CHAPTER 5

INPUT AND OUTPUT DATA FOR THE DEFECT PREDICTION MODEL
INPUT AND OUTPUT DATA FOR THE DEFECT PREDICTION MODEL

5.1 NEED FOR ANALYSIS OF DATA

Prediction of defects / errors in software systems supports software quality engineering through improved scheduling and project control. It can be a key step towards steering the Verification and Validation activities and improving the effectiveness of the whole process. It enhances the V&V activities by focusing on error prone critical components and can be used to improve software process control and achieve high reliability software. In short, prediction of errors in software systems can be used to direct cost effective quality enhancement efforts to development [12]. For this the data available has to be analyzed critically and the relevant input and output data has to be identified.

Neural Network were studied as a means of capturing the intricate interrelationship between various input factors and the output which are mainly the number of defects in the end product, be it a document or code. The biggest advantage of Neural Network is, it can be trained to learn.

5.2 FRAMEWORK FOR CAPTURING DIFFERENT ACTIVITIES

After studying the defects that are detected during the V & V cycle, we can fit a framework [7]. A study of the development environment of any software shows that the following activities take place as part of software development life cycle. This is pictorially depicted in Figure 5.2.1
The diagram is to be read from left to right. The designer/tester use interface (X) to view and manipulate Information (Y), to realise the product (Z).

During Requirements Phase, the designer uses Functional Requirements and employing or utilizing his software knowledge as well as domain expertise, generates the Software Requirements Specification (SRS). Similarly during the Design Phase, the designer uses the Software Requirement Specification document along with his software knowledge and domain expertise, designs the software modules for each specification and generates the Software Design Document (SDD). Next comes the implementation
phase, where the programmer or software designer uses the SRS, SDD and coding guidelines along with his language knowledge, domain knowledge and previous coding experience generates the code. During the above three phases, the designer can be the same person or different persons. Similarly in the testing phase, the test engineer uses SRS, SDD and code and employs his domain knowledge, previous experience and language knowledge to create effective test cases to test the code and detect errors.

A study of the defects detected reveals, that the defects are introduced in all phases of the SDLC, but the V&V Cycle ensures most of the defects are detected as early as possible. Implementation of a Defect Prevention Mechanism depends on the defect logging, documentation, root cause analysis, defect prevention learning and implementation of a defect prevention scheme for future Projects.

5.2.1 Phases of Defect Introduction

Most of the activities of the Defect Prevention Methodology requires a facilitator and the Quality Assurance agency acts as facilitator. Developing this model for quality assessment is one way to predict the number of defects / errors and organize activities to prevent the occurrence of these defects.

According to the Computer Finance Magazine, errors in software requirements and software design documents are more frequent than errors in the source code itself. Percentage of defects introduced in software in each phase is given in Table 5.2.1.1
Table 5.2.1.1 Percentage of Defects introduced in Software in each Phase

<table>
<thead>
<tr>
<th>Software Development Phase</th>
<th>Percentage (%) of defect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>20%</td>
</tr>
<tr>
<td>Design</td>
<td>25%</td>
</tr>
<tr>
<td>Coding</td>
<td>35%</td>
</tr>
<tr>
<td>User Manual</td>
<td>12%</td>
</tr>
<tr>
<td>Error Correction</td>
<td>8%</td>
</tr>
</tbody>
</table>

Defects introduced during the requirements and design phases are not only more probable but also more severe and more difficult to remove. Front end errors in requirements and design can be detected by reviews and inspections much before code testing.

### 5.2.2 Types of Software Defects:

All software defects, that have been encountered or discovered by examination or operation of the software product have been analyzed. An analysis of the defects detected shows the different types of possible software defects are:

**Requirements defect:** A mistake made in the definition or specification of the customer needs of a software product. This includes defects found in functional specifications; interface, design, and test requirements; and specified standards.

**Design defect:** A mistake made in the design of a software product. This includes defects found in functional descriptions, interfaces, control logic, data structures, error checking, and standards.


**Code defect:** A mistake made in the implementation or coding of a program. This includes defects found in program logic, interface handling, data definitions, computation, and standards.

