Chapter 3

Research Design and Methodology

3.1 Introduction

A systematic approach to perform any research is explained in research design. Research methodology includes methods or techniques used at different levels of research. The complete research process layout is depicted through a research model in next section. Initially most commonly used metrics to evaluate object oriented-design in various researches are empirically analyzed for further utilization and to justify the need of new measures evaluated from basic structure of class at design level. The proposed design complexity metrics are validated to predict induced faults and to acquire the reliability in design phase. This chapter provides an overview of various activities and techniques used to target the research.

3.2 Research Process

As shown in figure 3.1, the research process involves various activities. At the initial steps the most commonly used metrics to capture the design complexity i.e. CK metrics and Source Lines of Code (SLOC) are chosen for empirical evaluation. The CK-metrics are already theoretically evaluated against Weyuker’s properties (Chidamber & Kemerer, 1994). The SLOC is also well known size metrics used for both traditional and object-oriented codes evaluation. These metrics are empirically evaluated to predict the induced faults of different severity level at early stage. The five metrics out of seven chosen above are evaluated after coding phase. For early and intensive feedback to manage the quality of software, four design level complexity metrics are proposed, whose values are calculated before coding phase starts. The present research focuses on a two-step validation of proposed design complexity metrics along with theoretical basis of cognitive theory to predict faults induced in a class. This model is helpful in achieving some level of reliability in design phase. In the first step of validation the metrics are analytically evaluated against two formal sets of properties for software metrics. In the second step, the metrics are associated theoretically and empirically to induced faults in a class for
identification of high risk components (classes). Fault data is collected from the test report.

![Fig. 3.1 Research Model to Acquire Reliability in Object-oriented Software](image)

3.3 Analytical Evaluation of the Complexity Metrics

Analytical evaluation is done for theoretical/internal validation of a measure/metric means to verify that metrics is capable to measure what it is suppose to measure. Consistent with the desire to move metrics research into a more rigorous footing, it is desirable to have a formal set of criteria with which proposed metrics can be analytically evaluated. Design complexity metrics are analytically evaluated against two formal sets of properties for software metrics, as first set refereed is meant for good software metrics/design metrics developed by Weyuker and the second one meant for good software complexity metrics introduced by Briand et al. In detail, both framework of evaluation are discussed below.

3.3.1 Weyuker Framework

Weyuker has developed a formal list of nine properties for software metrics and has evaluated a number of existing software metrics using these properties (Weyuker, 1988). These properties include notions of monotonicity, interaction, non-coarseness, non-uniqueness and permutation. Some authors have also critised the Weyuker’s properties. Fenton suggests that Weyuker’s properties are not predicated on a single consistent view of complexity (Fenton, 1991). Cherniavsky and Smith suggest that Weyuker’s properties should be used carefully since the properties may only give necessary, but not sufficient conditions for good complexity metrics (Cherniavsky and Smith, 1991). Even though these properties are widely known formal analytical approach and is therefore chosen for this analysis. This approach was used by Chidamber and Kemerer to analytically validate a suite of OO metrics.
A summary of Weyuker’s properties in the context of OO systems is listed below:

1) **Non-coarseness**: Given a class P and a metric μ, another class Q can always be found such that: \( \mu(P) \neq \mu(Q) \). This implies that not every class can have the same value for a metric, otherwise it has lost its value as a measurement.

2) **Granularity**: Just as Property 1 (Non-coarseness) argues that it is undesirable for a measure to be too ‘coarse,’ in the sense of rating too many programs as being of equal complexity, this property argues that it is also not desirable for a measure to be too ‘fine’ and assign to every program a unique complexity. The reasoning behind this property is that although having a measure which is very ‘fine’ can be technically accepted, it would, in practical terms, lead to overkill. The returns gained by such a fine measure will be marginal when compared to the time and effort required to compute it.

3) **Non-uniqueness**: There can exist classes P and Q, with no features in common, such that \( \mu(P) = \mu(Q) \). This implies that two classes can have the same metric value, i.e., the two classes are equally complex. This means that if two distinct classes doing totally different things are equally complex in nature, the possibility exists that the metric will be able to demonstrate the equality of their complexity irrespective of their functionality. This property is really a subset of property 2. Property 3 talks specifically about two classes doing totally different things, while property 2 talks about any two classes.

4) **Design details are important**: This property expresses the condition that there are functionally equivalent programs with different complexities. The intuition behind this property is that even though two class designs perform the same function, the details of the design matter in determining the metric for the class.

5) **Monotonicity**: For all classes P and Q, the following must hold: \( \mu(P) \leq \mu(P+Q) \) and \( \mu(Q) \leq \mu(P+Q) \), where \( P+Q \) implies combination of P and Q. This implies that the metric for the combination of two classes can never be less than the metric for either of the component classes.
6) **Non-equivalence of Interaction:** There can exist classes P, Q, R such that $\mu(P) = \mu(Q)$ does not imply that $\mu(P+R) = \mu(Q+R)$. This suggests that interaction between P and R can (not always necessarily) be different than interaction between Q and R resulting in different complexity values for P+R and Q+R.

