when an object of that class receives a message	Chidamber & Kemerer
Lack of cohesion of methods	LCOM	The sets of methods a class that are not related through sharing of some of the class fields	Chidamber & Kemerer
Inheritance coupling	IC	The number of parent classes to which a given class is coupled; coupling occurs when 1) an inherited method uses an attribute that is defined in a new or redefined method, 2) one of its inherited methods calls a redefined method or 3) one of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined model	Tang, Kao & Chen
Coupling between methods	CBM	The total number of new or redefined methods to which all the inherited methods are coupled	Tang, Kao & Chen
Average method complexity	AMC	The average method size for each class; size is equal to the number of Java binary codes in the method	Tang, Kao & Chen

cohesion and coupling. A shortcoming is that they do not map directly to long-standing design principles—encapsulation, data hiding, and separation of concerns. In addition, these design principles apply to all software, not just object oriented code.

4. Design Characteristics

4.1. Popular Software Design Principles

As software is designed, there are well-established principles applied by software developers. These principles guide the development effort with the goal of creating quality software that is understandable, testable, and measurable. Three of the most popular design principles are encapsulation, data hiding, and separation of concerns. In Meilir Page-Jones' book, encapsulation is defined as [10]:

> The grouping of related ideas into one unit, which can thereafter be referred to by a single name.

In programming languages, encapsulation refers to one of two related but distinct notions, and sometimes to the combination thereof. First, it is a language construct that facilitates the bundling of data with the modules (or other functions) operating on that data. Secondly, it is language mechanism for restricting access to some of the object's components.

Sometimes, encapsulation and data hiding are used interchangeably. However, data (information) hiding is a software design principle originally introduced by David L. Parnas. In his paper written almost 40 years ago, Parnas used data hiding to discuss the criteria leading to good modularity in structured programming [11]:

> Every module in the [...] decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

Specifically, in object-oriented programming, data hiding shields internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. Decision logic that does not use public global and parameter data indicates strong data hiding (since other objects cannot access this data) and decision logic that uses public global and parameter data indicates weak data hiding (since other objects can access this data).

The term, separation of concerns, was first used by Edsger W. Dijkstra in his 1974 paper "On the Role of Scientific Thought". Dijkstra wrote that separation of concerns [12]:

> Even if not perfectly possible is yet the only available technique for effective ordering of one's thoughts.

Separation of concerns has grown into a software design characteristic for separating a program into objects such that each object addresses a separate concern. A concern is a logic or a set of information that affects the code of a program. Decision logic that connects modules and uses public global and parameter data indicates weak separation of concerns, since this logic and data is influencing the operation of one or more other objects.

Encapsulation, data hiding, and separation of concerns are design principles that are, sometimes, misunderstood and, at other times, misapplied. Figure 1 illustrates an important view of the overlap and uniqueness among these design principles. In this view, program and data logic elements are essential factors in determining encapsulation, data hiding, and separation of concerns. The intersection among encapsulation, data hiding, and separation of concerns represents principle overlap. High use of local data drives encapsulation; low use of global data promotes data hiding; low inter-module program logic establishes separation of concerns. Yet, there is interaction among these principles. When good encapsulation is achieved, software exhibits good data hiding or a low level of global data usage. By contrast, poor encapsulation results in poor data hiding or higher use of global data. A similar relationship for data hiding exists with encapsulation. If good data hiding is achieved, software exhibits good encapsulation or a high level of local data usage. In inverse, as poor data hiding is exhibited, it is accompanied by lower use of local data. When high encapsulation is achieved, software exhibits a high separation of concerns since inter-module influence is low from local data and program logic. When lower encapsulation is

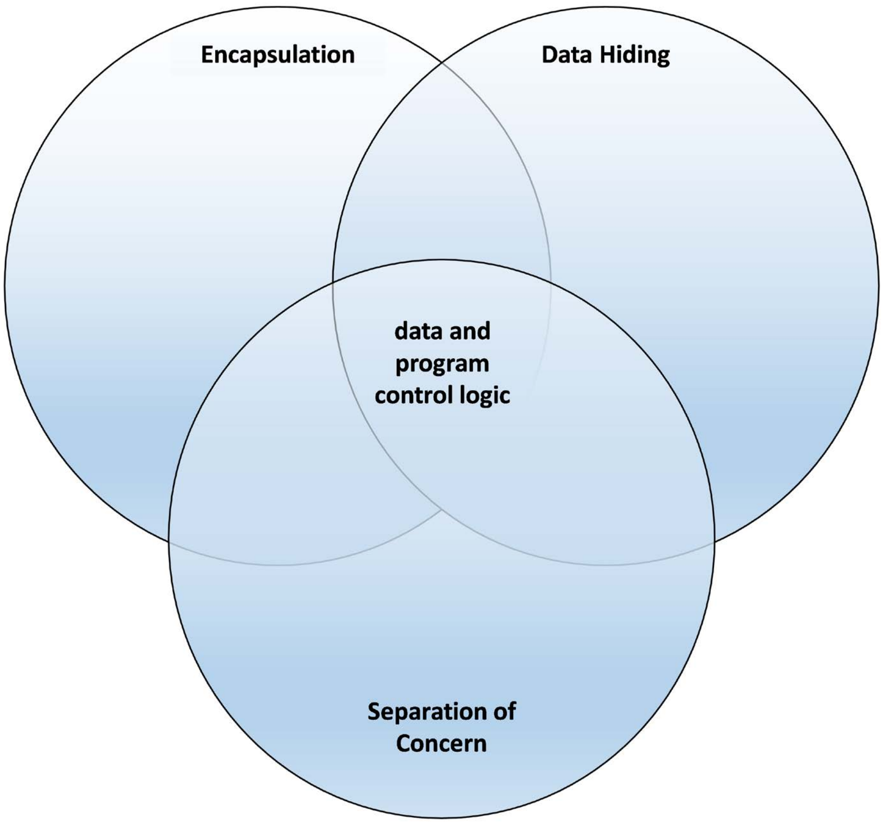

Figure 1. Encapsulation, data hiding and separation of concerns interaction.

realized, it is accompanied by lower separation of concerns as less local data usage will promote more global data and program control logic that reaches into other modules. Finally, when the intersection of data hiding and separation of concerns is considered, software exhibiting high data hiding will result in high separation of concerns. When lower data hiding is realized, it is accompanied by lower separation of concerns as more global data usage will promote more data and program control logic that reach into other modules.