**Document defect:** A mistake made in a software product publication. This includes mistakes made to requirements, design, or coding documents.

**Test case defect:** A mistake in the test case causes the software product to give an unexpected result.

**Other work product defect:** Defects found in software artifacts that are used to support the development or maintenance of a software product. This includes test tools, compilers, configuration libraries, and other computer-aided software engineering tools.

**5.3 Analysis of Error Data**

The driving force behind this study is that, a great deal about the quality and reliability of software can be said from its defect history. From a study of the empirical data collected from different missions over different phases of development, it is clear the raw count of defects do not narrate the whole story about reliability and quality. Further analysis is required to determine the types of defects, when and where they were introduced, how they were detected and their impact on the software operation.

Several studies whose focus is on improving software reliability and quality support the conclusion that poorly understood or documented interfaces between software modules are a major cause of software error [133]. In a study of software redundancy, it was
found that inadequate understanding on the part of the programmers of the specifications or the underlying coordinate system was a major contributor to the program faults. Systems which had high interactions with other sub-systems, had proportionately more errors than less interactive systems.

In large embedded systems, the software requirements change throughout the software development process, even during system testing. This is largely due to unanticipated behavior, complex hardware / software interfaces and software / software interfaces in the system being developed. A Satellite Launch Vehicle is a similar system.

The general input data available were the total source code, language used, documentation standard, design guidelines, coding guidelines, errors detected during requirements and design reviews, code inspection and testing, design level testing, independent testing - black box and white box testing, integration and system testing, designer experience, tester experience, use of tools for testing etc.

The input data was acquired from different launch vehicle flights for the four identified phases. The factors were different in different phases. In the requirements phase, the root cause analysis of defects could identify 27 factors as possible causes. A critical analysis confirmed the possibility of reducing these factors to 4 important factors which accounts for 90% of the variability of defects detected in the software components used for the study. Similarly in the design phase too there were 19 factors which could again be reduced to 4. In the coding and testing phases there were 7 factors each, which were
also reduced to 4 again by classifying them under the four categories namely – complexity of the problem, documentation adequacy, human factor and processes employed.

5.3.1 Software Requirements phase

This phase can be called as the ‘problem analysis phase’ – the user requirements are analyzed and the Software Requirements Specification (SRS) generated must be as complete, consistent and correct as possible. An analysis of the errors detected revealed that they are mainly requirements error, untraceable requirements, and wrong/incomplete or inconsistent requirements due to documentation problems. If the number of review recommendations is very high, it reflects on the quality of documentation, extent of analysis carried out and complexity of the problem. If there are missing or incomplete or inconsistent requirements, it is again dependent on the quality of the documentation, communication between different teams, domain understanding of the programmer. If untraceable requirements are existing, it reflects on the understanding of the systems development methods employed for realizing the software, and the mode of interaction among the teams.

As we can see, the errors in SRS are of different types. On tracing back the reason for occurrence of these errors or looking at the causes resulting in these errors in the Requirements Phase, we can say there are 27 factors contributing to these. These could be categorized into 4 main factors such as- complexity of the problem, adequacy of the documentation, the human factors and the process flaws.
5.3.1.1 Complexity Factor $R_C$

The complexity of the problem decides to a great extent the defects that occur in the SRS. If the problem is fairly simple, the defects that are likely to occur is few. But in a large embedded system, the complexity is large and the probability of introducing errors is also large. There are five parameters which can give an insight into the complexity of the problem.

a. Size of the document $R_{C1}$

The size of the document factor $R_{C1}$ feeding into complexity is given in the range 0 to 0.99 as given below. For the software components considered, the size and complexity are given in Table 5.3.1.1.