7) **Permutation Changes Complexity:** This property requires that permutation of elements within the item being measured change the metric value. The intent is to ensure that the possibility exists for metric values to change due to permutation of program statements. This property is meaningful in traditional program design, where the ordering of if-then-blocks could alter the program logic (and consequent complexity). In object-oriented design, a class is an abstraction of the problem space, and the order of statements within the class definition has no impact on eventual execution or use. For example, changing the order in which methods are declared does not affect the order in which they are executed, since methods are triggered by the receipt of different messages from other objects. In fact, Cherniavsky and Smith specifically suggest that this property is not appropriate for object oriented design metrics at class level (Cherniavsky & Smith, 1991). However, it might still be appropriate for metrics concerned with object oriented implementation since complexity would depend on the order of statements within a method.

8) **Renaming Property:** This property requires that when the name of the measured entity changes, the metric should remain unchanged.

9) **Interaction Increases Complexity:** For all classes P and Q, $\mu(P) + \mu(Q) \leq \mu(P+Q)$, where $\mu(P+Q)$ is the metric of the class formed by combining P and Q. The principle behind this property is that when two classes are combined, the interaction between classes cannot decrease the metric value. This property need not necessarily be true in the context of object-oriented design. If two classes P and Q, which have some common properties, are combined into one single class, the common properties will appear only once in the resultant class. Thus $\mu(P+Q)$, the metric of the combined class, is likely to be less than the summation of the metrics $\mu(P)$ and $\mu(Q)$ of the individual classes. If $\delta$ is the metric value of the commonality between the two classes P and Q, then $\mu(P+Q) = \mu(P) + \mu(Q) - \delta$, which implies that $\mu(P+Q) \leq \mu(P) + \mu(Q)$.  

62
Thus we can see that this property may not be appropriate in the context of object-oriented design. In fact, all six metrics proposed by Chidamber and Kemerer failed to satisfy this property (Chidamber & Kemerer, 1991).

### 3.3.2 Briand et al Framework

The purpose of this framework is to provide more credibility to the design complexity measures in succession with evaluation using Weyuker’s properties. In this framework, Briand et al have given five properties, which a complexity measure must satisfy to be useful. Several researchers like Misra and Misra (2004), Kushwaha and Misra (2006), Chhabra and Gupta (2009) have also used this set of properties to evaluate and compare complexity metrics. The basic definitions in the framework, which will be used in the evaluation process, are given as under.

**System:** An object-oriented system $S$ is represented as a $<E, R>$, where $E$ represents the set of elements of $S$ and $R$ is a binary relation on $E$ ($R \subseteq E \times E$) representing the relationships between elements of $S$.

**Module:** For a given object-oriented system $S = <E, R>$, a system $m = <Em, Rm>$ is a module of $S$ if and only if $Em \subseteq E, R \subseteq Em \times Em$ and $Rm \subseteq R$. A module $m$ may be a class or a subprogram.

**Complexity:** The complexity of an object-oriented system $S$ is a function $\text{Complexity}(S)$ that is described by Property 1 to Property 5 summarized below;

1) **Nonnegative:** The complexity of a system $S = <E, R>$ is nonnegative if $\text{Complexity}(S) \geq 0$.

2) **Null Value:** The complexity of a system $S = <E, R>$ is null if $R$ is empty i.e. $R = \emptyset \Rightarrow \text{Complexity}(S) = 0$.

3) **Symmetry:** The complexity of a system $S = <E, R>$ does not depend on the convention chosen to represent the relationships between its elements i.e.

$$(S = <E, R> \text{ and } S^{-1} = <E, R^{-1}>) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1})$$

4) **Module Monotonicity:** The complexity of a system $S = <E, R>$ is no less than the sum of the complexities of any two of its modules with no relationships in common i.e. $S = <E, R>$ and $m_1 = <Em_1, Rm_1>$ and $m_2 = <Em_2, Rm_2>$ and
\[ m_1 \cup m_2 \subseteq S \text{ and } R_{m_1} \cap R_{m_2} = \emptyset \Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2). \]

5) **Disjoint Module Additivity:** The complexity of a system \( S = <E, R> \) composed of two disjoint modules \( m_1, m_2 \), is equal to the sum of the complexities of the two modules i.e. \( (S = <E, R> \text{ and } S = m_1 \cup m_2, \text{ and } m_1 \cap m_2 = \emptyset) \Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2). \)

In this research, all four of the design complexity metrics (CMCM, CICM, CALM and CCOM) are evaluated with the properties included in both frameworks discussed above.

