MODULE 4

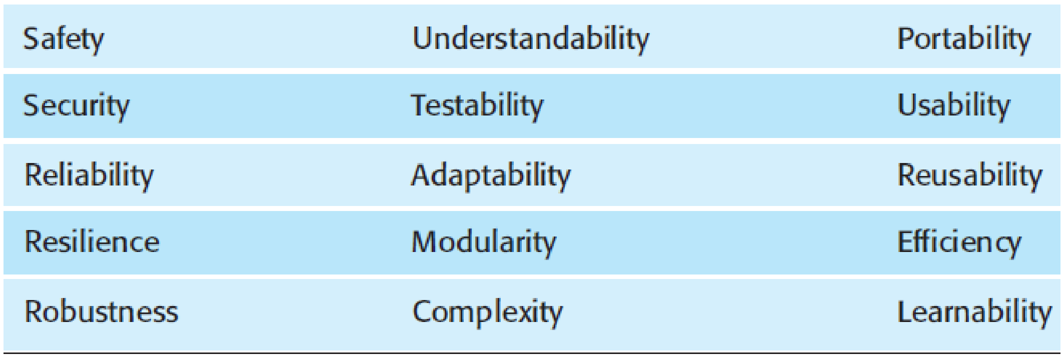
CHAPTER 2: QUALITY MANAGEMENT

2.1 Software Quality

- Software quality is the degree of conformance to explicit or implicit requirements and expectations.
- The assessment of software quality is a subjective process where the quality management team has to use their judgment to decide if an acceptable level of quality has been achieved. The quality management team has to consider whether or not the software is fit for its intended purpose. This involves answering questions about the system's characteristics.
- For example:
    1. Have programming and documentation standards been followed in the development process?
    2. Has the software been properly tested?
    3. Is the software sufficiently dependable to be put into use?
    4. Is the performance of the software acceptable for normal use?
    5. Is the software usable?
    6. Is the software well-structured and understandable?

- Software quality is not just about whether the software functionality has been correctly implemented, but it also depends on some of the nonfunctional requirements.

There are 15 important software quality attributes as shown in the below Figure 2.1



Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Figure 2.1 Software quality attributes

Process-Based quality

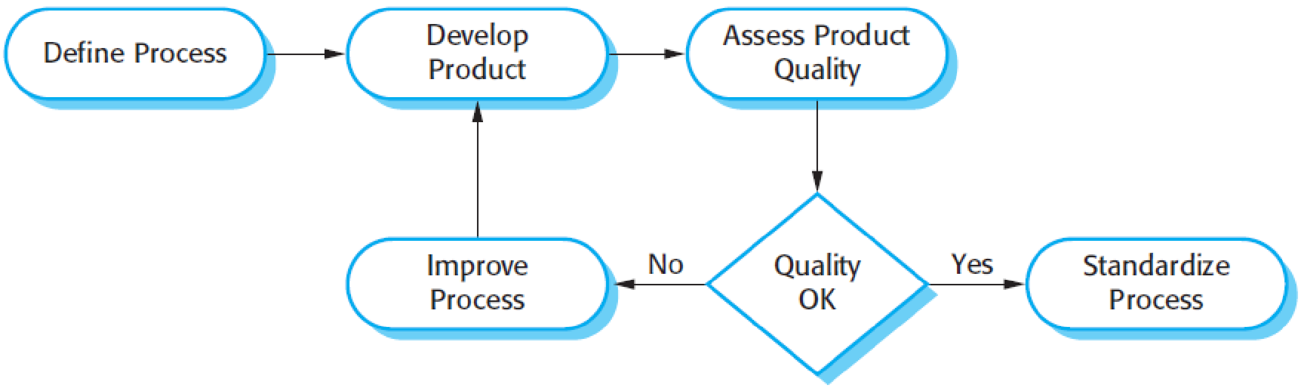


Figure 2.2 Process based quality

- In the above figure 2.2, it explains about the process-based approach to achieve product quality.
- A manufacturing process involves configuring, setting up, and operating the machines involved in the process. Once the machines are operating correctly, product quality naturally follows. Then measure the quality of the product and change the process until you achieve the quality level that you need.

