Chapter 5  

Reliability Focused Quality Model for Object Oriented Design

Measuring software reliability remains a difficult problem because we do not have a good understanding of the nature of the software. There is no clear definition to what aspects are related to software reliability [1]. It is therefore tempting to measure something related to reliability to reflect the characteristics if we cannot measure reliability directly. As shown in section 2.6 of chapter 2 the two main factors affecting software reliability are fault introduction, fault removal and operational profile. As faults are removed, failure intensity drops and reliability tends to increase. When faults are being introduced, they often are included in new features or design changes that are all made one time, resulting in failure intensity and a step decrease in reliability [2]. In order to have a timely control on the quality of the software, prediction and estimation of the software reliability made in the early design phase will always be better. However the existing software reliability models are much slanted towards implementation rather than design. In this chapter a new model called Reliability Focused Quality Model for Object Oriented Design (RFQMOOD) is put forth.

5.1 RFQMOOD

RFQMOOD is very much inspired from QMOOD [3]. The initial set of design quality attributes in QMOOD is functionality, effectiveness, understandability, extendibility, reusability and flexibility. However, the quality attributes concentrated by RFQMOOD are reliability, reusability, testability and maintainability.

The object oriented design properties are the one which predicts the nature of quality attributes. In QMOOD [3] one metric across one design property is linked, where as in
RFQMOOD completely a new framework has been set up. The new framework is called Framework for Predicting the Reliability of Object Oriented Design (FPROOD). The hints for this framework are taken from the TAPROOT framework [4]. RFQMOOD uses a set of metrics to describe the nature of each design property instead of relying upon a single metric as done in [3]. Figure 5.1 shows the RFQMOOD model.

**Figure 5.1. Reliability Focused Quality Model for Object Oriented Design**

Another difficulty with the earlier model is the instability to account for dependency among quality attributes. In RFQMOOD, reusability, testability and maintainability are found to have greater impact on reliability.

As the name suggests the prime goal behind setting the new model was to predict the reliability of the object oriented software at the early design phase. Various design components were identified followed with related metrics, which corresponds, to several design properties. These metrics are then mapped to the design properties and design attributes (maintainability, testability and reusability). Once the values across all these metrics are obtained these values are further utilized for predicting the defect density of
the software. After obtaining the defect density by taking the base of the exponential model the reliability of the product is predicted in the early design phase.

5.2 Identifying Design Quality Attributes

The prime goal behind selecting the design quality attributes was to have such set of attributes which can be used to predict the reliability of object oriented software in it’s early design phase. Thus reliability was selected as an essential quality attribute.

There are many different models for software quality that have incorporated reliability. ISO 9126 defines six quality attributes one of which is reliability. IEEE standard 982.2-1988 states, “A software reliability management program requires the establishment of a balanced set of user quality objectives, and identification of intermediate quality objectives that will assist in achieving the user quality objectives. Since reliability is an attribute of quality, it can be concluded that software reliability depends on high quality software.

An important objective in adopting the object oriented approach for design and implementation has been to develop reliable, adaptable, and flexible software system quickly. One way to achieve this has been by encouraging inclusion of “reusability” as an important attribute of object oriented design quality assessment. Software reusability has a positive impact on software reliability, software efficiency and time to market [5]. That is why reusability is considered as one of the dependent attribute for predicting reliability.

The third attribute that has been considered is “testability”. Testability is the amount of test resources needed to reach acceptable test coverage. The testing stages (which include problem solving) before release of software, it should provide some degree of certainty that the product has required features and that it will operate without failures. This attribute relates to the amount and effort and time needed for (white box and black box) testing activities. A great number of test cases might be needed to reach acceptable test coverage. In the early design phase certain object oriented design metrics and indicators can be utilized for predicting the testability, which can assure effective test coverage and therefore the reliability of software. The method of improving testability is the way of improving reliability [6][7].

The fourth attribute that is considered is “maintainability “. Maintainability is the aptitude of the source code to under go repair and evolution. The more it is subject to evolution and repair the less reliable is the current version. Maintainability therefore becomes on of the important attribute for predicting reliability.

Reusability, testability and maintainability attributes once evaluated will be utilized for predicting the reliability of the software product. Figure 5.2 shows the interdependency among the attributes.
Because the focus here is mainly on reliability the proposed model is called RFQMOOD i.e., Reliability Focused Quality Model for Object Oriented Design.