### 3.4 Empirical Evaluation of the Complexity Metrics

This section describes the empirical/external validation process. This validation process is to associate design metrics with some important external measure such as induced faults in a class, by taking two approaches i.e. theoretical and empirical. Cognitive theory forms a theoretical basis for proposition of design complexity metrics constituting cognitive complexity and their association with faults induced in a class. The details of theory are described in next section. A controlled laboratory experiment is conducted for collecting design metrics and fault data as empirical data to validate the metrics. The use of experimental studies allows the researcher to manipulate the variables of interest and control other variables which are not of main interest. Since this research is an attempt to propose a new set of design metrics and validate this set of metrics, thus controlled experiment is the most appropriate option for collecting data. In this study, we attempt to empirically validate the design complexity metrics (CMCM, CICM, CALM and CCOM), by demonstrating that they are predictors of faults in a class. This involves some sort of statistical analysis to predict the faults and determine the significance of the result.

Sub-sections provide the outline of empirical evaluation of design metrics.

#### 3.4.1 Analysis Methods

A summary of the analytical techniques that we used during empirical validation of design metrics, and their mapping to the objectives of the analysis are given in table below:
Table 3.1 The Steps of Research Process and Analytical Techniques Used

<table>
<thead>
<tr>
<th>Objective</th>
<th>Analysis Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validate Existing Metrics to Predict Faults of Different Severity</td>
<td>Univariate and Multivariate Linear Regression Analysis and Cp Mallows Function</td>
</tr>
<tr>
<td>Validate Metrics/Test Hypotheses</td>
<td>Univariate and Multivariate Linear Regression Analysis</td>
</tr>
<tr>
<td>Build Fault Prediction Model</td>
<td>Multivariate Linear Regression Analysis</td>
</tr>
<tr>
<td>Evaluate Predication Model</td>
<td>Spearman’s Rank Correlation Analysis</td>
</tr>
</tbody>
</table>

Linear Regression Analysis

Regression analysis is one of the most powerful and widely acclaimed statistical techniques, which deals with finding appropriate models to represent the stochastic relationship among the variables charactering any real world phenomenon. In regression models a large body of literature centres around linear regression models due to its simplicity, easier analysis and better developed inferential procedures. Linear regression is used to construct models when the dependent variable is continuous or count, as in our case. Linear regression is used to validate the metrics and to construct prediction model. Linear regression correlates the change in a variable called dependent variable to the other variable(s) called independent variables. If we have \( n \) observations on dependent and independent variables such that \( y=f(x) \) where \( f \) is linear then we can express the dependent variable as

\[
Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n
\]

The above equation is general form of linear regression applicable for both univariate and multivariate. Where \( \beta_0, \beta_1, \beta_2, \ldots \ldots, \beta_n \) are the unknown regression coefficients. \( \beta_0 \) is known as constant and \( \beta_1, \beta_2, \ldots \ldots, \beta_n \) are known as coefficients during the analysis. The coefficients are estimated using maximum likelihood function and it measures the probability of observing the set of dependent variables values. The value of coefficient may be negative or positive, depending on correlation between dependent variable and predictor. The detail information of linear regression model includes coefficient, constant, statistical-significance (i.e. p-value),
R²-statistics (R²-value) and standard error for the estimation (std err.). Statistical significance measures the significance level of coefficient and estimated impact of predictors (low p-value is favourable). The R²-statistics measures, how much of the variability in response is explained in the regression model. It also shows the strength of correlation. Large R²-value means more effect of predictors and higher accuracy of the regression model. The less value of standard error is must for better estimation.

Mallows Cp Function

Mallows Cp is a statistical function of the error sum of squares for the full model and that for the reduced model (Mallow, 1973). So the maximum value of R² (adj.) and minimum value of Mallows Cp function for a given reduced model is a criterion to choose the best subset of explanatory variables, with better predictive capacity. It is possible that a subset (metrics used in combination) may or may not predict the number-of-faults of different severity level with equal capability, so need to analyze separately for ungraded, high, medium and low severity faults. During best subset regression analysis, number of variables (vars), R²-adjusted statistics {R² (adj.)} and value of Mallows Cp function (Mallows Cp) are taken in addition to R²-statistics and standard error for the estimation to get detail information of various models.