There are pragmatic implications associated with the relationship among encapsulation, data hiding, and separation of concerns. Note the 7 regions in Figure 2 and consider region 1. This region which is the intersection of all three design principles represents good encapsulation, good data hiding and good separation of concerns. What is the software engineering implication of the combined states of these three design principles? As described in Table 2, enhancing, modifying, fixing, and reusing success is best when encapsulation, data hiding, and separation of concerns are all good. As software changes are implemented, design activities should not try to introduce global references or call additional modules to sustain good design principles.

4.2. Design Principle Taxonomy and Evolution Guidelines

In Table 2, each region represents potential effects on software when alternative combinations of good or poor encapsulation, data hiding, and separation of

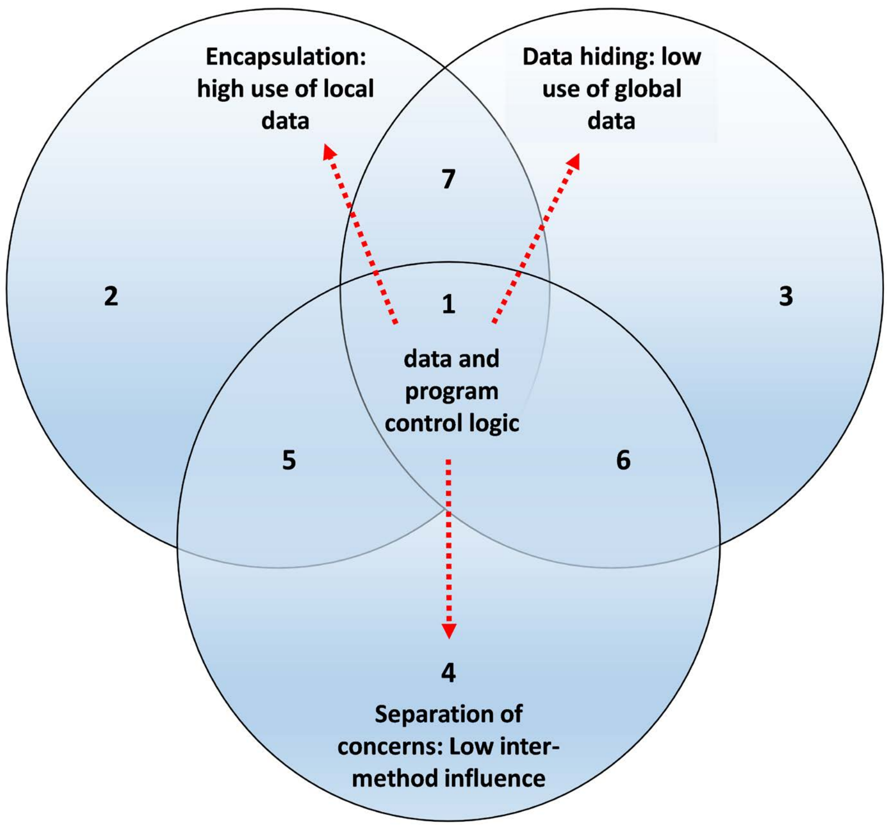
Encapsulation: high use of local data
Data hiding: low use of global data
7
2
1
3
data and program control logic
5
6
4
Separation of concerns: Low inter-method influence

Figure 2. Regions of encapsulation, data hiding and separation of concerns.

Table 2. Design principles and software tactics.

Region #	Encaps ulation	Data Hiding	Separation of Concerns	Tactic When Enhancing, Modifying, Fixing or Reusing
1	good	good	good	Best chance of success; try not to introduce global data references or call additional modules.
2	good	poor	poor	Pinpoint the used global data to see if the modifications affect it. Check how the called modules are affected by the prescribed changes.
3	poor	good	poor	Check the effects on called modules. Make sure the changes are not referencing global data.
4	poor	poor	good	These 'leaf' modules tend to be the most reused—keep it that way by not calling new modules. Be wary of the global data referenced—just pulling the module for reuse without its global data will not work.
5	good	poor	good	The main issue is global data—do not try to reuse without its global data environment.
6	poor	good	good	Good chance of success here. Not much local data used so this is probably highly computational. Also reusable since not dependent on called modules.
7	good	good	poor	This could be a transaction driven module calling several subordinates. Looks clean except for the substructure it calls. When reusing you may need all the modules it calls.

concerns exist. Let's examine region 2, this region represents a design with good encapsulation and poor data hiding and separation of concerns. Since this design has good encapsulation, global data and interdependent control logic among modules produce poor data hiding and separation of concerns. When work is conducted on this design type, modifications should pinpoint the risk of global data changes and the affected modules should be closely monitored for ripple effects. Consider another area, region 3. This region represents a design with good data hiding and poor encapsulation and separation of concerns. When implementing software changes to this class of software, it is important to check the effects on called modules and to ensure that changes are not referencing global data. In Table 2, additional scenarios are described for combinations of encapsulation, data hiding and separation of concerns interaction and potential actions taken when working with various combinations of subjective design qualities.

The proof in the pudding is the usefulness of these design principles in doing what must be done with software modules—constructing, enhancing, modifying, reusing, understanding, and testing. When a module is modified or reused, design principles provide a sensitivity than can help avoid errors and increase productivity. The above Venn diagram has seven distinct regions delineating unique advice and reuse sensitivity based on their encapsulation, data hiding, and separation of concerns qualities. Table 2 addresses issues unique and identifiable by the underlying three design principles. The Venn diagram and table can be useful desktop guides making software developers aware of the sensitivities and issues that should be recognized when dealing with or modifying software. Likewise, these tactics could enhance a modification walkthrough or review to insure the pertinent and unique issues are emphasized and focused upon. It goes without saying that the proposed modifications should only enhance and improve the design so as to not degrade it. The explicit design principles, above Venn diagram, and table can be desktop guides assisting this effort. Even more effective would be quantitative metrics for these design principles. Since the establishment of the values of good and poor is completely subjective, this approach would be strengthened when an objective quantitative measure for encapsulation, data hiding, and separation of concerns is determined. The next sections will develop such quantitative metrics which will ground these concepts and make them operational.

5. McCabe Metrics

Today, the McCabe metric set includes module, design and data metrics. Module level metrics are cyclomatic complexity and essential complexity. Design level metrics are module design complexity, design complexity, and integration complexity. There are two special object-oriented metric derived from the design level metrics. Object design complexity and object integration complexity are special cases of design level metrics calculated for object oriented programs. A third set is data metrics. These metrics are calculated by transferring mathematical concepts of cyclomatic complexity to the data world. Global data complexity is a data metric for public data. Specified data complexity is a metric that supports any selected set of data. The selected set can be public, private, or parameter data, or any combination of these three data types.