5.3 Identifying Object Oriented Design Properties

Design properties are tangible concepts that can be directly assessed by examining the internal and external structure, relationship and functionality of the design components, attributes, methods and classes. An evaluation of class definition for its external relationships (inheritance type) with other classes and the examination of its internal component attributes, and method reveal significant information that objectively captures the structural and functional characteristics of class and its objects.

To represent the design quality attributes discussed earlier the selected design properties and their definition are shown in table 5.1.

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Design Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Coupling</td>
<td>Defines the interdependency of classes inside a package and to the classes of another package.</td>
</tr>
<tr>
<td>2.</td>
<td>Cohesion</td>
<td>The degree of relatedness of an encapsulated unit (such as a component or a class); that is, in general, it is better to have high cohesion.</td>
</tr>
<tr>
<td>3.</td>
<td>Inheritance</td>
<td>A measure of “is-a” relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.</td>
</tr>
<tr>
<td>5.</td>
<td>Complexity</td>
<td>A measure of degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.</td>
</tr>
<tr>
<td>6.</td>
<td>Abstraction</td>
<td>A measure of generalization-specialization aspect of design. Classes in a design, which have one or more descendants, exhibit this property of abstraction.</td>
</tr>
<tr>
<td>7.</td>
<td>Design Size</td>
<td>A measure of number of classes, methods, packages and abstracts classes used in a design.</td>
</tr>
</tbody>
</table>
5.4 Identifying Object Oriented Design Component

The sets of components, which can help analyze, represent, and implement an object oriented design includes methods, class, packages, relationships and class hierarchies.

5.5 Identifying Object Oriented Design Metrics

Based on what a practitioner wants to emphasis, a set of metrics can be selected for predicting and assessing the quality attributes. The object oriented metric survey [8] shows the various metrics, method metrics, system metrics, inheritance metrics and so on. Various metric suite especially for object oriented design are available. But it was observed that these metric values could be obtained late when the source code was available.

<table>
<thead>
<tr>
<th>Table 5.2 FPROOD Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design Metric</strong></td>
</tr>
</tbody>
</table>
| **Class** | a. DIT (Depth of inheritance tree)  
  b. RFC (Response for class)  
  c. CBO (Coupling Between Objects) | a. Number of methods  
b. Number of Children (NOC) | a. WMC (consider Number of methods in class)  
  if (activity diagram across key methods available then consider CC) |
| **Package** | Coupling Metric:  
  a. Instability  
  \( I = \frac{Ce}{Ca+Ce} \)  
  b. Abstractness  
  \( A = \frac{Na}{N} \)  
  c. Distance from main sequence line.  
  Cohesion Metric:  
  a. Relational cohesion  
  \( H = \frac{(R+1)}{N} \) | a. Number of classes in a package (N)  
b. Number of abstract classes in a package (Na) | a. Number of relationship between classes in a package (R)  
b. Afferent Coupling (Ca)  
c. Efferent Coupling (Ce) |
| **System** | a. Average number of methods per class | a. Number Of class  
b. Total number of methods  
c. Total number of package | a. Total length of inheritance chain |

As this work is aimed at finding the quality attribute status during the early design phase, a set of metrics whose values can be actually obtained at the early design phase is essential. Therefore a new framework called FPROOD is introduced. Table 5.2 shows the FPROOD framework. This framework is distributed across two vectors category and granularity.

At the category level design metric, size metric and complexity metrics are placed. At the granularity level there are class, package and system. Here direct consideration to method
has not been given because to extract method level metrics there is no detail information regarding it in the early design phase. However, if compared to TAPROOT framework it can be observed that a new granularity called package is introduced. The metrics across the package category can collectively used to project the coupling cohesion design property, which is a good indicator of design quality.

Section 5.7 gives the detail exploration of the metric set chosen. But before that a brief overview of the cognitive theory of object-oriented metrics is discussed below.

5.6 Cognitive Theory of Object Oriented Metrics

This theory [9] hypothesizes that the structural properties of a software component (such as its coupling) have an impact on its cognitive complexity. Cognitive complexity is defined as the mental burden of the individuals who have to deal with the component, for example, the developers, the testers, inspectors and maintainers.

High cognitive complexity leads to a component exhibiting undesirable external qualities, such as increased fault proneness and reduced maintainability. Accordingly object oriented product metrics that affect cognitive complexity will be related with fault proneness.

It should be noted that if the cognitive theory is substantiated this could have important implications. It would provide us with a clear mechanism that would explain the introduction of faults into object-oriented application. Figure 5.3 summarizes it.