Spearman’s Rank Correlation Analysis

The Spearman’s rank correlation coefficient, rs is used to evaluate the faults prediction model and test the significance of correlation between predicted faults and faults identified in a class during testing phase. It provides a nonparametric significance test that works well with ranked data without precise proportional scaling and can be used to detect relationships other than the linear one. For two independent evaluations X₁ and X₂ of n items that are to be correlated, the values of X₁ and X₂ are ranked from 1 to 'n' according to their relative size within the evaluations means. For each X₁ and X₂ pair in relative rank, differences in rank ‘d’ are computed. The sum of all squared ds denoted by \( \sum d^2 \) is used to compute \( r_s \) as follows:

\[
r_s = 1 - \frac{6\sum d^2}{n(n^2 - 1)}; \quad -1.0 \leq r_s \leq +1.0
\]
Where \( r_s \) is the coefficient of Rank Correlation, \( n \) is the number of paired observation, and \( d \) is the difference between the ranks for each pair of observations. Threshold value of \( r_s \) is calculated for corresponding \( n \)-value and is the critical value of student’s t-distribution. If the observed value of \( r_s \) is greater than threshold value of \( r_s \) then there is significant correlation between predicted faults and identified by the testing team and model is accurate for predicting the faults in design phase.

### 3.4.2 Independent and Dependent Variables

All the variables for this study are measured at the system design level. The four complexity metrics are the key independent variables for this study. They are; Class Member Complexity Measure (CMCM), Class Inheritance Complexity Measure (CICM), Class Aggregation Level Measure (CALM), Class Cohesion Measure (CCOM). Dependent variable is induced faults in a class. This variable is measured from the test report.

### 3.4.3 Data Collection

Reliability of a software system is probability of failure free operation for a period of time under specified conditions. Failure in software system is caused by some fault.

Thus to manage the reliability of software product, we use fault forecasting techniques by estimating the induced faults in a class. First we empirically analyse the CK-metrics and SLOC to predict the induced faults of different severity in a class, the scholar collected the data from NASA public data set KCI, which is stored at NASA data repository after collected and validated by the Metrics Data Program (MDP, 2006, http://sarpresults.ivv.nasa.gov/view research/107.jsp) from NASA Metrics Data program. (http://mdp.ivv.nasa.gov/). This data was collected from a storage management system for receiving and processing ground data which was implemented in C++ programming language. This project consists of 145 classes that are comprised of 2107 methods and 43K SLOC (Source Lines of Code) of C++. This data set contains the metrics data and fault data. Fault data is categorized according to severity into five different levels which is processed and analysed in Chapter 4 to predict induced faults at early stage.
Secondly, we proposed four design metrics to manage the reliability of software product before the coding starts, while constructing class during system design to predict the induced faults. Faults induced due to the structural behaviour are identified from test report and also from post-release fault report. Differences between two reports is due to their way of considering a fault. During testing maximum possible test-cases are developed with the intention of finding faults during software operation at class level and system level both. Set of test-cases may exclude some of inputs for which the system may fail during actual execution after implementation. After proper testing and reviewing of the system there are very-very low possibility that unidentified fault causes failure. The post-release fault-report includes only faults those cause actual failure. Software reliability certainly goes up with the passage of time and removal of faults. Still at any time, probability of software failure is not zero. Thus true post-release fault report generation is a timely process and time period is same for which the software is used without any modification. Due to time limit and unavailability of historical data of software regarding actual execution, test-report are used to collect the induced fault in a class. The four proposed metrics CMCM, CICM, CALM and CCOM are used to capture the design complexity and evaluated from the design document before the coding starts.

It is an established fact that bigger classes are likely to have a fault. These are also some evidence to support the association between various complexity-product metrics and class size in the study by Jie Xu et al. (2008), there is significant correlation between all CK metrics except NOC and size metrics. Thus all class taken for the source of empirical data to validate the design complexity metrics belongs to particular range of size to neglect the confounding effect upto some level. The class size is measured by total number of method in the class. Validation process of metrics includes the various class in which number of methods are at least two and at most six. For developing a fault prediction model through empirical validation process, four medium size commercial project are chosen, having 20 to 25 classes. These projects are developed using C++ language. From this pool of 92 classes only 25 classes are chosen randomly from a subset having the size specified above (i.e. having number of methods 2 to 6).

The evaluation of work to collect the metrics and fault data is carried out manually. Design document was generated by the development team before the
coding phase starts are used for preparing metric data. Similarly test report for fault-data collection.

To further validate the faults prediction model, four projects are chosen, meant to perform different functionality. All four projects have same number of classes but classes are of various size. Size of project in terms of number of classes is fixed to simplify the statistical analysis process to evaluate the developed model.

3.5 Cognitive Theory of Object-Oriented Metrics

A theoretical basis for developing quantitative models relating early stage product metrics and external quality metrics has been provided in (Briand, Wuest, Ikonomovski, and Lounis, 1998), and is summarized in Figure 3.2.