2.2 Software standards

- Software standards play a very important role in software quality management.
- Standards should be applied to the software product or the software development process.
- Different tools and methods to support the use of these standards may be chosen.
- Software standards are important because:
    1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company.
    2. Standards provide a framework for defining what „quality‟ means in a particular setting.
    3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices.

There are two types of software engineering standards as shown in the figure10:

1. Product standards:

- These apply to the software product being developed.
- Product standards include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

2. Process Standards:

- These define the processes that should be followed during software development. They should encapsulate good development practice.
- Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Figure 2.3 Product and process standards

2.2.1 The ISO 9001 standards framework

- There is an international set of standards that can be used in the development of quality management systems in all industries, called ISO 9000.
- ISO 9001 applies to organizations that design, develop, and maintain products, including software. The ISO 9001 standard was originally developed in 1987, with its most recent revision in 2008.
- The ISO 9001 standard is not itself a standard for software development but is a framework for developing software standards. It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined.

- This framework consists of 9 core processes as shown in the Figure 2.5

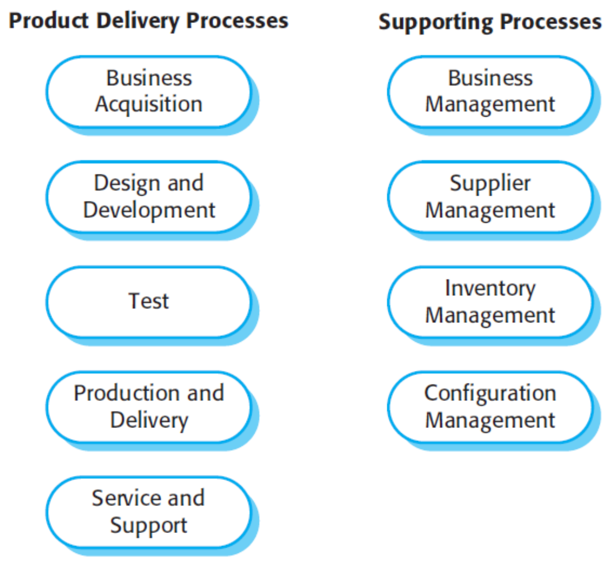


Figure 2.4: ISO 9001 core processes

- If an organization is to be ISO 9001 conformant, it must document how its processes relate to these core processes.
- It must also define and maintain records that demonstrate that the defined organizational processes have been followed.
- To be conformant with ISO 9001, a company must have defined the types of process shown in Figure 2.4 and have procedures in place that demonstrate that its quality processes are being followed. This allows flexibility across industrial sectors and company sizes.
- The relationships between ISO 9001, organizational quality manuals, and individual project quality plans are shown in Figure 2.5