![Figure 5.3 Theoretical Basis for the Development of O O Product Metrics.](image)

Keeping in mind the cognitive theory and the basic objective of predicting the reliability of object oriented software in early design phase the above set of metrics in table 5.2 has been laid under FPROOD.

5.7 Exploring FPROOD Metric Set

It is well known truth that the later a problem is found in the development life cycle, the more difficult and expensive it is to remove it. Thereby, the identification of potentially problematic modules, before the underlying design decisions are “ossified” into code, is one of the main objectives of the design metrics.
The greater the inheritance relation is, the greater the number of methods a class is likely to inherit, making it more complex and therefore requiring more testing. A method with the complex decision structure will be harder to test and maintain and is more error prone. Complexity metrics based on the above criteria allow to pinpoint potentially troublesome classes, methods or packages, thus helping in planning of the review and test efforts.

Size metrics are usually used in conjunction with complexity metrics and the distinction between them is sometimes unclear. Larger classes or methods are harder to understand, to reuse, to test and to maintain. Class size metrics, for instance, reflect the effort required to build, understand and maintain a class. The same reasoning can be applied to complexity. Size metrics play an important part in the normalization of composite metrics. Without this normalization, it is not meaningful to compare attributes such as total effort, optimal number of test cases and number of failures among several projects. Hence, size metrics help to achieve a baseline for inter project comparison.

5. 7.1 Class Metrics

There are several object-oriented methodologies, whichever is the methodology, design of class is constantly declared to be central to the object oriented paradigm. Class design is the highest priority in OOD, since it deals with the functional requirements of the system; it must occur before system design and program design. Given the importance of class design, the metrics outlined here are specifically designed across the category: design, size and complexity. The design properties mainly concentrated by class metrics are inheritance, complexity, abstraction and design size.

5.7.1.1 Depth of inheritance (DIT):

DIT of a class is the maximum length of all inheritance paths from the class to the root class of its inheritance hierarchy.

The lower DIT is better.

5.7.1.2 Number of Children (NOC):

The number of immediate subclasses of a class in the class hierarchy. A high value may indicate:

1. Better reuse
2. Misuse of subclasses
3. Higher testing effort

So no clear statement which value indicates better statement.

5.7.1.3 Weighted Method Per Class (WMC):
The WMC is the count of methods implemented within a class or the sum of the complexities of the methods. The second method is difficult to implement since not all methods are assessable within the class hierarchy due to inheritance. Therefore in most of the cases where WMC is used as a metric indicator all method complexities are considered to be unity, then WMC = NOM, the number of methods.

5.7.1.4 Coupling Between Object (CBO):

CBO for a class is a count of the number of other classes to which it is coupled. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one-use methods or instance variables of another.

5.7.1.5 Response for a Class (RFC):

RFC gives the number of methods that can potentially be executed in response to a message received by an object of that class [11]. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

5.7.1.6 Number of Methods (NOM):

It counts the total number of methods per class.

5.7.2 Package Metrics

The package metrics considered here revolve mainly around cohesion and coupling principle of package.

5.7.2.1 Package Cohesion Principle

The Release Equivalent Principle (REP)[12]:

The granule of reuse is the granule of release. A reusable element, be it a component or a cluster of classes; cannot be reused unless it is managed by a release system of some kind.

The Common Closure Principle (CCP)[12]:

Classes that change together belong together. When we group classes that change together into the same packages, then the package impact from release to release will be minimized.

The Common Reuse Principle (CRP)[12]:

91
Classes that aren’t used together should not be grouped together. A dependency upon a package is a dependency upon everything within the package. When a package changes, and it’s release number is dumped, all clients of that package must verify that even if they work with new packages nothing they used within the package actually changed.

If classes that are grouped together are not used together then the packages will require the efforts of upgrading and validating. These principles are mutually exclusive.

Several metrics have been proposed in the literature in order to measure class cohesion in object-oriented systems. The major existing class cohesion metrics have been presented in detail and are categorized in [13]. Most of the ways of measuring the cohesion of program fragments are based upon techniques for program slicing. For this we require a working program in hand. If we decide to measure the cohesiveness at design phase we can concentrate at the package architecture. There is no obvious measurement procedure for the level of cohesion in a given module [14].

Here we have concentrated on relational cohesion (H) metric proposed by Martin. This metric captures the cohesion of the classes inside a package. As classes inside a package should be strongly related, the cohesion should be high. H is defined as

\[
H = \frac{R+1}{N} \quad \text{(5.1)}
\]

Where R is the number of relationship between the classes in a package and N is the number of classes in the package.