Fig. 3.2 Theoretical basis for the Development of Object-oriented Product Metrics as Early Indicator of External Attributes

This theory hypothesizes that the structural properties of a software component have an impact on its cognitive complexity. Cognitive complexity is defined as the mental burden of the individuals who have to deal with the software component (class), e.g. the developers, testers, inspectors, and maintainers. The high cognitive complexity of classes may result in many different types of problems such as low maintainability or high fault-contents/fault-proneness. However, induced faults are not only an important quality aspect related to class cognitive complexity but also the easiest one to observe and measure, hence its use in this study. For instance, Munson and Khoshgoftaar state that there is a clear intuitive basis for believing that complex programs have more faults in them than simple programs. This implies that a cognitive theory of object-oriented metrics would explain how and why certain metrics are associated with fault-proneness. The rationale for the theory is the common belief
that the structural properties of a class have an impact on its cognitive complexity (Briand et al., 2000)

Four constructs of the cognitive theory applicable on object-oriented software metrics explicit by figure 3.3 are as under:

(i) Structural class properties

(ii) Recall

(iii) Comprehension

(iv) External attributes

In software engineering work, we only measure the first and fourth constructs. We have object-oriented design metrics to measure structural class properties and metrics of the incidence of faults to measure reliability as external attribute. These two measurable constructs are related and explained by the relations through recall and comprehension in three steps. First step; structural properties of class to recall, second step; recall to comprehension and third step; comprehension to external attributes. Some studies discussed below provide the evidence of association between structural properties capturing the dependencies (among and within classes) and understandability/induced faults/external attributes.
as a method. However, Henderson-Sellers (1996) notes that a class is also an important type of (compound) chunk.

3.5.3.2 Memory Span and STM

Short term memory (STM) can be thought of as a container, where a small finite number of concepts can be stored. If data are presented in such a way that too many concepts must be associated in order to make a correct decision then the risk of error increases. The classical theory of STM proposes that information went into an intermediate storage where it had to be rehearsed before it can go into the relatively permanent Long Term Memory (LTM) (Anderson, 1995). If an item left STM before a permanent LTM representation was developed, it would be lost forever. One could not keep information in STM forever since new information would always be coming in and push the old information out. Essentially, information had to "do time" in STM before it can get to LTM. Furthermore, it was believed that STM had limited capacity.

In a software engineering context, Hatton (1997) argues that Miller (1957) shows that humans can cope with around 7±2 pieces of information (or chunks) at a time in STM, independent of information content. He then refers to the text of Hilgard et al. (197f). Hilgard et al. note that the contents of long-term memory are in a coded form and the recovery codes may get scrambled under some conditions. Short-term memory incorporates a rehearsal buffer that continuously refreshes itself. Hatton suggests that anything that can fit into short-term memory is easier to understand and less fault-prone. Pieces that are too large or too complex overflow, involving use of the more error-prone recovery code mechanism used for long-term storage. Tracz (1979) argues that STM has a fixed size, and therefore software complexity must not exceed the number of items capable of being held in STM at one time, and that the size of these items should be small enough and well formed enough to be handled by STM. He then also refers to the 7±2 limitation of STM. Hatton uses this reasoning in explaining complexity for procedural and object-oriented applications (Hatton, 1998).

3.5.3.4 Working Memory

Instead of the traditional view of STM, Baddeley (1986) has proposed a model of working memory in which a controlling central executive supervises and coordinates a number of subsidiary slave systems. Two slave systems examined were: (a) the phonological loop, which is assumed to be responsible for the manipulation of speech-
based information, and (b) the visuo-spatial sketchpad, which is assumed to be responsible for setting up and manipulating visual images.

The phonological loop is assumed to have two parts: a phonological store that is capable of holding speech-based information and an articulatory control process based on inner speech. I will focus on the phonological loop since it is of most relevance to the cognitive complexity theories of object-oriented programs.

Baddeley, Thompson and Buchanan (1975) and Vallar and Baddeley (1982) found that the amount of correct recall of a series of words depends on the number of syllables in the words. Thus, the memory span is controlled by the speed at which information can be rehearsed (Baddeley, 1986). Memory span is then determined by the number of items that can be refreshed before they fade away, which depends on how rapidly the trace fades and how long it takes to articulate each item and hence refresh the memory trace.

3.5.3.5 Memory Span Revisited

The question remains as to the nature of the relationship between the phonological loop and memory span. Zhang and Simon (1985) performed a series of studies that address this issue. They used Chinese characters to measure memory span, and found that their overall results apply well to English materials. A simple hypothesis would state that memory span reflects a constant number of chunks. They found that is not so. For instance, in one study their subjects were presented with familiar Chinese radicals, words and characters each of which can be considered a single chunk. Radicals do not have commonly used oral names. Characters have single syllable pronunciations. Words have two syllable pronunciations. They found that span was largest for the characters, suggesting that span is a function of how long the material takes to articulate. The extent of familiarity affects the time it takes to articulate. Therefore, material that is highly familiar will result in a larger span. They propose the following span equation measured in number of chunks (C):

\[ C = \frac{T}{a + b(S - 1)} \]

where \( T \) is a fixed time interval that reflects the duration of the underlying storage parameter, say 2 seconds, \( a \) is the amount of time to bring each new chunk into the
The object-oriented strategies of limiting a class responsibility and reusing it in multiple contexts results in a profusion of small classes in object-oriented systems (Wilde et al. 1993). For instance, Chidamber and Kemerer (1994) found in two systems studied that most classes tended to have a small number of methods (0-10), suggesting that most classes are relatively simple in their construction, providing specific abstraction and functionality. Another study of three systems performed at Bellcore found that half or more of the methods are fewer than four Smalltalk lines or two C++ statements, suggesting that the classes consist of small methods (Wilde et al., 1993). Many small classes imply many interactions among the classes and a distribution of functionality across them.