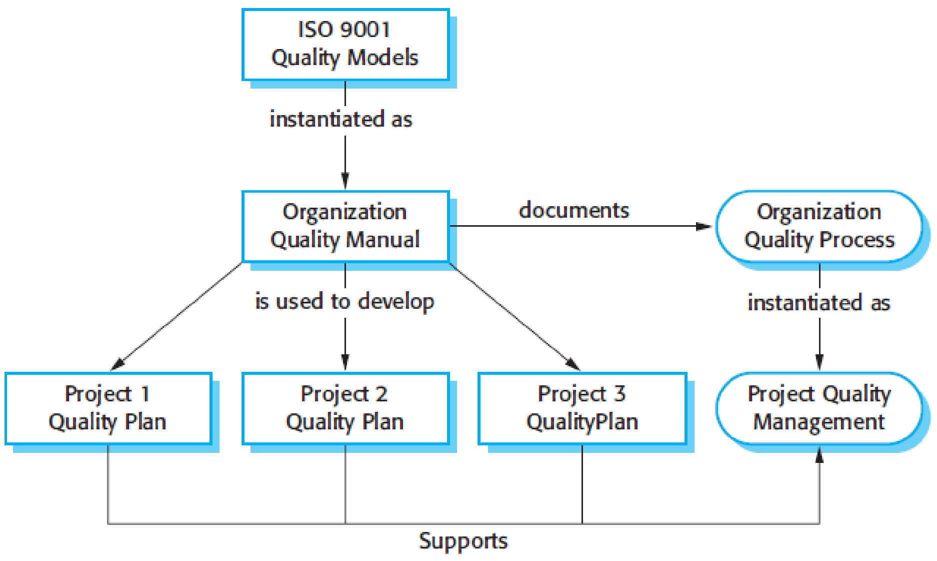


Figure 2.5: ISO 9001 and quality management

- This diagram explains how the ISO 9001 framework can be used as basis for software quality management process.
- Some software customers demand that their suppliers should be ISO 9001 certified. The customers can then be confident that the software development company has an approved quality management system in place.
- Some people think that ISO 9001 certification means that the quality of the software produced by certified companies will be better than that from uncertified companies. This is not necessarily true.

2.3 Reviews and Inspections

- Reviews and inspections are quality assurance (QA) activities that check the quality of project deliverables.
- This involves examining the software, its documentation and records of the process to discover errors and omissions and to see if quality standards have been followed.
- Reviews are not just about checking conformance to standards. They are also used to help discover problems and omissions in the software or project documentation.
- The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.

2.3.1 The review process

The review process is structured into three phases

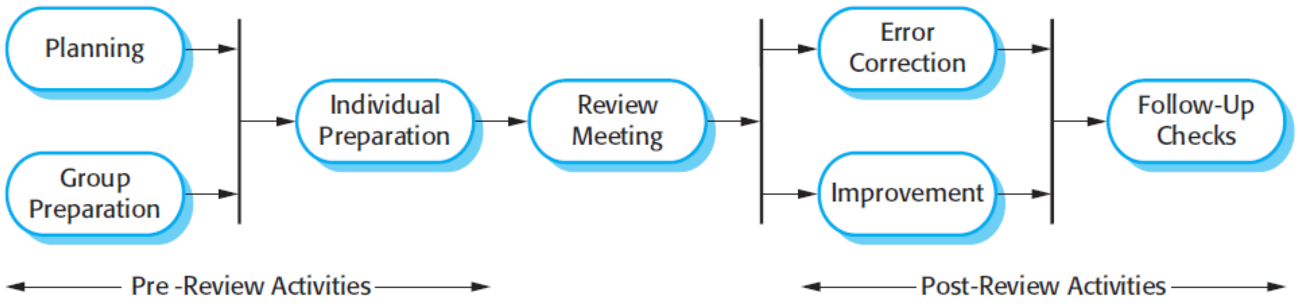


Figure 2.6: The software review process

1. Pre-review activities:

- These are preparatory activities that are essential for the review to be effective. Pre-review activities are concerned with review planning and review preparation.
- Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed.
- During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant standards. They work independently to find errors, omissions, and departures from standards.
- Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. The review meeting:

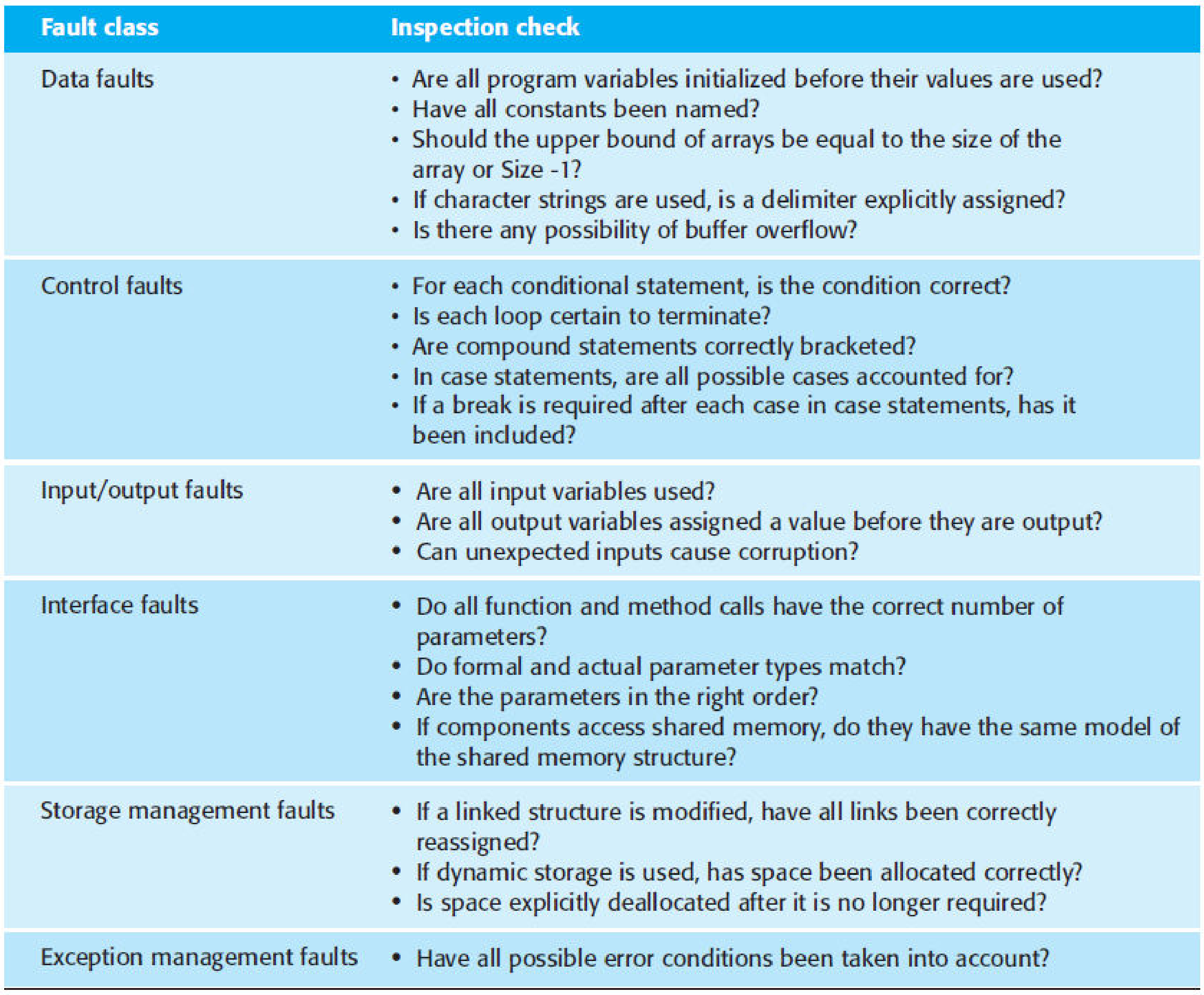
- During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team.
- The review should be short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken.

3. Post-review activities:

- After a review meeting has finished, the issues and problems raised during the review must be addressed. This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.
- After changes have been made, the review chair may check that the review comments have all been taken into account. Sometimes, a further review will be required to check that the changes made cover all of the previous review comments.

2.3.2 Program inspections

- Program inspections are 'peer reviews' where team members collaborate to find bugs in the program that is being developed.
- Program inspections involve team members from different backgrounds who make a careful, line-by-line review of the program source code.
- They look for defects and problems and describe these at an inspection meeting.
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or features that have been omitted from the code.
- The review team examines the design models or the program code in detail and highlights anomalies and problems for repair.
- During an inspection, a checklist of common programming errors is used to focus the search for bugs as shown in Figure 2.7



Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Figure 2.7: An inspection checklist

- These checklists should be regularly updated, as new types of defects are found. The items in the checklist vary according to programming language because of the different levels of checking that are possible at compile-time.
- For example, a Java compiler checks that functions have the correct number of parameters but a C compiler does not.

2.4 Software Measurement and Metrics

Software measurement

- Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process.
- By comparing these values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software.
- For example, say an organization intends to introduce a new software-testing tool. Before introducing the tool, record the number of software defects discovered in a given time. After using the tool for some time, repeat this process. If more defects have been found in the same amount of time, after the tool has been introduced, then you may decide that it provides useful support for the software validation process.
- The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality.