<table>
<thead>
<tr>
<th>Pages</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-30</td>
<td>0.25</td>
</tr>
<tr>
<td>30-50</td>
<td>0.50</td>
</tr>
<tr>
<td>50-70</td>
<td>0.75</td>
</tr>
<tr>
<td>&gt;70</td>
<td>0.99</td>
</tr>
</tbody>
</table>

b. System Functional Requirement Documentation Adequacy $R_{C2}$

If all the system functions are documented and clear, then the compliance factor is assigned 0.99. If there are ‘a’ functional requirement and ‘b’ are missing, then the compliance factor $R_{C2} = \frac{a-b}{a}$
c. Completeness of requirements identification $R_{C3}$

If some of the requirements are not identified or missed or wrong are denoted as ‘c’ out of a total number of ‘d’ number of requirements, then completeness $R_{C3} = \frac{d-c}{d}$

d. Identification of Interfaces $R_{C4}$

Similarly the number of interfaces not adequately identified or understood, which resulted in the errors can be calculated. If there are ‘a’ interfaces out of which ‘b’ are not defined, then $R_{C4} = \frac{a-b}{a}$

e. Understanding hardware functioning $R_{C5}$

Improper understanding of the hardware factors like interrupts, timers, memory etc could have caused the defect. This may be also calculated as above. Identify the total hardware functions and identify the numbers missed out. If there are ‘a’ hardware functions out of which ‘b’ are not defined, then $R_{C5} = \frac{a-b}{a}$

Thus we can see that there are five factors which give an insight into the complexity of the problem. Let each factor be denoted as $R_{Ci}$, ranging from 1 to 5. Then the input factor $R_C$ is given as $R_C = \sum_{i=1}^{n} \frac{R_{Ci}}{n}$ where, ‘n’ is 5

5.3.1.2 Documentation Adequacy $R_D$

The Requirements analysis and the SRS document generated is the starting phase of the SDLC. If all the software requirement specifications are not documented correctly and completely, there will be errors in the design and code realised. So the adequacy of documents plays an important role.
There are ten factors $R_{D1}$ to $R_{D10}$ feeding into this factor. Similar to the range given for computing $R_C$, this is also calculated on a range of 0 to 0.99.

a. Format of the document $R_{D1}$

If the document is following a standard format like IEEE or in-house format, its compliance factor $R_{D1}$ is 0.99, else it is 0.5. These templates allow the requirements analyst to focus on the content and helps to ensure key items are not overlooked.

b. Compliance to ISPD format $R_{D2}$

ISPD is the ISRO Software Process Document Standard. If all the sections given in the ISPD template are adhered to, then the factor assigned is 0.99. If there are ‘a’ number of sections out of which ‘b’ number are missed, then compliance factor

$$R_{D2} = \frac{b-a}{b}$$

c. Clarity of expression $R_{D3}$

If the statements used for defining conditions, limits etc are unambiguous and the requirements are documented clearly then the factor is 0.99. If ‘a’ requirements out of ‘b’ requirements are unclear, then the factor $R_{D3} = \frac{b-a}{b}$

d. Imprecise or unsystematic specification $R_{D4}$

While specifying the requirements, if some of the specifications are described in an unsystematic manner or are not very precise in description, $R_{D4}$ is computed as
follows. Out of ‘a’ specifications if ‘b’ are imprecise or unsystematically documented, then \( R_{D4} = \frac{a-b}{a} \)

e. Adequacy of description \( R_{D5} \)

If the descriptions given in the SRS are not adequate to convey the required meaning, those descriptions are documented as errors. If there are ‘d’ specifications inadequately documented out of ‘c’ specification, the factor of compliance \( R_{D5} = \frac{c-d}{c} \)

f. Clear definition of assumptions & dependencies \( R_{D6} \)

Often the assumptions and dependencies while specifying the requirements are in the designer’s mind and are not documented. This will result in wrong design if the designer is another person. So this factor plays an important role in the defects that can creep into the design. This is also computed as, out of ‘n’ number of assumptions and dependencies, if ‘m’ are missed out, then \( R_{D6} = \frac{n-m}{n} \)

g. Constraints, if any \( R_{D7} \)

In any system, there will be certain constraints which have to be accounted while specifying the requirements. Unless clearly stated, during the design phase this might be missed. Hence if there are ‘p’ constraints and ‘q’ are missed out, then \( R_{D7} = \frac{p-q}{p} \)
h. Defining all interfaces $R_{D8}$