### 5.7.2.2 Package Coupling Principle

1. **The Acyclic Dependencies Principle (ADP):**

The dependencies between packages must not form cycles. If a cycle is identified between packages involving a new package breaks the cycle.

1. **The Stable Dependencies Principle (SDP) [15]:**

Depend in the direction of stability.

*Stability*

Stability is related to the amount of work required to make a change. One sure way to make a software package difficult to change is to make lots of other packages depend upon it.

A package with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent packages.
Figure 5.4 shows three packages depending upon package X, and therefore it has three good reasons not to change. We say that it is responsible to those three packages. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say it is independent.

Now consider the case in figure 5.5. It shows a package Y that is very instable package. Y has no other packages depending upon it; we say that it is irresponsible. Y also has three packages that it depends upon, so changes may come from these external sources. We say Y is dependent.

**Stability Metrics**

**Afferent Coupling** ($C_a$)

The number of classes outside the package that depend upon classes inside the package (i.e., incoming dependencies)

**Efferent Coupling** ($C_e$)
Chapter 5: Reliability Focused Quality Model for Object Oriented Design

The number of classes outside the package that classes inside the package depends upon. (i.e., outgoing dependencies)

\[
\text{Instability (I)} = \frac{C_e}{C_a + C_e} \quad \text{--------- (5.2)}
\]

This is a metric that has a range [0..1]. If there are no outgoing dependencies then I will be zero. If there are no incoming dependencies then I will be one and the package is instable. “Depend on packages whose I metrics is lower than yours.”

2. The Stable Abstraction Principle (SAP):

Stable packages should be abstract packages.

Explanation:

We have package structure such that instable packages at the top, and the stable packages at the bottom. Packages at the top are instable and flexible, but those at the bottom are very difficult to change. And this leads to dilemma, “Do we want package in our design that are hard to change?”

The highly stable packages at the bottom of the dependency network may be very difficult to change, but according to Open Closed Principle [16] they do not have to be difficult to extend.

If the stable packages at the bottom are also highly abstract, then they can be easily extended. This means that it is possible to compose our application from instable packages that are easy to change, and stable packages that are easy to extend. Thus SAP states that the packages that are most depended upon (i.e., stable) should also be the most abstract. To measure the abstractness we have abstractness metric.

Let \( N_c \) be the number of classes in the package, let \( N_a \) be the number of abstract classes in the package. Then abstractness \( A \) is given by

\[
A = \frac{N_a}{N_c} \quad \text{--------- (5.3)}
\]

This is a metric that has a range [0..1]. A value of zero means that the package contains no abstract classes. A value of one means that the package contains nothing but abstract classes.

The \( A Vs I \) Graph

The SAP can know be related in terms of \( I \) and \( A \) metrics. \( I \) should increase as \( A \) decrease. That is concrete package should be instable while abstract package should be stable.
To understand figure 5.6. Let us consider the following figure 5.7 and decide where to put X on the A-I graph.

The upper right corner of the A-I graph represent packages that are highly abstract and that nobody depends upon. This is the zone of uselessness. We do not want X to leave there.

On the other hand, the lower left point of the A-I represents packages that are concrete and have lot of incoming dependencies. This point represents work case for package. Since the elements there are concrete, they cannot be extended the way abstract entities can; and since they have lots of dependencies, the change will be very painful. The is the zone of pain. So we will not like to place X here also.

Maximizing the distance between these two zones gives us a line called the main sequence. We would like our packages sit on this line if at all possible. A position on this
line means that the package is abstract in portion to its incoming dependencies and is concrete in portion to its outgoing dependencies. In other words, the classes in such a package are confirming to the Dependency Inversion Principle [16]

**Distance metric**

To know how far a package is from the main sequence we have distance metric D.

\[
D = \left( \frac{|A + I - 1|}{\sqrt{2}} \right) \quad \text{(5.4)}
\]

This ranges from [0..~0.707]. The more convenient metric is normalized distance \(D'\).

\[
D' = \left| \frac{A + I - 1}{\sqrt{2}} \right| \quad \text{range [0..1].} \quad \text{(5.5)}
\]

Zero indicates that the package is directly on the main sequence. One indicates that the package is as far away as possible from the main sequence.

### 5.7.3 System Metrics

The following system metrics are used.

**5.7.3.1 Average Number Of Methods Per Class**

It should be less than the specified value. Bigger averages indicate too much responsibility in too few classes.