Software metric

- A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured.
- Examples of metrics include the size of a product in lines of code, number of reported faults in a delivered software product, the number of person-days required to develop a system component.

There are two types of software metrics.

1. **Control (Process) metrics**
   - Control metrics are usually associated with software processes control metrics support process management.
   - Examples of control or process metrics are the average effort and the time required to repair reported defects.
2. **Predictor metrics**
   - Predictor metrics are associated with the software itself and are known as ‘product metrics’ which helps to predict characteristics of the software
   - Examples of predictor metrics are the cyclomatic complexity of a module, the average length of identifiers in a program, and the number of attributes and operations associated with object classes in a design.

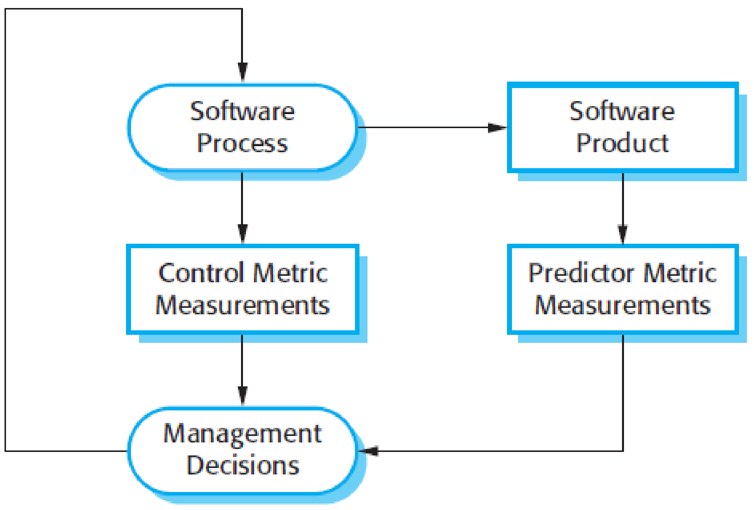


Figure 2.8 Predictor and control measurements

Both control and predictor metrics may influence management decision making, as shown in Figure 2.8.
Managers use process measurements to decide if process changes should be made, and predictor metrics to help estimate the effort required to make software changes

<u>There are two ways in which measurements of a software system may be used:</u>

1. **<u>To assign a value to system quality attributes:</u>** By measuring the characteristics of system components, such as their cyclomatic complexity (it is the total number of control statements used in a program), and then aggregating these measurements, you can assess system quality attributes, such as maintainability.

2. **<u>To identify the system components whose quality is substandard Measurements:</u>** can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

Figure 2.9 shows the relationship between some external software quality attributes and internal attributes.

- Quality attributes such as maintainability, understandability and usability are external attributes that relate to how developers and users experience the software.
- Internal attributes are related to the quality characteristics that one is concerned with size, complexity.