5.1. Unit Level Metrics

In 1976, McCabe introduced a number of software metrics. Cyclomatic complexity, v, is a measure of the number of paths through a program [13]. The number of paths can be infinite if the program has a backward branch, and cyclomatic measure is built on the number of basis paths through the program. Cyclomatic complexity is derived from a flowgraph and mathematically computed using graph theory. Cyclomatic complexity is a measurement of the logical complexity of a module and the minimum effort necessary to qualify a module. For the remainder of this research, a module and a method are considered to be synonymous and the word, module, is used to represent both software code types. v is the number of linearly independent paths in a module and, consequently, the minimum number of paths that one should test using McCabe Structured Testing Methodology. There are three mathematical ways to calculate cyclomatic complexity; a simple way is determining the number of decision predicates. Then, cyclomatic complexity is calculated as:

1. $v(G) = \text{number of predicates} + 1$

By examining the decision statements, the design predicates can be counted resulting in the cyclomatic complexity metric. It is important to recognize that many decision statements have compound conditions. An example is a compound IF statement [13]:

If $A = B$ and $C = D$ and $E = F$ then

In this example, $v(G)$ is equal to 4 (three IF design predicates + 1). Cyclomatic complexity recognizes that compound predicates increase program logic complexity and integrates complex decision constructs in order to calculate $v(G)$. An upper limit of 10 for a testable software unit is proposed because McCabe's research found software with $v(G) > 10$ was less manageable, more difficult to test, and less reliable.

In McCabe's original article [13], essential complexity (*ev*) was also defined. Essential complexity is a measurement of the degree to which a module *G* contains unstructured constructs.

2. *ev* = the cyclomatic complexity of a reduced flowgraph, $v(G)$

Reduction is completed by logically removing all structured constructs. Essential complexity can range from 1 to *v*. When *ev* is low, a module is well-structured and can easily be decomposed into multiple modules. In contrast, when *ev* is high, a module is not well-structured and cannot be easily be decomposed into multiple modules [13]. McCabe methodology utilizes graphical representation of software (see **Figure 3**). A flowgraph is an architectural diagram of a software module logic, and a flowgraph is constructed from written code. It is a visualization of module decision logic. A Battlemap is an architectural diagram of a software design. A Battlemap is constructed from written code and is a hierarchical visualization of software modules or classes and methods. Both of these code visualizations are tools used to analyze software design and testing requirements.

High C_{encap} and Low C_{encap}

Coeus – S2S – S2SUtilServiceImpl.getNKeyPersons_MODLTR-BOM_GRF-1

v = 12	ldv = 12
ev = 5	pgdv = 0
iv = 11	C_{encap} = 1.00

Coeus – S2S – PHS398TrainingSubAwardBudgetV1_0Generator.getPHS398TrainingBudget_MODLTR-ANE_GRF-1

v = 12	ldv = 10
ev = 7	pgdv = 3
iv = 5	C_{encap} = .769

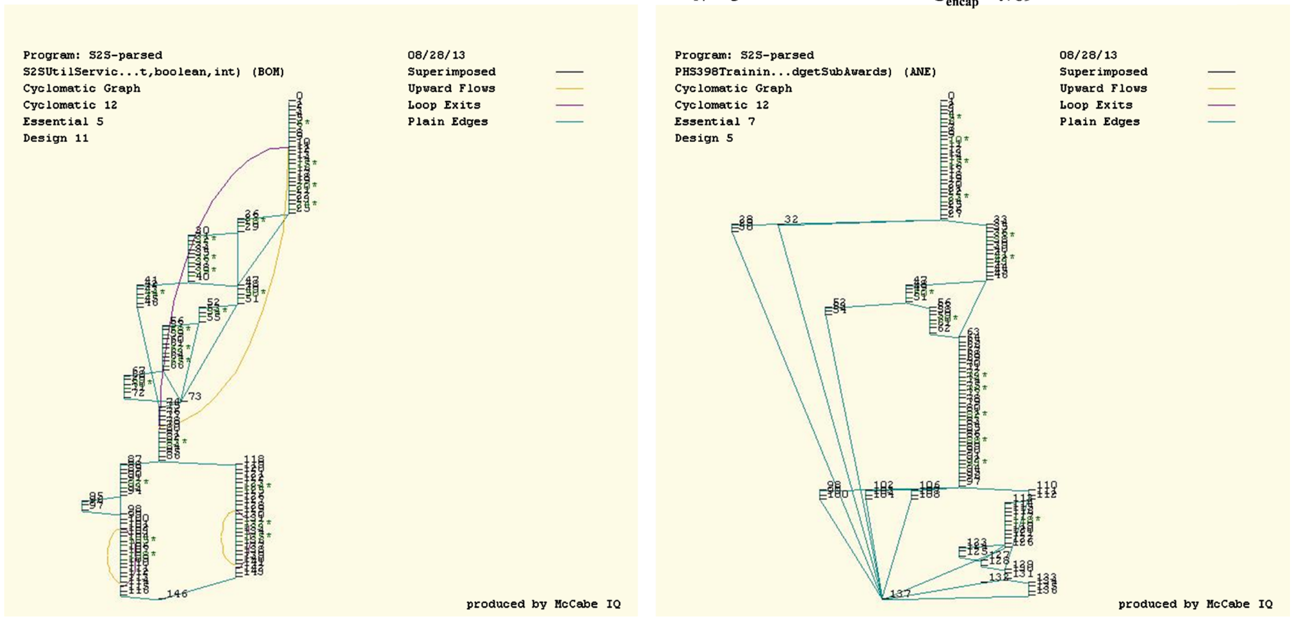

Figure 3. Flowgraphs and design coefficients for kauli coeus software.

5.2. Design Level Metrics

In 1989, McCabe and Butler extended cyclomatic complexity-based metrics into design principles. Module design complexity, *iv*, is a measurement of the decision structure of a module, *G*, which controls the invocation of *G*'s immediate subordinate modules.

3. *iv* = the cyclomatic complexity of a reduced flowgraph, *v*(*G*)

Reduction is completed by logically removing all decision constructs that do not significantly impact calls and returns from subordinate routines. It is a quantification of the testing effort of a module as it calls its subordinates [14]. Module design complexity is mathematically computed using a reduction technique on a flowgraph. The technique eliminates decision predicates that do not significantly impact calls to subordinates. Module design complexity varies from 1 to *v*. When *iv* is low (*iv* = 1), a module does not conditionally call subordinates. In contrast, when *iv* is high (*iv* = 15), a module executes many conditionally calls to subordinates. In order to measure the proportion of logic that is associated with calls, ivdensity (*iv*/*v*) was defined. High ivdensity is a design characteristic that indicates high use of calls in module logic; low ivdensity indicates low use of calls in module logic (ivdensity ranges between 0 and 1) [14]. Two other metrics, design complexity and integration complexity, were introduced. Design complexity, S_0, is a measurement of the decision structure which controls the invocation of modules within the design. It is a quantification of the testing effort of the calls in the design, starting with the top or root module, trickling down through subordinates and exiting through the top. Design complexity is calculated as $\sum iv$. The third metric, integration complexity, is a measurement of the integration tests that qualify the design tree. Integration complexity, S_1, is a quantification of a basis set of integration tests and measures the minimum integration testing effort of a design. Each S_1 test validates the integration of several modules and is known as a subtree of the whole design tree [14].