In any complex system, there will be software to hardware interfaces, software to software interfaces. A good SRS will specify all these interfaces clearly so that no description is missed out. But often in SRS reviews, it is observed out of ‘f’ interfaces ‘e’ are either not described or not complete and then the factor $R_{D8} = \frac{f-e}{f}$

i. Error Handling defined $R_{D9}$

Most of the embedded system have many error handling features so that even under error conditions, the system behaviour is in a defined manner. So it is required that in SRS all error handling requirements are also defined so that subsequent design and implementation will include these. So if there are ‘x’ error handling requirements and ‘y’ are missed out, then $R_{D9} = \frac{x-y}{x}$

j. Incomplete documentation $R_{D10}$

In an SRS if ‘a’ requirements are to be specified, quite often some of the requirements are not completely documented or some of the inputs and outputs descriptions are partial. So out of the ‘a’ requirements if ‘b’ are not complete then $R_{D10} = \frac{a-b}{a}$

Thus Documentation adequacy factor $R_D$ is the sum total of all factors from a to j. Documentation adequacy is denoted by $R_D = \sum_{i=1}^{n} \frac{R_{Di}}{n}$ where, ‘n’ is 10
5.3.1.3. Human Factor $R_H$

A defect made due to human factor during software development is detected as a failure. Tracing back, the human error reveals an inappropriate work system, inadequate programming skills, misunderstanding of the requirements etc. The human factor $R_H$ is dependent on the following 9 factors. The software designers’ knowledge and capability play a great role in the errors created in a particular phase of the SDLC. These are defined as Human errors. A detailed analysis of the defects in SRS revealed the following factors which could be attributed to designer’s mistake play a major role.

a. Domain Experience $R_{H1}$

If the designer has previous domain experience, he will be able to elicit the software requirements correctly from the functional requirements. Misunderstanding due to complexity of the problem, wrong assumptions made and some of the requirements not properly analyzed are the common mistakes made. If the constraints and interfaces are not proper, he will be able to elicit them and include them in the SRS based on earlier experience.

b. Technical skill $R_{H2}$

Without necessary knowledge of the system and necessary skills for performing the requirements analysis, he will not be able to understand and write the software requirements specifications properly.
c. Previous Track Record $R_{H3}$

This is very important because if the designer has had a history of delivering software products which contained large number of defects, there is every chance this work also will be having numerous errors.

d. Credits $R_{H4}$

This factor is important because basic training and qualification play a major role in cognitive ability and analytic ability of the software designer.

e. Productivity $R_{H5}$

Productivity is the ability of the designer to generate good SRS under time constraints with minimum errors. Quality of the code generated within a stipulated time depends on the productivity of the designer.

f. Attention to details $R_{H6}$

Attention to details is an important factor. A designer who does not pay much attention to details is likely to generate an SRS which will have many deficiencies.

g. Tools familiarity $R_{H7}$

If tools are used for Requirements Analysis, and if the designer is not familiar with the tools, the turnaround time will be large. Also if the notations and usage are not properly understood, the document generated may not be correct.
h. Team player $R_{H8}$

Team work is important in any project. If a designer is willing to participate in the reviews and discussions with others in the team, the requirements elicitation will be more complete.

i. Communication $R_{H9}$

A designer with good communications skills will be able to clarify all doubts and express the requirements clearly.

For each of the above factors a to i, the Project leader can give grades for the designer as A, B, C where A is excellent and is 0.99, B is good and 0.75 and C is poor and has 0.5. The Human factor $R_H = \sum_{i=1}^{n} \frac{R_{Hi}}{n}$ where ‘n’ is 9.

5.3.1.4 Process Factors $R_P$

Inadequacy of the processes chosen for requirements elicitation, have also been the cause of some of the defects. An analysis shows that where excellent processes were adopted the numbers of errors introduced were less. The factors identified for Process Flaws are:

a. Planning $R_{P1}$

Planning is very important in any phase of activity. Without proper planning, the end product will not be realized correctly.
b. Model adopted for analysis $R_{P2}$

The adequacy of the model adopted plays an important role in the correctness and completeness of the SRS generated. If the model adopted cannot map certain requirements, the SRS will be having deficiencies.

c. Information Elicitation $R_{P3}$

The information elicitation is dependent on the process adopted. Information elicitation requires that all relevant information required for the software to function correctly is documented correctly. If the process employed is capable of capturing all information, then the SRS generated will be good.

d. Traceability Establishment $R_{P4}$

This is mandatory to ensure no functional requirement is missed out. A systematic method adopted to map all functional requirements ensures that traceability is established between functional requirements and SRS. There should be no extra requirement or untraceable requirement.