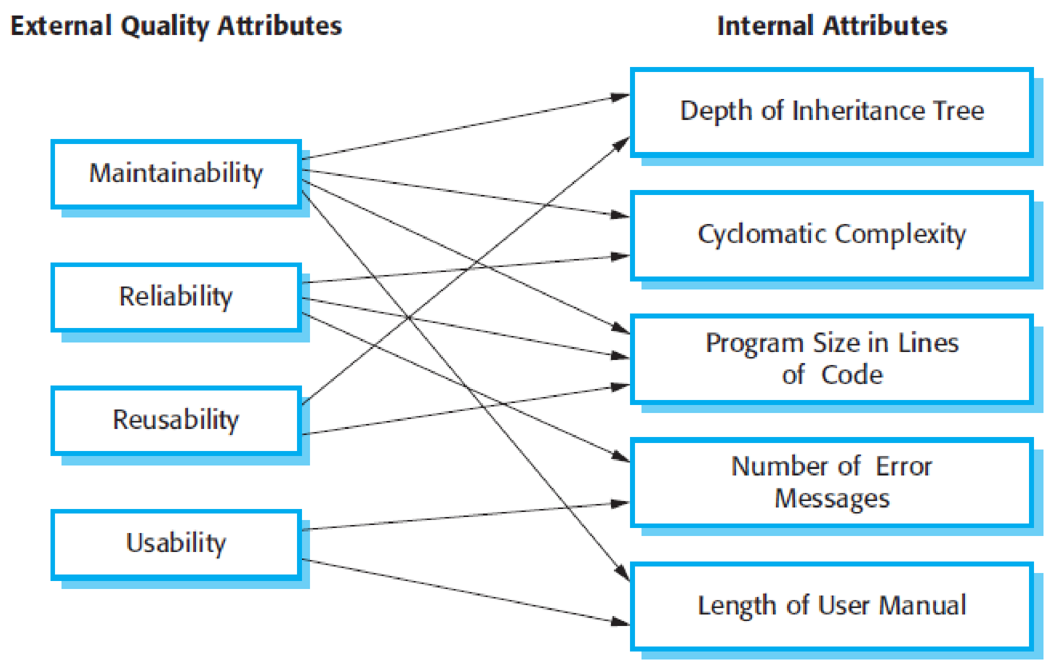


Figure 2.9: Relationships between internal and external software

If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold:

1. The internal attribute must be measured accurately.
2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related to the value of the attribute that can be measured.
3. This relationship between the internal and external attributes must be understood, validated, and expressed in terms of a formula or model