After introducing these three metrics in 1989, McCabe and Butler developed two variations of S_0 and S_1 for testing of object-oriented code. These metrics took advantage of object oriented design efficiencies to reduce the number of design tests. Object design complexity (OS_0) is a measurement of the decision structure which controls the invocation of modules within an object-oriented design. It is a quantification of the testing effort of all calls in the design, starting with the top module, trickling down through subordinates and exiting through the top. Object integration complexity (OS_1) is a measurement of the integration tests that qualifies the design tree for an object-oriented program including any unresolved module references. A subtree is a sequence of calls from a module to descendant modules and of returns from them. Just as integration complexity design defines the number of test subtrees in the required basis set for that design, OS_1 defines the number of linearly independent design subtrees in a basis set for an object-oriented design [15].

5.3. Data Level Metrics

McCabe metrics are not limited to measurement of logic and decision structure. The data domain includes two classes of metrics [14].

4. *gdv* = the cyclomatic complexity of a reduced flowgraph, $v(G)$
5. *sdv* = the cyclomatic complexity of a reduced flowgraph, $v(G)$

Reduction is completed by logically removing all decision constructs that do not significantly the use of global or selected data [14]. Global data complexity (*gdv*) is a measurement of the decision structure of a module which controls the use of global and parameter data. *gdv* is the cyclomatic complexity after the decision predicates are logically reduced that do not impact data defined either outside the module or passed as formal parameters. Second, specified data complexity is a flexible, configurable measurement of data targeted by a software developer. Specified data complexity (*sdv*) is a measurement of the decision structure of a module which controls the use of selected local, global, and parameter data. *sdv* is the cyclomatic complexity after the decision predicates are logically reduced that do not impact target (selected) local, global and parameter data. Specified data complexity can be used in many ways. For the purposed of this research, it is used to quantify data metrics for measuring encapsulation, data hiding, and separation of concerns [16].

Specified Data Metrics

Six specified data metrics are used in building design coefficients for encapsulation, data hiding, and separation of concern. Below is the definition of these six metrics.

- local data complexity (*ldv*): specified data complexity of local (private) data; $0 \le ldv \le v$;
- ldvdensity (ldv/v): high ldvdensity is a design characteristic that indicates high use of local (private) data in module logic; low ldvdensity indicates low use of local (private) data in module logic; 0 ≤ ldvdensity ≤ 1;
- public global data complexity (*pgdv*): specified data complexity of global (public) data excluding parameter data; $0 \le pgdv \le v$;
- pgdvdensity ($pgdv/v$): high pgdvdensity is a design characteristic that indicates high use of global (public) data in module logic; low pgdvdensity indicates low use of global (public) data in module logic; 0 ≤ pgdvdensity ≤ 1;
- parameter data complexity (*pdv*): specified data complexity of parameters (passed arguments) excluding public global data; $0 \le pdv \le v$;
- pdvdensity (pdv/v): high pdvdensity is a design characteristic that indicates high use of parameters in module logic; low pdvdensity indicates low use of parameters in module logic; 0 ≤ pdvdensity ≤ 1.

These specified data metrics and their density metrics form the foundation for design coefficients.

6. Design Coefficients

In statistics, correlation is a measure of the degree of relatedness of variables. It can used by a business analyst to determine if variables, such as temperature and ice cream sales, rise and fall in a related manner. For a sample pair, correlation analysis can yield a numerical value that represents the relatedness of temperature and ice cream sales over time. The coefficient of correlation, r, is a popular measure of correlation. The statistic r is the Pearson product-moment correlation, named after Karl Pearson, an English statistician [17]. The coefficient of correlation is a measure of the linear correlation between two variables. It is a number that ranges from −1 to 0 to +1, representing the strength of the relationship between the variables. An r value of +1 indicates perfect positive correlation. When r is +1, a positive change in one variable completely explains a positive change in the second variable. When r is −1, a change in one variable completely explains an inverse change in the second variable. An r of 0 means no linear relationship between the two variables. In this case, any change in one variable does not explain change in the second variable. In statistics, a proxy variable is a measured variable that is used in place of a variable that cannot be measured. In order for a variable to be a proxy, it must have a close correlation, not necessarily linear or positive, with the inferred value. Many times in correlation analysis, proxy variables are used to determine if there is a relationship between two variables. These concepts associated with the coefficient of correlation are used as a basis for cyclomatic complexity-based design metrics.

Encapsulation, data hiding, and separation of concerns are design concepts without ordinal or interval measurement. Being able to measure software's design quality is fundamental when assessing the progress of design as it relates to construction. There is now widespread acknowledgement that using subjective design characteristics is a common practice when measuring overall software quality. In practice, these design concepts are measured using subject measurements such as a Likert Scale—very low, low, moderate, high, and very high. However, the Likert Scale is uni-dimensional and only gives limited choices, and the space between each choice is not equidistant. Therefore, a Likert Scale fails to measure the true value of these design concepts. Also, it is likely that software developers' subjective values are influenced by previous development work or opinions of peers. Frequently, people tend to avoid choosing the extremes options on the scale (very low and very high) because of the negative social implications, even when an extreme choice would be best.

Cyclomatic complexity-based design metrics for encapsulation, data hiding, and separation of concerns are measured as coefficients. Each of these design characteristics has a representative coefficient, C_x, where x is either encapsulation (encap), data hiding (dh), or separation of concerns (soc), and each measures the spread of the design characteristics from low to high. For example, the encapsulation coefficient measures encapsulation based upon cyclomatic complexity-based measurements. Unlike the coefficient of correlation, these design measurements are not a measure of linear correlation between two variables. Rather, they are proxy variables used to replace previous subjective measures for encapsulation, data hiding, and separation of concerns. The coefficients range from 0 to +1, representing the spread of the design characteristics from low to high. When C_x is 0, the design characteristic is the lowest and when C_x is 1, the design characteristic is the highest. Using cyclomatic complexity-based measures to derive measurements for these design characteristics provides several advantages. First, the measurements are derived from McCabe metrics which have an acknowledged place in software metrics. McCabe metrics are also objective in that different software developers looking at the same software calculated the same result. McCabe's cyclomatic complexity has a research implication with software quality. Research studies have shown that modules with higher cyclomatic complexity ($v > 10$) are harder to test and exhibit poorer quality [13]. These benefits are integrated into the following cyclomatic complexity-based design metrics.