In one experimental study with students and professional programmers Boehm-Davis et al. (1992) compared maintenance time for three pairs of functionally equivalent programs (implementing three different applications amounting to a total of nine programs). Three programs were implemented in a straight serial structure (i.e., one main function, or monolithic program), three were implemented following the principles of functional decomposition and three were implemented in the object-oriented style, but without inheritance. In general, it took the students more time to change the object-oriented programs and the professionals exhibited the same effect, although not as strongly. Furthermore, both the students and professionals noted that they found that it was most difficult to recognize program units in the object-oriented programs and the students felt that it was also most difficult to find information in the object-oriented programs.

Widenbeck et al. (1999) made a distinction between program functionality at the local level and at the global (application) level. At the local level they argue that the object-oriented paradigm’s concept of encapsulation ensures that methods are bundled together with the data on which they operate, making it easier to construct appropriate mental models and specifically to understand a class individual functionality. At the global level, functionality is dispersed among many interacting classes, making it harder to understand what the program is doing. They support this in an experiment with equivalent small C++ (with no inheritance) and Pascal programs whereby the subjects were better able to answer questions about the functionality of the C++ program. They also performed an experiment with larger programs. The number of correct answers for the subjects with the C++ program (with inheritance) on questions about its functionality
was not much better than guessing. While this study was done with novices, it supports the general notions that high coupling makes them more difficult to understand.

It has been stated that "Inheritance gives rise to distributed class descriptions. That is, the complete description for a class D can only be assembled by examining D as well as each of D's super-classes. Because different classes are described at different places in the source code of a program (often spread across several different files), there is no single place a programmer can turn to get a complete description of a class (Leijter et al., 1992). While this argument is stated in terms of source code, it is not difficult to generalize it to design documents.

Cant et al. (1994) performed an empirical study whereby they compared subjective ratings by two expert programmers of the complexity of understanding classes with objective measures of dependencies in an object-oriented system. Their results demonstrate a correlation between the objective measures of dependency and subjective ratings of understandability. In an experience report on learning and using Smalltalk (Nielsen et al. 1989), the authors found that the distributed nature of the code causes problems when attempting to understand a system.

3.5.1 External Attributes and Comprehension

One way to operationalize cognitive complexity is to equate it with the ease of comprehending a class. Although, to our knowledge, the relationship between comprehension and reliability has not been directly evaluated thus far, intuitively, it makes sense. If classes are difficult to comprehend then there is a greater probability that faults will be introduced during development, and also that it will be more difficult to detect faults during the development fault detection activities, such as testing and inspections. This depicts a positive association between comprehension and reliability of a class.

3.5.2 Comprehension and Recall

It has shown above that dependencies in object-oriented software play an important role in its comprehension and have reviewed common dependency metrics.

Cant et al. have proposed a framework that elaborates on the impact of structure on understandability. This framework has been extended to object-oriented software (Cant et al., 1994). At the core of the cognitive framework proposed by Cant et al.
(1995) is a human memory model. This memory model consists mainly of short-term and long-term memory (STM and LTM respectively). In the same light (Tracz, 1979) has claimed that the organization and limitations of the human memory are perhaps the most significant aspects of the human thought process which affect the computer programmer. Hence, it can be argued that the human memory model is a reasonable point of departure for understanding the impact of structural properties on comprehension.

Schneiderman (1977) proposed that performance on a recall task would be a good measure of program comprehension. He demonstrates that in one experiment recall scores were strongly correlated with modifiability of programs. He then suggests that one way of assessing the ease of comprehension and modifiability of a program is to test its ease of memorization. In a second experiment he found a strong correlation between recall scores and scores on a comprehension test (Schneiderman, 1977). Hence, we can postulate that there is covariation or an association between recall and comprehension.