**5.7.3.2 Number of Classes**

Total number of classes that is present in the system

**5.7.3.3 Total Number of Methods**

It is the count of total number of methods in the system across the various classes.

**5.7.3.4 Total Number of Packages**

It is the total number of packages in the system.

**5.7.3.5 Total Length of Inheritance Chain**

It is the total number of edges in the inheritance hierarchy.

### 5.8 Mapping Between Design Metrics, Properties And Attributes

Table 5.3 shows the mapping between design metrics and design properties and table 5.4 shows the mapping between design metrics and design attributes.
### Table 5.3 Mapping between Design Properties and Metrics

<table>
<thead>
<tr>
<th>Sr.No</th>
<th>Design Property</th>
<th>Design Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Coupling</td>
<td>Instability (I), Afferent coupling (Ca), Efferent coupling (Ce), Abstractness (A), Distance from main sequence (D)</td>
</tr>
<tr>
<td>2.</td>
<td>Cohesion</td>
<td>Number of relationship between classes in a package(R), Relational cohesion (H)</td>
</tr>
<tr>
<td>3.</td>
<td>Inheritance</td>
<td>Depth of inheritance (DIT), Number of Children (NOC), Number of ancestor (NOA)</td>
</tr>
<tr>
<td>4.</td>
<td>Complexity</td>
<td>Weighted method per class (WMC), Number of Method (NOM), Total length of inheritance chain (TIC), Response per class (RFC).</td>
</tr>
<tr>
<td>5.</td>
<td>Abstraction</td>
<td>NOM, DIT</td>
</tr>
<tr>
<td>6.</td>
<td>Design Size</td>
<td>NOM, NOC, NOC in package (N), Number of abstract classes (Na), Total number of packages (TOP)</td>
</tr>
</tbody>
</table>

### Table 5.4 Mapping between Design Metrics and Attributes

<table>
<thead>
<tr>
<th>Metric</th>
<th>Reusability</th>
<th>Testability</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOM</td>
<td>The classes with large number of methods “are likely to be more application specific, limiting the possibility of reuse”</td>
<td>High NOM indicates increased complexity and therefore difficult to maintain.</td>
<td></td>
</tr>
<tr>
<td>DIT</td>
<td>Higher values (percentage of DITs of 2 and 3 would show a higher degree of reuse)</td>
<td>A deep inheritance tree will be difficult to test.</td>
<td>Deeper inheritance tree increases the complexity and therefore difficult the maintenance task.</td>
</tr>
<tr>
<td>CBO</td>
<td>Higher CBO indicates classes that may be less likely for reuse.</td>
<td></td>
<td>Higher CBO indicates classes more difficult to maintain.</td>
</tr>
<tr>
<td>RFC</td>
<td>Larger RFC Makes testing complicated.</td>
<td></td>
<td>Classes with large RFC have a greater complexity hence difficult to maintain.</td>
</tr>
</tbody>
</table>
Chapter 5: Reliability Focused Quality Model for Object Oriented Design

After this metrics are computed they are forwarded to defect density model which has too been revised to be utilized in the early design phase of object oriented design.

5.9 Defect Density Model

As discussed earlier in chapter 2 the existing defect density models are not suitable for predicting the reliability of the object-oriented software in the early design phase. Here a change to the model proposed by Malaiya and Denton has been targeted at Fs. The new factor Fods (factor for object oriented design structure) is computed using the following formula:

\[
\text{Defect Density} = C \times F_{\text{ph}} \times F_{\text{pt}} \times F_{\text{m}} \times F_{\text{ods}} \quad \text{------ (5.6)}
\]

As this research work is aimed at predicting the reliability of object-oriented software as a system in whole the Fph factor is considered to take default value of 1 from table 2.6. The values across the team factor Fpt are taken from table 2.7. Here it is assumed that a single team develops the entire software. The values across Fm factor comes from Table 2.8, where as Fods is given by which has been formulated using information from the object oriented design metrics.

\[
F_{\text{ods}} = \Sigma \text{mean} (\text{WMC+RFC+CBO+DIT+NOM+NOC}) + \text{DMS} - H \quad \text{------ (5.7)}
\]

C in eqn. 5.6 is also having a different meaning from that of the original model. Here C acts as a constant of proportionality. In Malaiya and Denton model [10] C is the number of defects per KLOC, which is estimated using the past data from the same organization.
In eqn. 5.6 C is the number of defects observed in designing object-oriented software by the software designing team of that organization.