For all the factors a to d, a grading similar to Human error was applied here which was graded by the Project Leader. If A is assigned, then it indicates the factor is 0.99, for B the factor is 0.75 and for C the factor is 0.5.

The process factor $R_P = \sum_{i=1}^{n} \frac{R_{Pi}}{n}$ where ‘n’ is 4
5.3.2 Software Detailed Design Phase

Software Design is a process of problem solving and planning for a software solution. Software design documentation describes how the software will meet the requirements and the output of this phase is a description of the program suitable for implementation in a specific language. The total number of defects detected by reviewing the design document is an index of the quality of the product. If the number of review recommendations are large, then it indicates that the design is not clear, consistent or complete. Design defects are wrong design, incomplete or inadequate designs. A design module which cannot be traced to requirements is also a defect.

An analysis of the defects detected during the design phase reveals that they are of the form of design errors, untraceable to requirements, wrong or incomplete design. The input factors contributing to these errors are analyzed and there are 19 factors identified as the causes for these errors. They are correlated into four factors which are identified as – complexity of the problem, adequacy of documentation, the human factors and the process flaws.

5.3.2.1 Complexity Factor $D_C$

The complexity of the problem can be a reason for the defects in this phase and is denoted by $D_C$. The factors feeding into this phase which can contribute in causing the errors are:
a. Size of the document $D_{C_1}$

The size of the document is an indication of the complexity of the problem. The factor $D_{C_1}$ which is related to document complexity is assigned as given in Table 5.3.2.1.

<table>
<thead>
<tr>
<th>Pages</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-50</td>
<td>0.25</td>
</tr>
<tr>
<td>50-100</td>
<td>0.5</td>
</tr>
<tr>
<td>100-300</td>
<td>0.75</td>
</tr>
<tr>
<td>&gt;300</td>
<td>0.99</td>
</tr>
</tbody>
</table>

b. Traceability of all functional and interface requirements $D_{C_2}$

If the traceability process is inadequate, all the software requirements will not be mapped into the software design. If the interfaces are not understood correctly, software may not function correctly. As given in the requirements phase, out of ‘a’ functional & interface requirements, the requirements which are not mapped into design are identified as ‘b’ and the factor of compliance is computed as $D_{C_2} = \frac{a-b}{a}$

c. Unchanged requirements $D_{C_3}$

Quite often, all the functional requirements and interface requirements may not change from one mission to another. Very often some of them may be changed, while a few may be modified and quite a large number may be old. If there are ‘a’ functional & interface requirements, and the number of requirements unchanged are ‘c’, the factor of compliance is computed as $D_{C_3} = \frac{c}{a}$
d. Interpretation of requirements $D_{C4}$

Misunderstanding of requirements, often result in design errors. Also inappropriate documentation for module calling methods and file/global variable definitions, often cause errors. The misunderstanding of hardware interfaces and method of hardware accessing were typical common errors. Similarly, if there are ‘a’ functional & interface requirements, and during review it is observed that ‘d’ requirements are wrongly interpreted, the factor of compliance is computed as $D_{C4} = \frac{a-d}{a}$

Thus cumulative Design complexity factor $D_C$ is computed as follows

$$D_C = \sum_{i=1}^{n} \frac{D_{Ci}}{n}$$

where ‘n’ is 4

5.3.2.2 Documentation Adequacy $D_D$

These errors occur while translating the requirements specifications into design. The requirements as understood by the requirements engineer may not be understood properly by the designer; may be due to inadequacy or lack of clarity of the requirements document. And this may result in errors while generating the Software Detailed Design. The factors which will result in design defects due to documentation can be attributed to the following factors.

a. Format of the document $D_{D1}$

If a standard format is followed, there will no lapses or mistakes in the document. If the document is following a standard format like IEEE or in-house format, its compliance factor $D_{D1}$ is 0.99, else it is 0.5. These templates allow the requirement analyst to focus on the content and help to ensure key items are not overlooked
b. Compliance to ISPD format $D_{D2}$