6.1. Encapsulation Coefficient

The encapsulation coefficient is a measurement of the quality of data bundling within a module operating of that data; a measurement of the decision logic that uses (refers to) local data and global data. The coefficient measures the relative magnitude of local data manipulation to the magnitude of combined local and global manipulation. Decision logic that uses local data indicates high encapsulation (since the data is exclusively bundled with the module) and decision logic that uses global data indicates low encapsulation (since the data is not exclusively bundled with the module).

Module encapsulation coefficient:

$$C_{encap} = f\left(\text{cyclomatic complexity, local complexity,}\right.$$
$$\left.\text{public complexity, public global data complexity}\right)$$

$$C_{encap} = f\left(v, ldv, pgdv\right)$$

(rationale: ldv is a measurement of local data manipulation within a module; high ldv indicates that module logic is bundled using local data)

$$C_{encap}{}^{1} = \left(\frac{ldvdensity - pgdvdensity}{ldvdensity + pgdvdensity}\right) = \frac{\left(\frac{ldv}{v} - \frac{pgdv}{v}\right)}{\left(\frac{ldv}{v} + \frac{pgdv}{v}\right)} = \left(\frac{ldv - pgdv}{ldv + pgdv}\right)$$

where ldv and $pgdv$ not = 0 and $0 \le C_{encap}{}^{1} < |1|$; -1 indicates low encapsulation and 1 indicates high encapsulation.

So that encapsulation, data hiding, and separation of concerns behave quantitatively with common magnitude, encapsulation is scaled to be:

$$C_{encap} = \left(C_{encap}{}^{1} + 1\right)/2$$

where $0 \le C_{encap} \le 1$; 0 indicates low encapsulation and 1 indicates high encapsulation.

Where ldv and $pgdv = 0$, $C_{encap} = 0$.

A peculiar and sometimes pathological case occurs when both *ldv* and *pgdv* are zero. This strange case implies that the algorithm references no data—no local data nor any global data. Even though the denominator is zero in the above formula, the convention of defining the metric value to be zero is used. This convention gives an encapsulation coefficient indicating the lowest (worse) value for encapsulating data. Over and above this mathematical anomaly, these bizarre modules deserve careful inspection. Often they are doing something strange that affects data below the syntactic level of the language being used. For example, in a real-time system, a module referencing no data may just be looping in order to waste some time so it can properly synchronize. It appears that it is using no data but that is not the case. It is affecting the system clock which is data below the syntactic level of language. Change the units in the system clock or get a faster processor and the low encapsulation in this logical construct would cause errors. Another case that has been witnessed is dividing by zero in Ada which triggers an exception recovery procedure. Although there is no syntactic data being used, there is compiler generated data keeping track of these anomalies—so, in fact, this is low encapsulation with risky properties. This same argument applies to data hiding (defined next) when *ldv*, *pgdv* and *pdv* = 0. When modules use no data, they are exhibiting the lowest encapsulation and data hiding characteristics.

6.2. Data Hiding

The data hiding coefficient is a module measurement of the quality of hiding a module's internal data members and limiting data access from external objects.

Module data hiding coefficient:

$$C_{dh} = f\left(\text{cyclomatic complexity, local data complexity,}\right.$$
$$\left.\text{public global data complexity, parameter data complexity}\right)$$

$$C_{dh} = f\left(v, ldv, pgdv, pdv\right)$$

(rationale: *pgdv* and *pdv* are measurements of public global and parameter data manipulation within a module; low *pgdv* and *pdv* indicate that module logic is hiding data from other modules)

$$C_{dh} = 1 - \left(\left(pgdvdendisy + pdvdensity\right) / \left(ldvdensity + pgdvdensity + pdvdensity\right)\right)$$

$$C_{dh} = 1 - \left(\left(\left(pgdv/v\right) + \left(pdv/v\right)\right) / \left(\left(ldv/v\right) + \left(pgdv/v\right) + \left(pdv/v\right)\right)\right)$$

$$C_{dh} = 1 - \left(\left(pgdv + pdv\right) / \left(ldv + pgdv + pdv\right)\right)$$

where *ldv* and *pgdv* and *pdv* not = 0 and $0 \le C_{dh} \le 1$; 0 indicates low data hiding and 1 indicates high data hiding.

Where *ldv* and *pgdv* and *pdv* = 0, $C_{dh} = 0$.

6.3. Separation of Concerns

Module separation of concerns coefficient: module separation of concerns (soc) is a design characteristic for partitioning a design into modules such that each module addresses a separate internal function concern, not relying on other modules.

$$C_{soc} = f\left(\text{cyclomatic complexity, module design complexity,}\right.$$
$$\left.\text{global data complexity, parameter data complexity}\right)$$

$$C_{soc} = f\left(v, iv, pgdv, pdv\right)$$

(rationale: *iv*, *pgdv* and *pdv* are measurements of module logic influencing one or more other modules; low *iv*, *pgdv* and *pdv* indicate that module logic is not influencing other module logic or data manipulation)

$$\begin{aligned} C_{soc} &= \left(\left(1 - ivdensity\right) + \left(1 - pgdvdensity\right) + \left(1 - pdvdensity\right)\right)/3 \\ &= \left(\left(1 - \left(iv/v\right)\right) + \left(1 - \left(pgdv/v\right)\right) + \left(1 - \left(pdb/v\right)\right)\right)/3 \\ &= \left(3 - \left(\left(iv + pgdv + pdv\right)/v\right)\right)/3 \\ &= 1 - \left(\left(iv + pgdv + pdv\right)/3v\right) \end{aligned}$$

where $0 \le C_{soc} \le 1$; 0 indicates low separation of concerns and 1 indicates high separation of concerns.

6.4. Design Coefficient Examples

In Table 3, the design coefficients are calculated for a set of software modules, presented as module pairs A and B, C and D, and E and F. The v for these modules is 10. Modules A and B demonstrate the effect of local and public global data complexity in determining the encapsulation coefficient. In these modules, module design and parameter data complexity are constant. In module A when local data complexity is high (10) and public global data complexity is low (1), the encapsulation coefficient is high (0.909) indicating the module exhibits high encapsulation. In contrast in module B, when local data complexity is low (1) and public global data complexity is high (10), the encapsulation coefficient is low (0.091). In contrasting these two modules, module A uses large quantities of local data and low quantities of global data which are good encapsulation qualities.