Once the defect density is obtained the next task is to predict the reliability, which requires the estimation of following parameters of the exponential model and logarithmic model. It assumes that at any time, the rate of finding (and removing defects) defects is proportional to the number of defects present.

**5.10 Estimating Exponential and Logarithmic Model Parameters**

**5.10.1 Estimating $\beta_0$**

Since the $\beta_0^E$ and $\beta_0^L$ represents the total number of faults that will be detected, it can be estimated using the estimate for the initial defect density, $D_0$ and $D_{\text{min}}$ respectively. As suggested by Musa et al., we can assume that about 5% new defects would be created during debugging. Thus $\beta_0^E$ and $\beta_0^L$ becomes

$$\beta_0^E = 1.05 \times D_0 \times I_s \quad \text{(5.8)}$$

$$\beta_0^L = 1.05 \times D_{\text{min}} \times I_s \quad \text{(5.9)}$$

$I_s = \text{Software Size}$

As this work considers number of classes per package as software size the $I_s$ is replaced by $N_c$. Therefore eq. (5.8) becomes

$$\beta_0^E = 1.05 \times D_0 \times N_c \quad \text{(5.10)}$$

and equation 5.9 becomes

$$\beta_0^L = 1.05 \times D_{\text{min}} \times N_c \quad \text{(5.11)}$$

**5.10.2 Estimating Fault exposure ratio (K)**

K is the fault exposure ratio during the testing period. The value of K is sometimes approximated by $4.2 \times 10^{-7}$ failures per fault, the average value determined by Musa et al. Li and Malaiya have suggested that K varies with the initial defect density and has given the following expression. Agreeing with the same K is estimated using equation 5.10. Here $D_0$ is the defect density per package.

$$K = (1.2 \times 10^{-6} / D_0) \times e^{0.05*D_0} \quad \text{(5.12)}$$

**5.10.3 Estimating $\beta_1$**

$\beta_1^E$ is the constant of proportionality. It is estimated as below.

$$\beta_1^E = (K/(\text{NOM} \times \text{TOC}) \times (1/r)) \quad \text{(5.13)}$$
\[ \beta_1^L = \left( \frac{K_{\text{min}}}{T_L} \right) e^{\frac{(D_0 - D_{\text{min}})}{D_{\text{min}}}} \]  

---(5.14)

Were,
\[ \text{NOM} = \text{Number of methods in the system}, \]
\[ \text{TOC} = \text{Total length of inheritance chain}, \]
\[ r = \text{Object instruction rate of the computer}. \]

Here NOM and TOC are used to represent the source and object instructions that would result after the code development.

### 5.10.4 Estimating Testing Time \( t_1 \)

Originally \( t_1 \) is given by
\[ t_1 = \frac{-\ln (0.1)}{\beta_1^E} \]  

---(5.15)

However this gives the testing time too small for the product’s reliability prediction so the modified \( t_1 \) eqn. 5.16 was used for considerable testing time. The eqn. 5.15 stands appropriate for the other models because they are working out the task late in the implementation phase. But for the model in the early design phase we need to increase the testing time. Here \( t_1 \) is testing time in seconds (CPU time)
\[ t_1 = \frac{-\ln (0.001)}{\beta_1^E} \]  

---(5.16)

### 5.11 Estimating Failure Intensity (\( \lambda \))

\[ \lambda = \beta_0^E \times \beta_1^E \times e^{t_1 \beta_1^E} \]  

---(5.17)

Failure intensity is the indirect measure of reliability. The estimated failure intensity can be utilized for estimating software reliability. Thus if \( t_1 \) is the testing time then \( \lambda(t) \) will give us failures per second.

### 5.12 Estimating Reliability (\( R \))

\[ R = e^{-\lambda t_1} \]  

---(5.18)

Here \( R \) finds the reliability across each package. Once we obtain the reliability per package the software reliability can be predicted for the entire system by taking the mean of all the package reliability. Thus using the design metrics one can predict the reliability of the software in the early design phase.

RFQMood tool is developed to predict the reliability of an object-oriented software in the early design phase. The details of the tool are shown in appendix E.
Chapter 5: Reliability Focused Quality Model for Object Oriented Design

The RFQMOOD model starts with computing the object oriented design metrics and providing it as an object design structure to compute defect density. Once the defect density is found, failure intensity is estimated and therefore the reliability of the object oriented software. Thus this model can be used for predicting the reliability of object oriented software in the early design phase.

Reference