3.5.3 Structural Properties and Recall

Structural properties that capture dependencies among and within classes are believed to exert significant influence on understandability, for example, coupling and cohesion. Coupling metrics characterize the static usage dependencies among the classes in an object-oriented system (Briand et al., 1999). Cohesion metrics characterize the extent to which the methods and attributes of a class belong together (Briand et al., 1998). Inheritance is also believed to play an important role. Coupling and inheritance capture dependencies among the classes. These dependencies in an object-oriented artifact make it difficult for someone to recall information about the artifact. Some commonly known effects in cognitive psychology to explain this are: interference effects, fan effects, familiarity, and memory span. Interference effects occur when a subject learns intervening material after some initial material. The initial material will be more difficult to recall. Fan effects occur when a concept that has been learned has many other concepts associated with it. This leads to that concept being difficult to recall. Familiarity occurs when a concept in memory is repeatedly recalled, and so it becomes easier to recall again. Memory span limits the amount of information that one can process in memory at one time. Based on studies of engineers comprehending an object-oriented system, we postulate that the way they trace through the class hierarchy when trying to understand the relationship among classes can be mapped to the cognitive
effects above. For example, when an engineer is trying to comprehend a class A and encounters an attribute whose type is another class B, then he will proceed to class B to comprehend what it is doing. This results in an interference effect while comprehending class A. The more class A has connections to other classes, the more difficult it will be to recall information about A due to the fan effect. Furthermore, due to memory span limitations, it is difficult to reason about the full functionality of class A if it is too large and unfamiliar. If class A is an attribute in many other classes, then it will be consulted often during comprehension and therefore will be familiar.

The contemporary models of human memory allow us to explain and identify the impact of object-oriented metrics on recall of information. Their basic elements are discussed in sub sections below.

3.5.3.1 Object-Oriented Chunks

Cant et al. (1995) argue that program comprehension consists of both chunking and tracing. Chunking involves recognizing groups of statements and extracting from them information, which is remembered as a single mental abstraction. These chunks are further grouped together into larger chunks forming a hierarchical structure. Tracing involves scanning through a program, either forwards or backwards, in order to identify relevant chunks. Subsequently, they formulate a model of cognitive complexity for a particular chunk, say D, which is the sum of three components: (1) the difficulty of understanding the chunk itself; (2) the difficulty of understanding all the other chunks upon which D depends; and (3) the difficulty of tracing the dependencies on the chunks upon which D depends. Davis (1984) presents a similar argument where he states "Any model of program complexity based on chunking should account for the complexity of the chunks themselves and also the complexity of their relationship."

Cant et al. (1985) make a distinction between elementary and compound chunks. Elementary chunks consist only of sequentially self-contained statements. Compound chunks are those which contain within them other chunks. Procedures containing a number of procedure calls are considered as compound chunks. At the same time, procedures containing no procedure calls may also be compound chunks. If a procedure contains more than one recognizable subunit, it is equivalent to a module containing many procedure calls in the sense that both contain within them multiple subchunks. Subsequent work by Cant et al. (1994) operationally defined a chunk within object-oriented software
articulatory mechanism, $b$ the time to articulate each syllable in the chunk beyond the first, and $5$ is a measure of chunk complexity which is taken to be the average size of a chunk in syllables.

If one were to apply the above results directly to a situation where a chunk is an object-oriented class, then the following initial hypotheses may be stated:

Memory span depends on the size of a class, such that the span is smaller the larger the classes. Span is larger for familiar classes than unfamiliar classes.

### 3.5.3.6 Activation Theory and LTM

Long-term memory (LTM) has both structure and performance characteristics. It can be considered as a knowledge base of symbol usage. It is convenient to represent it as a weighted multi-graph where a concept is a node and an edge is an association. The edge carries a weight representing how often it has occurred, and a label indicating the kind of association. The extent to which a concept is important has to do with the number of different edges leading to it. How well it is remembered depends on how often and how recently it has been seen. Anderson (1995) makes the point that even for a mildly challenging task such as performing a multiplication of two numbers, it is necessary to rely on long-term memory. Therefore, using arguments based on limitations of working memory to explain the understandability of object-oriented systems are likely to be simplistic. A more fruitful approach is perhaps to understand better LTM since this is also relevant to the understandability of object-oriented systems.

Thus, rehearsal serves to keep information in working memory for as long as it is rehearsed, with the span being affected by speed of rehearsal. At some point, however, information must also be retained for a longer period of time.

Anderson (1993) has proposed activation theory to explain the workings of LTM. This stipulates that the strength of the trace to information in memory determines in part how active it can get and hence how accessible it will be. The amount of activation spread to a memory depends on the strength of that memory. Each time a memory trace is activated, it will increase a little in its strength. Strength of a trace can also be increased by repeated practice. Loss of information from memory is affected by the passage of time. Therefore, memory activation is determined by:
How recently it was used, and how much of the memory was practiced (i.e., how frequently the same information was encountered) another factor, interference effects (Anderson, 1995), influences the ability to retain information. If intervening information is learned after some initial information, then, the retention of the initial information is not as strong.