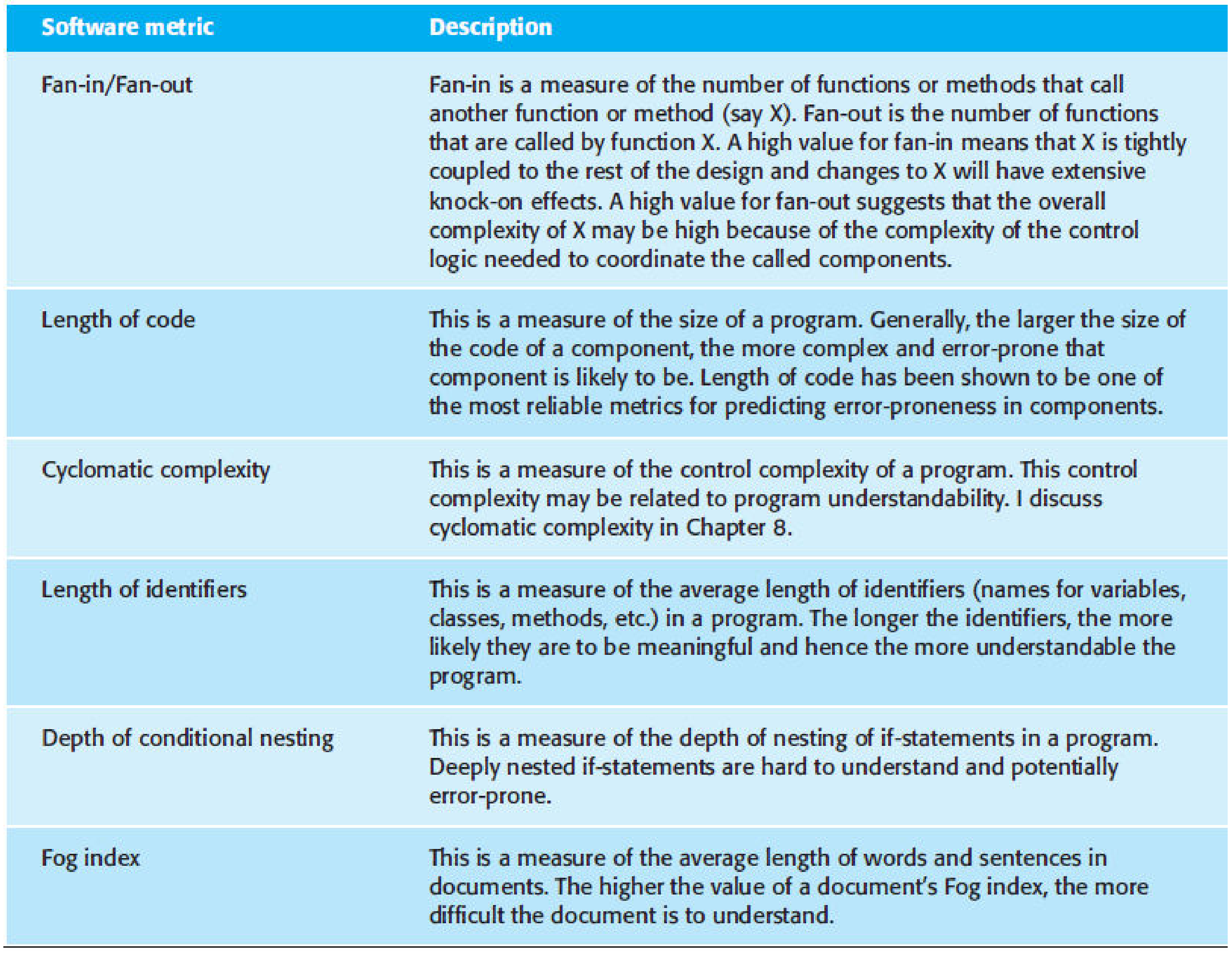
2.4.1 Product metrics

Product metrics are predictor metrics that are used to measure internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.

Product metrics fall into two classes:

1. **Dynamic metrics**: which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.
2. **Static metrics**: which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used

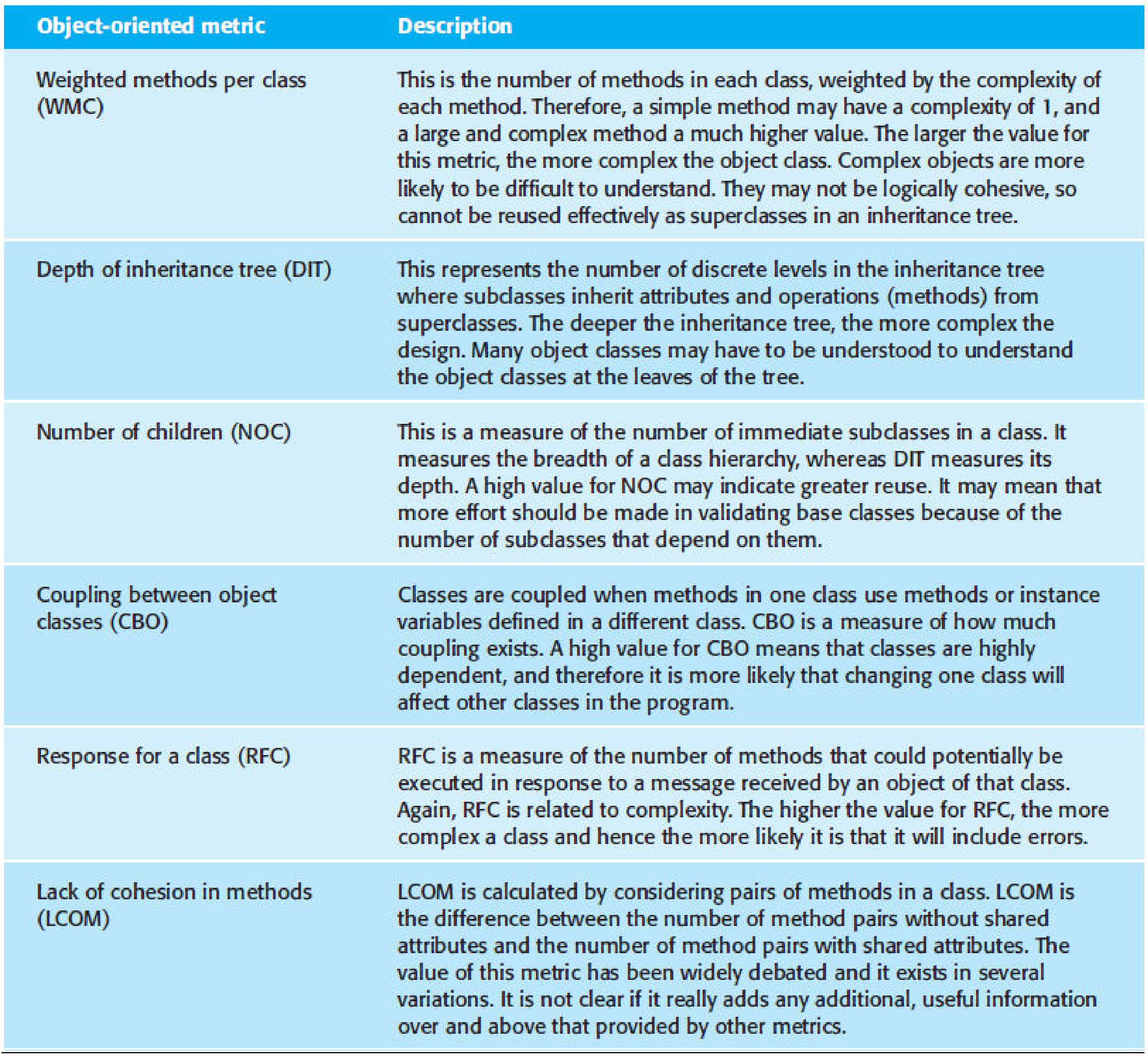
The metrics in Figure 2.10 are applicable to any program but more specific object oriented (OO) metrics have also been proposed.



Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Figure 2.10: Static software product metrics

Figure 2.11 summarizes Chidamber and Kemerer"s suite (sometimes called the CK suite) of six object oriented metrics (1994).



Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Figure 2.11: The CK object-oriented metrics suite

2.4.2 Software component analysis

Each system component can be analyzed separately using a range of metrics.
The key stages in this component measurement process are shown in the Figure 2.12.

1. **Choose measurements to be made:** The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined.
2. **Select components to be assessed**: You may not need to assess metric values for all of the components in a software system. Sometimes, you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality.
3. **Measure component characteristics**: The selected components are measured and the associated metric values computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool.

4. **<u>Identify anomalous measurements</u>**: After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric.
5. **<u>Analyze anomalous components</u>**: When you have identified components that have anomalous values, you should examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor quality component. There may be some other reason for the high value, so may not mean that there are component quality problems.

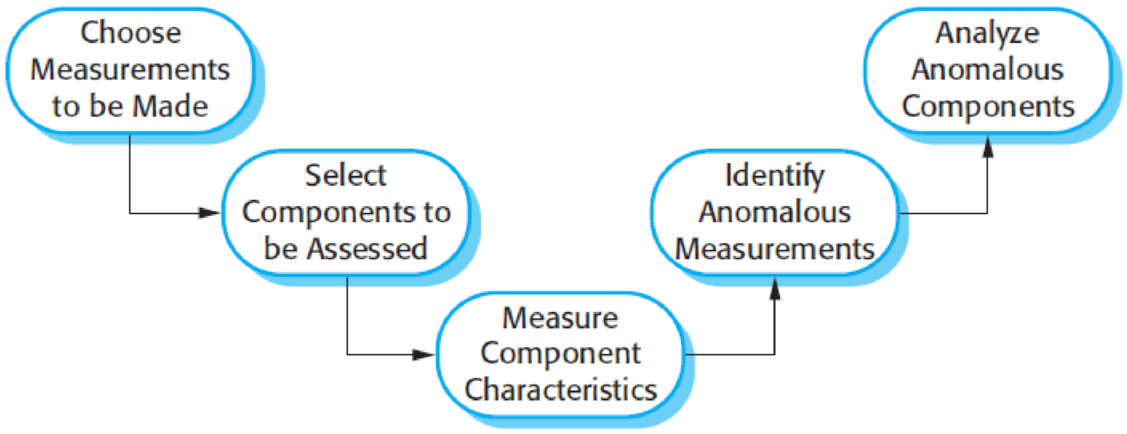

Choose Measurements to be Made
Select Components to be Assessed
Measure Component Characteristics
Identify Anomalous Measurements
Analyze Anomalous Components

Figure 2.12: The process of product measurement