If the ISPD standard is followed, chances of missing requirements are less. Also the design will be done in a logical way. Similarly standard notations shall be used. If all the sections given in the ISPD template are adhered to, then the factor assigned is 0.99. If ‘a’ number of sections out of ‘b’ number is missed, then compliance factor $D_{D2} = \frac{b-a}{b}$

c. Code and Date Structure defined $D_{D3}$

The architecture design should represent all the program and data structure. This is to ensure all the requirements are translated into suitable structure and documented in the design document. If there are ‘n’ code and data structures and during review of SDD it is observed that ‘m’ code and data structure are not defined, then $D_{D3} = \frac{n-m}{n}$

d. Processing defined $D_{D4}$

In the design document, if the processing of all the modules given in the structure chart are not defined, the number of modules for which the processing is not defined or inadequately described is noted. Out of ‘p’ modules if processing of ‘q’ modules is not described, then $D_{D4} = \frac{p-q}{p}$

e. Data Dictionary $D_{D5}$

A data dictionary has to be described in the design document giving all the required details. If there are ‘n’ data variables and ‘m’ are missing, then $D_{D5} = \frac{n-m}{n}$

The cumulative factor $D_D$ for design phase documentation adequacy is computed as

$$D_D = \sum_{l=1}^{n} \frac{D_{Di}}{n}$$

where ‘n’ is 5
5.3.2.3. Human Factor $D_H$

The Human factors which are important and which play a decisive role in deciding the number of defects that are likely to be introduced in the software design are given below. Each factor is assigned grades A, B, C by the project leader where $A=0.99$, $B=0.75$ and $C=0.5$.

a. Clear understanding of requirements $D_{H1}$

The requirements as documented in SRS might not be clearly understood by the designer if he is working in that domain for the first time. Basic cognitive ability of designer also plays a role in improving this factor.

b. Discipline to design guidelines $D_{H2}$

This is an indication whether the designer adheres to best practices defined in the organization. If there is no discipline to design guidelines, chances of design errors are more.

c. Experience $D_{H3}$

Experience plays a role in understanding the domain, order of events, not making wrong assumptions, not leaving out any requirements. Specific application knowledge is very important.

d. Technical skills $D_{H4}$

If the designer does not possess the required technical skills, the requirements will not be translated correctly into the design.
Quality of Design $D_{H5}$

Quality of Design is a factor which indicates whether design is carried out to minimize resource utilization, whether scalability is kept in mind and proven modules are reused.

The human factor $D_H$ in design phase is computed as

$$D_H = \sum_{l=1}^{n} \frac{D_{Hl}}{n}$$

where ‘n’ is 5

Next the design processes are analyzed to find the flaws that can result in defects.

### 5.3.2.4 Process Factor $D_P$

These errors are due to inadequacy of the processes chosen for software design. The process flaws which can result in defects are identified as given below:

a. Modular Design $D_{P1}$

   The architectural model should define the module structure. As far as possible the program should be modular for better planning, better accommodation of changes and effective testing and debugging. If out of ‘n’ modules, it is observed that ‘m’ modules can be further split into modules for better understanding of design, then

   $$D_{P1} = \frac{n-m}{n}$$

b. Defining Error Handling $D_{P2}$

   If the system requirements are not understood properly, the ability to predict the exact behaviour of software for all kind of inputs and for different states will be missed. While reviewing SDD, if out of ‘x’ error handling requirements, only ‘y’ are addressed in the design document, then

   $$D_{P2} = \frac{x-y}{x}$$
c. Interfaces of Hardware and Software $D_{P3}$

Often misunderstanding of software interfaces have resulted in inappropriate documentation of calling methods or global definitions. Also lack of understanding of hardware interfaces have resulted in wrong hardware accessing or handling. During review of SDD, if out of ‘m’ hardware and software interfaces, ‘n’ are missed in the SDD, then $D_{P3} = \frac{m-n}{m}$

d. Function Faults $D_{P4}$

These were mainly due to misunderstanding of system functions and module functions. Out of ‘c’ modules if ‘d’ modules were not meeting the system and module function, then $D_{P4} = \frac{c-d}{c}$

e. Suitable for language of implementation $D_{P5}$

The design adopted for implementation has to be suitable for language of implementation. This factor is assigned 0.99 if design is suitable or 0.5 if not suitable.