The additional concept of spreading activation stipulates that the concepts in memory are connected in a network. Whenever a concept is activated, surrounding concepts are activated as well. For instance, if two concepts are associated, and if one of these concepts is activated, then the second concept is primed, meaning that the memory trace to the second concept has also been partially activated. In fact, studies show that in general the more associations to a piece of information that are learned, the more difficult it becomes to recall the information (Anderson, 1974). This is explained by the limited capacity of spreading activation. Therefore, the more paths from a concept, the less activation will go down any path. This is known as the fan effect.

3.5.3.7 Familiarity

It was noted above that the more a memory is practiced (i.e. it becomes more familiar) the stronger the trace to that memory. This mechanism can be formulated in terms of object-oriented applications.

One factor that is contended to have an impact on program complexity is chunk familiarity (Henderson-Sellers, 1996). It is argued that chunks that are referenced more often will be more familiar since they are used more often. Therefore, metrics that are potential indicators of familiarity would be predicted to be positively associated with the comprehension of the classes.

Burkhardt et al. (1998) performed a detailed study of object-oriented program comprehension. They recorded the files accessed while subjects were studying a C++ program, as well as verbal protocols. They also asked their subjects to subsequently perform a reuse and documentation task. The subjects consisted of both novices and professionals, but we only focus on the professional results. The program had an inheritance hierarchy with one root, as well as a number of other classes not within the inheritance hierarchy.

They found that the most consulted classes were at the top of the inheritance hierarchy, at the bottom of the inheritance hierarchy (i.e., deepest classes), and those that provide services used by most other classes but were not in the main inheritance
hierarchy. The classes lowest in the hierarchy are the ones first referenced in the "main" function. This indicates that the classes with zero values of DIT (i.e., root classes), with the highest values on DIT, and those with high values of export coupling are consulted more frequently, and therefore one can further argue that they would be quite familiar to software engineers.

The classes that were least consulted tended to be in the middle of the inheritance hierarchy, and a class representing an already familiar type of data (for example, string). The latter suggests that if the subjects are already familiar with the services of a class, either because of domain knowledge or previous development experience, they are not likely to consult it, irrespective of its export coupling or position in the inheritance hierarchy.

Another interesting observation was made when the session was divided into three chronological stages. At stages 2 and 3 classes up the inheritance hierarchy whose methods are used frequently by its children are consulted often. This suggests that higher values of NOC mean higher familiarity since the more children the more likely that a parent will be consulted frequently. Furthermore, this also suggests that classes with high values of descendant-based export coupling will be consulted often and hence be more familiar.

It can be concluded from the above stated observations that the following are characteristics of classes that would be most familiar to engineers since they are consulted often:

- Classes at the root of the inheritance hierarchy (DIT=0)
- Classes at the bottom of the inheritance hierarchy (largest DIT)
- Classes with many children (large NOC)
- Classes with high values of "Other" and "Descendant" based export coupling
- The exception to the above are classes that implement common (well known) domain or software engineering services.

3.5.3.8 Tracing

Thus, interference due to having to learn intervening material and fan-effects reduce recall ability.

Henderson-Sellers (1996) notes that tracing disrupts the process of chunking. This occurs when it becomes necessary to understand another chunk, as when a method calls
another method in a different class, or when an inherited property needs to be understood. Such disruptions cause interference and increases the chances that knowledge of the original chunk is lost. In fact, tracing dependencies is a common task when understanding object-oriented software. Wilde et al. (1993) found that programmers have to understand a method's context of use by tracing back through the chain of calls that reach it and tracing the chain of methods it uses. Their findings were from an interview study of two C++ object-oriented systems at Bellcore and a PC Smalltalk environment. The three systems investigated span different application domains. Furthermore, they found that one has to trace inheritance dependencies, which may be considerably complicated due to dynamic binding. A similar point was made in Leijter et al. (1992) about the understandability of programs in such languages as C++ that support dynamic binding.

In one study of how programmers comprehend code Koenemann and Robertson (1991), it was found that they followed an "as-needed" strategy. This means that subjects restrict their understanding to parts of the code they consider to be relevant to the task. Their task consisted of making modifications. The authors divided the code into that which is directly relevant to the modification task. Directly relevant code was examined the most. Code of intermediate relevance was examined to a lesser extent. Subjects studied intermediate relevant code if they discover an interaction that they judge needs to be understood to complete the task. This involved tracing procedure and function calls backwards and forwards. Although this study was on a procedural program, the same behaviour as the above two object-oriented studies was exhibited.

### 3.6 Conclusion

As a good research design begets a successful research, similar is the motto of the chapter. At the start, the scholar opines about the research design and its importance in any research. It includes the different models under which the work is strategically carried out. Among the key models discussed are Weyuker, Briand et al. and cognitive theory. Further different statistical methods are employed in the work is also discussed. Overall the entire chapter expresses how logically the work is carried out.
References