Table 3. Cyclomatic complexity-based design coefficients.

Module Name	v	iv	ldv	pgdv	pdv	C_{encap}	C_{dh}	C_{soc}
						$\left(C_{encap}{}^{1} + 1\right)/2$	$1 - \left(\left(pgdv + pdv\right)/\left(ldv + gdv + pdv\right)\right)$	$1 - \left(\left(iv + pgdv + pdv\right)/3v\right)$
A high C_{encap}	10	1	10	1	0	0.909	0.909	0.933
B low C_{encap}	10	1	1	10	0	0.091	0.091	0.633
C high C_{dh}	10	1	10	0	0	1.000	1.000	0.967
D low C_{dh}	10	1	0	10	10	0.000	0.000	0.300
E high C_{soc}	10	0	10	0	0	1.000	1.000	1.000
F high C_{soc}	10	10	10	10	10	0.500	0.333	0.000
G high C_{encap}	20	1	20	1	0	0.952		
H low C_{encap}	20	1	1	20	0	0.048		

Module B uses low quantities of local data and high quantities of global data which are poor encapsulation qualities. The encapsulation coefficient behaves mathematically consistent with these intuitive qualities. Values between these extremes are determined by the actual values for local data and public data complexities for a given module.

In Table 3, modules C and D demonstrate the effects of local data, public global data, and parameter data complexities in calculating the data hiding coefficient. In the data hiding coefficient example, module design complexity is a constant. In module C, local data complexity is high (10), public global data complexity is low (0), and parameter data complexity is low (0). This module's data hiding coefficient is 1.000. In module D, the local data complexity is low (0) and the public global and parameter data complexities are high, 10 and 10, respectively. Module D's data hiding coefficient is 0.000. When contrasting these two modules, module C uses large quantities of local data and low quantities of both global and parameter data which are good data hiding qualities. Module D uses low quantities of local data and high quantities of global and parameter data which are poor data hiding qualities. The data hiding coefficient behaves mathematically consistent with these intuitive qualities. As with the encapsulation coefficient, values between these extremes are determined by the actual values for local data and public global and parameter data complexities for a given module.

The final example illustrated in Table 3 is the separation of concerns coefficient. In this example, modules E and F contrast the effects of module design, public global data, and parameter complexities. In this example, local data complexity is constant. In module E, module design, public global data, and parameter complexities are 0, 0, and 0, respectively. These low metrics indicate that module E has no interaction with other modules, either through control logic or data sharing. Its coefficient is 1.000. In contrast, module F's design, public global data, and parameter complexities are 10, 10, and 10, respectively. Module F has high levels of interaction with other modules through control logic and data sharing. Its separation of concerns coefficient is 0.000. Again, as with the encapsulation and data hiding coefficients, values between these extremes are determined by the actual values for local data and public global and parameter data complexities for a given module.

There is one other important illustration of the encapsulation, data hiding, and separation of concerns coefficients in Table 3. There are two modules, G and H, whose $v = 20$. By traditional McCabe measurement, these modules have twice as much decision logic as the previous 6 modules. This code is considered "too complexity" and is a candidate for decomposition. Compare the encapsulation coefficients for modules G and H with those for modules A and B. Modules A and B have encapsulation coefficients of 0.909 (high) and 0.091 (low). Modules G and H encapsulation coefficients are 0.952 (high) and 0.048 (low). Consider the higher encapsulation coefficients of modules A and G. Module G's encapsulation coefficient is higher than module A's, indicating it is better encapsulated. Even though module G's cyclomatic complexity is considered to be poorer than module A's, its encapsulation coefficient is better. This outcome reflects the fact that the module G makes good use of proportionally more local data and proportionally less public global data. The encapsulation design of module G is better and more desirable. When comparing modules H and B, module H has a lower encapsulation coefficient, 0.048 and 0.091, respectively. Module H's encapsulation coefficient is lower than module B's because it makes use of, proportionally, less local data and more public global than module B.

7. Design Coefficients Applied to Production Code

The Kuali Foundation is a consortium of interested universities, colleges and commercial firms that joined together to produce an enterprise software solution for the academic business model. The Kauli Foundation utilizes a community source developed software acquisition module. Using Kuali Enterprise Resource Planning (ERP) modules, the encapsulation, data hiding, and separation of concerns coefficients are calculated using production code. For this study, the Kauli Coeus, Version 5.0.1, and Kauli Financial System, Version 4.1.1, application code was utilized. Kauli Financial System is financial software that meets the needs of Carnegie class institutions, and Kauli Coeus is an application for administration of grants to federal funding agencies [18]. In **Figure 3**, flowgraphs for two Coeus modules are illustrated. Both modules have similar v's and ev's, which indicate similar testing requirements and code construction quality for maintainability. However, these metrics do not provide insight into their design quality.

Table 4 contains traditional McCabe metrics and complexity-based design

Table 4. Kuali coeus production code design coefficients.

Module/Module	v	ev	iv	ldv	pgdv	pdv	C_{encap}	C_{dh}	C_{soc}	Design Quality
A—(Coeus-S2S PHS398TrainingSub AwardBudgetV1_0Generator.getPHS398TrainingBudget_MODLTR-ANE_GRF-1)	12	5	11	12	0	0	1.000			Higher
B—(Coeus-Budget-QueryList.getFieldValue_MODLTR-JB_GRF-1)	12	7	5	10	3	0	0.769			Lower
C—(Coeus-S2S-SFLLLV1_1Generator.getReportEntity_MODLTR-BGE_GRF-1)	14	1	14	0	1	0		0.933		Higher
D—(Coeus-S2S-GlobalLibraryV2_0Generator.getAddressDataType_MODLTR-YI_GRF-1)	13	1	12	0	2	4		0.684		Lower
E—(Coeus-S2S-S2SUtilServiceImpl.getNKeyPersons_MODLTR-BOM_GRF-1)	12	7	5	0	2	0			0.750	Higher
F—(Coeus-S2S-PHS398TrainingSubAwardBudgetV1_0Generator.getPHS398TrainingBudget_MODLTR-ANE_GRF-1)	11	6	10	0	0	8			0.684	Lower

metrics for 6 Coeus modules including the two modules shown in **Figure 3**. These design coefficients are calculated for Coeus code using McCabe IQ software. The modules are paired by McCabe metrics, v and ev. In the table each module is provided a short name, A, B, C respectively, because the actual module name is long and complex. As pairs, A and B, C and D, and E and F have equal or similar v and ev metrics. For these pairs, a different design coefficient is provided. For the first pair, module A has $C_{encap} = 1.000$ and module B has $C_{encap} = 0.769$, indicating module A has better encapsulation that module B. Data hiding coefficients are calculated for modules C and D. Module C's C_{dh} is 0.933 which is a higher data hiding coefficient than module D's (0.684). C_{soc} is illustrated using modules E and F. With this module pair, the C_{soc} is close with module E's C_{soc} being slightly higher, 0.750 and 0.684, respectively. The magnitude of these coefficients behaves consistently with the design principles subjective values, and they are ordinal values providing an objective measurement of each design principle.