From the above factors a to e, the Design Process factor $D_P$ is computed as

$$D_P = \sum_{i=1}^{n} \frac{D_{P_i}}{n}$$

where $n = 5$
### 5.3.3 Coding Phase

The programmer gains a full understanding of the problem and generates the code. This is the phase when the software design is translated into a set of codes which becomes the final executable code.

Analysis of defects in the program reveals that they fall into the category of implementation errors, design errors, wrong requirements capturing, non-optimum way of implementation, dead or redundant code. The total number of defects given as inspection observations is an index of the quality of the product. If the number of review recommendations is large, then it indicates that the code in not well-written.

A root cause analysis of the errors detected during code inspection reveals the following inputs have an impact on the defects in the code - complexity of the problem, the human factor, language of implementation and adherence to coding guidelines.

#### 5.3.3.1 Complexity Factor $C_c$

As in requirements and design phases, if the problem is complex, the code also will be complex. An analysis of the complexity of the code reveals the following factors contribute in increasing the complexity.

a. **Size of the code**: $C_{C_{1}}$

   This is a simple metric which is assessed by the Lines of Code (LOC). If the code size is large the probability of errors likely to be introduced in code is also more.

Table 5.3.3.1 gives the complexity factor linked with LOC.
### Table 5.3.3.1 Complexity Factors and LOC

<table>
<thead>
<tr>
<th>LOC</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;5000</td>
<td>0.9</td>
</tr>
<tr>
<td>2000-5000</td>
<td>0.75</td>
</tr>
<tr>
<td>&lt;2000</td>
<td>0.5</td>
</tr>
</tbody>
</table>

b. Maximum cyclomatic complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. It provides the number of independent paths in a module of the code. If the module is complex, the chances of making errors are more. The cyclomatic complexity of the modules is assessed through static analysis. The number of modules having complexity more than 15 are taken as ‘a’ and the total number of modules are taken as ‘b’. Maximum cyclomatic complexity factor is computed as $C_{C_{2}} = \frac{a}{b}$

The coding phase complexity error is quantified as $C_{c} = \sum_{i=1}^{n} \frac{C_{ci}}{n}$ where ‘n’ is 2

### 5.3.3.2 Human Factor $C_{H}$

As stated in earlier phase, the experience and domain knowledge of the designer plays an important role in the correctness of code generated. While translating the design into code, the misunderstanding of module functions can result in omissions or wrong interpretation of the relationship of the module to the system functions. The Human factors is graded by the project leader for designer as A,B,C as is done in the requirements and design phase, for the following factors where A=0.99, B=0.75 and C=0.5
a. Language / system familiarity $C_{H1}$

If the person is not knowledgeable about the system/language, the program will not be error free. Good programming skills are essential for generating good code.

b. Domain expertise $C_{H2}$

Lack of experience or domain knowledge of the programmer will result in wrong assumptions regarding system states, pre-conditions and post conditions, misunderstanding about timing constraints or timing relationships among commands.

c. Understanding/Cognitive skills $C_{H3}$

Mistakes in applying the process without considering whether it is adequate for the task at hand, order of execution/storage mistakes depends on basic cognitive skills of the programmer.

The Human factor $C_H$ is computed using factors a to c as follows.

\[
C_H = \frac{C_{H1} + C_{H2} + C_{H3}}{3}
\]

5.3.3.3 Programming Language $C_L$

In flight software both assembly language and high level language are used. Assembly language is used for real time operating system where hardware / software interfaces are many. Chances of errors in assembly language are more, especially when handling interrupts, interfacing with other hardware. The factor $C_L$ is assigned as follows.

The factor $C_L = 0.9$, for Assembly language
$C_L = 0.8$, for High level language.
5.3.3.4 Coding Guidelines $C_G$

A systematic and disciplined programmer will write a program adhering to coding guidelines which are essentially good coding practices. Adherence to coding guidelines factor is computed based on the number of violation of coding guidelines. If the violations are less than 2, then compliance is 0.9. If the violations are between 2 & 5, then compliance is 0.8. If the violations are greater than 5, $C_G$ is 0.7. Details are given in Table 5.3.3.4

<table>
<thead>
<tr>
<th>No of guidelines violated</th>
<th>Compliance factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 2</td>
<td>0.9</td>
</tr>
<tr>
<td>2-5</td>
<td>0.8</td>
</tr>
<tr>
<td>&gt; 5</td>
<td>0.7</td>
</tr>
</tbody>
</table>

5.3.4 Testing Phase

Testing is the process of executing a program with the intention of finding an error. There is unit testing, integrated testing and system testing. At unit level, black box and while box testing is conducted. At integrated level the program is fully integrated and tested. At system level testing, the software is loaded onto the actual hardware and tested.

In the testing phase, the errors detected are classified as follows:

- Functional Errors – Not meeting the functional requirements
- Structural Error- Infeasible path, dead code, redundant codes
• System level errors- Not meeting the system level requirements, software to software interface mismatches or hardware to software mismatches and improper handling of hardware interrupts and wrong communication related.

The causes for defects in these phases were analyzed and the following input factors are identified

5.3.4.1 Black box testing $T_B$
The effectiveness of black box testing is computed based on the number of defects detected during black box testing of each module. If there are ‘b’ Black box test cases and the number of defects detected are denoted as ‘a’ and the factor $T_B$ compliance is computed as $T_B = \frac{a}{b}$

5.3.4.2 White box testing $T_W$
The effectiveness of white box testing is computed based on the number of defects detected during white box testing of each module. If there are ‘d’ White box test cases and the number of defects detected are denoted as ‘c’ and the factor $T_W$ compliance is computed as $T_W = \frac{c}{d}$

5.3.4.3 Human Factor $T_H$
In testing also human factors have a major role. Testing is as creative as designing. Planning the testing activity, creating good test cases all help in uncovering defects effectively. The Human factors for the test engineer as graded by the project leader are A,B,C where A=0.99, B=0.75 and C=0.5
a. Tester experience $T_{H1}$

If the testing engineer has previous testing experience, he will be good at designing test cases. Frequently missed areas by designers are out of bounds values, loop control errors, forgotten steps or cases. An experienced testing engineer can focus his effort to get maximum errors with minimum test cases.

b. Domain knowledge $T_{H2}$

As in design, good domain knowledge helps in designing effective test cases. Test cases which can uncover domain specific errors are possible only if the testing engineer has an equal or better understanding of the domain as the designer.

c. Language familiarity $T_{H3}$

This is required to detect language specific problems, which can result in errors.

d. Tool familiarity $T_{H4}$

Good understanding of testing tools will result in faster testing and better detection of errors.

Using the factors a to d, the cumulative Human factor $T_H$ in Testing phase is computed as follows

$$T_H = \sum_{i=1}^{n} \frac{T_{Hi}}{n}$$

where ‘$n$’ is 4
5.3.4.4 System Level Testing $T_S$

The defects detected during the testing with software loaded into target hardware come under this category. Various simulations with near flight conditions are also done as part of system level testing. Mainly hardware related error handling inadequacies and timing related anomalies are detected. If there are ‘$d$’ System level test cases and the number of errors detected are denoted as ‘$c$’ and the factor $T_S$ compliance is computed as $T_S = \frac{c}{d}$.

5.4. Data Identification for Defect Prediction

The defects in different phases of SDLC and their causes are analysed and understood. All the input factors responsible for inducing errors in different phases of SDLC are also identified. Next step is to prevent the occurrence of such defects. Defect prevention means learning from actual defects or error data, with the goal of developing specific plans to prevent defects from occurring in the future.

In this thesis, experience from different projects in flight software development has been employed. Projects employing iterative development offer a platform, as each iteration is complete in itself and can be analysed as a project. Software quality and reliability are crucial issues for embedded system. For any software organisation, the management would like to have some knowledge of the quality of the target software product as early as possible. For this a software defect prediction model is needed which can predict the number of defects in different phases of the Software Development Life Cycle.