8. Class Design Coefficients

An extension of design coefficients is the measurement of class encapsulation, data hiding, and separation of concerns. Extending design coefficients to the class level is valuable because the metrics provide objective measurement for a higher abstraction of code. Since a class is composed of modules, the class modules are used to calculate the class design coefficients. Instead of a simple aggregation of individual module coefficients, a weight average based upon cyclomatic complexity is used. Using module cyclomatic complexity as a weight, the class design coefficients take into account a module's granularity. When a large module (for example, $v = 40$) is poorly designed with low encapsulation, data hiding, and separation of concerns, that module's weight will negatively impact the class design coefficient. The inverse is true when design coefficients are positive. Utilizing this approach, a small well-designed module and a poorly-designed module do not cancel out their contribution to the class design coefficients.

Class encapsulation coefficient: a measurement of the level of bundling of data with modules operating on that data; a measurement of the proportion of decision logic that uses (refers to) local data and global data. The class encapsulation coefficient is calculated by summarizing the module encapsulation coefficients that are encapsulated in the class. It is the weighted average of the class module encapsulation coefficients.

$$O_{encap} = \left(\sum^{m} \left(v^{m}\right)\left(C_{encap}\right) \right) \Big/ \sum v^{m}$$

where m is a module.

Class data hiding coefficient: a class measurement of the level of hiding internal data members and limiting data access from external objects. The class data hiding coefficient is calculated by summarizing the module data hiding coefficients that are encapsulated in the class. It is the weighted average of the class module data hiding coefficients.

$$O_{dh} = \left(\sum^{m}\left(v^{m}\right)\left(C_{dh}\right)\right)\Big/\sum v^{m}$$

where m is a module.

Class separation of concerns coefficient: class separation of concerns is a design characteristic for separating a computer program into object classes, such that each class addresses a separate concern. The class separation of concerns coefficient is calculated by summarizing the module separation of concerns coefficients that are encapsulated in the class. It is the weighted average of the class module separation of concerns coefficients.

$$O_{soc} = \left(\sum^{m}\left(v^{m}\right)\left(C_{soc}\right)\right)\Big/\sum v^{m}$$

where m is a module.

Class Coefficient Example

Table 5 illustrates design coefficients at the class level. These examples are generated from Kuali code, using the Budget Construction (BC) module of the Kauli Financial System (KFS), Version 5.0.2. Budget Construction is the module that supports fiscal year-based budgeting and annual/monthly amount breakdowns. There are two classes in this example, TempListLookupAction and OjbPendingBCAppointmentFundingActiveIndicatorCoversion (OjbPendingBC). There are 15 and 2 methods in each class, respectively. TempListLookupAction exhibits lower values with values of 0.355, 0.469, and 0.206 for O_{encap}, O_{dh}, and O_{soc}, respectively. The class design coefficients for the OjbPendingBC class are 0.641, 0.786, and 0.524 for O_{encap}, O_{dh}, and O_{soc}, respectively. When contrasted to TempListLookupAction, OjbPendingBC is better encapsulated with higher data hiding, and it has better separation of concerns. Since TempListLookupAction is more granular with fewer methods, greater use of local data, and lower use of global data, its design coefficients reflect better levels of encapsulation, data hiding, and separation of concerns.

9. Design Coefficients for Module Types

There are software instances where encapsulation, data hiding, and separation of concerns behave unique to the module's function. In these instances, data sharing is not always bad; logical dependency is not always bad. Table 6 contains 5 examples of unique types of modules sometimes found in software solutions. Consider module A whose function is an execution switch. This module manages the control between a high level module and many lower level modules; it controls the calls to numerous subordinates through control logic rather than data. For this module type, a lower (0.659), rather than a higher C_{soc}, is expected, since directing subordinate modules requires control logic. Again, a lower C_{soc} is not necessarily bad. If a global data switch (module B) functions logically to direct

Table 5. Kauli financial system budget construction class design coefficients examples.

		v	C_{encap}	C_{dh}	C_{soc}
Class	temp List Lookup Action Class				
Methods	perform Lookup	3	0.330	0.500	0.000
	build Lock Key Message	4	0.500	0.570	0.330
	populate Lock Summary	6	0.460	0.550	0.280
	perform Question Without Input	4	0.330	0.500	0.170
	do Unlock Confirmation	6	0.380	0.500	0.220
	unlock	6	0.230	0.380	0.110
	submit Report	6	0.230	0.380	0.110
	get NEW Incumbent	1	0.330	0.500	0.000
	perform Extended Incumbent Search	1	0.330	0.500	0.000
	search	1	0.330	0.500	0.000
	get New Position	2	0.500	0.670	0.330
	perform Extended Position Search	12	0.330	0.380	0.360
	cancel	2	0.400	0.500	0.170
	clear Values	4	0.400	0.500	0.170
	start	5	0.380	0.500	0.130
Design Coefficients		63	$O_{soc} =$ 0.355	$O_{dh} =$ 0.469	$O_{soc} =$ 0.206
Class	Ojb Pending BC Appointment Funding Active Indicator Conversion				
Methods	sql ToJava	4	0.8	1	0.67
	java ToSql	3	0.43	0.5	0.33
Design Coefficients		7	$O_{soc} =$ 0.641	$O_{dh} =$ 0.786	$O_{soc} =$ 0.524

Table 6. Design coefficients for method types.

Method Name	v	iv	ldv	$pgdv$	pdv	C_{encap}	C_{dh}	C_{soc}
						$\left(C_{encap}{}^{1}+1\right)/2$	$1-\left(\left(pgdv+pdv\right)/\left(ldv+gdv+pdv\right)\right)$	$1-\left(\left(iv+pgdv+pdv\right)/3v\right)$
A—execution switch	45	45	0	1	0	0.000	0.000	0.659
B—global data switch	45	1	1	45	0	0.022	0.022	0.659
C—local data switch	45	1	45	0	0	1.000	1.000	0.993
D—small well-structured	10	1	5	0	0	1.000	1.000	0.967
E—large complex unstructured	34	28	10	15	0	0.400	0.250	0.431

global data through an application, it should use high volumes of public global and parameter data resulting in a low C_{encap} and C_{dh}. Module B's low C_{encap} (0.022) and C_{dh} (0.022) are not bad since they reflects the module's functional responsibility.

Other module types can be identified and classified by design coefficient metrics. A local data switch should exhibited good design coefficients. In **Table 6**, module C has high C_{encap} (1.000), C_{dh} (1.000) and C_{soc} (0.993), since the module type should have good encapsulation, data hiding, and separation of concerns. A small, well-structured module, such as module D, should also have good encapsulation, data hiding, and separation of concerns. Module D fits this profile with high C_{encap} (1.000), C_{dh} (1.000) and C_{soc} (0.993). However, a large, complex, unstructured module should not have the higher quality implications of a small, well-structured module. Module E illustrates this concept with high measurements for *v*, *iv*, *ldv*, *pgdv*, and *pdv*. The complex module's functionality included subordinate control logic, local data usage, and public global and parameter data usage. The impact of this complexity results in poor encapsulation, data hiding, and separation of concerns as reflected in low C_{encap} (0.400), C_{dh} (0.250) and C_{soc} (0.431).

10. Conclusions

Encapsulation, data hiding, and separation of concerns have been prominent design principles for over forty years. Subjectively, software developers have used them to assess the quality of designs. Quantifying encapsulation, data hiding, and separation of concerns design principles provide metrics valuable to software developers as software design tools. Our design metrics, coefficient of encapsulation, coefficient of data hiding, and coefficient of separation of concerns are three metrics which represent an important set of tools. Since the design structure of a program is an important component of logical complexity and data usage, the decision structure inherently contains the manner in which the design logic is implemented. These three design metrics address how software logic reflects the use of local, global, and parameter data in program logic. With these metrics, the quantification can be completed at two levels: module and class. Calculation of design coefficients represents a new analytical tool previously unavailable to software developers.

The coefficient of encapsulation, coefficient of data hiding, and coefficient of separation of concerns exhibit desired properties which support their applicability.

- The metrics are objective and mathematically rigorous. In addition to being intuitive consistent with low or high encapsulation, data hiding, and separation of concerns, it is critical that the metrics be objective. The same design principle viewed a two different times or by two different software developers yields the same coefficient values. If the metrics are not objective, the individual interest involved in a development effort will have differing interpretation reducing the effectiveness of the coefficients as tools.
- The metrics intuitively behave with the subjective degree of design quality.

When a module or class is well-designed, the encapsulation, data hiding, and separation of concerns are determined to be high, good or positive depending on the subjective stratification used. Accordingly, when a module or class is well-designed, the coefficient of encapsulation, coefficient of data hiding, and coefficient of separation of concerns are valued from 0 to 1 with 1 corresponding to high, good, or positive.

- The metrics should be of operational help. Metrics that correlate and estimate characteristics such as quality and maintainability are useful. If the metrics can directly drive the design modularization process.
- The design metrics introduced in this article are for quantifying design principles in procedural and object-oriented code. They are valuable for assessing and controlling encapsulation, data hiding, and separation of concerns. They are also valuable for application portfolio management as they provide measures of design dynamics throughout the application life cycle. If better encapsulation, data hiding, and separation of concerns is achieved in during software design and maintenance, there will be positive impact on application costs, reliability, and performance.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Xu, J., Ho, D. and Capretz, L.F. (2008) An Empirical Validation of Objected-Oriented Design Metrics for Fault Prediction. *Journal of Computer Science*, **4**, 571-577. https://doi.org/10.3844/jcssp.2008.571.577

[2] Selvarani, R., Nair, T.R.G. and Prasad, V.K. (2009) Estimation of Defect Proneness Using Design Complexity Measurements in Object-Oriented Software. *International Conference on Signal Processing Systems*, Singapore, 2009, 766-770. https://doi.org/10.1109/ICSPS.2009.163

[3] Zage, W., Zage, D., McDonald, P. and Khan, I. (1993) Evaluating Design Metrics on Large-Scale Software. *IEEE Software*, **10**, 75-81. https://doi.org/10.1109/52.219620

[4] Basili, V.R., Briand, L. and Melo, W.L. (1995) A Validation of Object-Oriented Design Metrics as Quality Indicators. University of Maryland, College Park.

[5] Sarkar, S., Kak, A.C and Rama, G.M. (2008) Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software. *IEEE Transactions on Software Engineering*, **34**, 700-720. https://doi.org/10.1109/TSE.2008.43

[6] Coleman, D., Ash, D., Lowther, B. and Oman, P. (1994) Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, **27**, 44-49. https://doi.org/10.1109/2.303623

[7] Chidamber, S.R. and Kemerer, C.F. (1994) A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, **20**, 476-493. https://doi.org/10.1109/32.295895

[8] Bansiya, J. and Davis, C.G. (2002) A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transaction on Software Engineering*, **28**, 4-17.

https://doi.org/10.1109/32.979986

[9] Tang, M.-H., Kao, M.-H. and Chen, M-H. (1999) An Empirical Study of Object-Oriented Metrics. *Proceedings of the Software Metrics Symposium*, Boca Raton, 4-6 November 1999, 242-249.

[10] Page-Jones, M. (2000) Fundamentals of Object-Oriented Design in UML. Addison-Wesley Longman Publishing Co., Inc., Boston.

[11] Parnas, D.L. (1972) On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, **15**, 1053-1058. https://doi.org/10.1145/361598.361623

[12] Dijkstra, E.W. (1982) On the Role of Scientific Thought. Springer-Verlag, New York, 60-66. https://doi.org/10.1007/978-1-4612-5695-3_12

[13] McCabe, T.J. (1976) A Complexity Measure. *IEEE Transactions on Software Engineering*, **2**, 309-320. https://doi.org/10.1109/TSE.1976.233837

[14] McCabe, T.J. and Butler, C.W. (1989) Design Complexity Measurement and Testing. *Communications of the ACM*, **32**, 1415-1425. https://doi.org/10.1145/76380.76382

[15] McCabe, T.J., Dreyer, L.A., Dunn, A.J. and Watson, A.H. (1994) Testing an Object-Oriented Application. *The Journal of the Quality Assurance Institute*, **8**, 21-27.

[16] McCabe Software. Software Metrics Glossary. http://www.mccabe.com/iq_research_metrics.htm

[17] http://en.wikipedia.org/wiki/Correlation_coefficient

[18] Solutions for Higher Education. http://www.kuali.org/