13 Software Quality

Learning Objectives

- Explain the importance of software quality to software users and developers
- Define the qualities of good software
- Design methods of measuring the required qualities of software
- Monitor the quality of the processes in a software project
- Use external quality standards to ensure the quality of software acquired from an outside supplier
- Develop systems using procedures that will increase their quality

13.1 Introduction

While quality is generally agreed to be 'a good thing', in practice what is meant by the 'quality' of a system can be vague. We need to define precisely what qualities we require of a system. However, we need to go further – we need to judge objectively whether a system meets our quality requirements and this needs measurement. This would be of particular concern to someone like Brigette at Brightmouth College in the process of selecting a package.

For someone – like Amanda at IOE – who is developing software, waiting until the system exists before measuring it would be leaving things rather late. Amanda might want to assess the likely quality of the final system while it was still under development, and also to make sure that the development methods used would produce that quality. This leads to a different emphasis – rather than concentrating on the quality of the final system, a potential customer for software might check that the suppliers were using the best development methods.

This chapter examines these issues.

13.2 The Place of Software Quality in Project Planning

Quality will be of concern at all stages of project planning and execution, but will be of particular interest at the following points in the Step Wise framework (Figure 13.1).

- *Step 1: Identify project scope and objectives* Some objectives could relate to the qualities of the application to be delivered.
- *Step 2: Identify project infrastructure* Within this step, activity 2.2 identifies installation standards and procedures. Some of these will almost certainly be about quality.
- *Step 3: Analyze project characteristics* In activity 3.2 ('Analyze other project characteristics – including quality based ones') the application to be implemented is examined to see if it has any special quality requirements. If, for example, it is safety critical then a range of activities could be added, such as *n*-version development where a number of teams develop versions of the same software which are then run in parallel with the outputs being cross-checked for discrepancies.

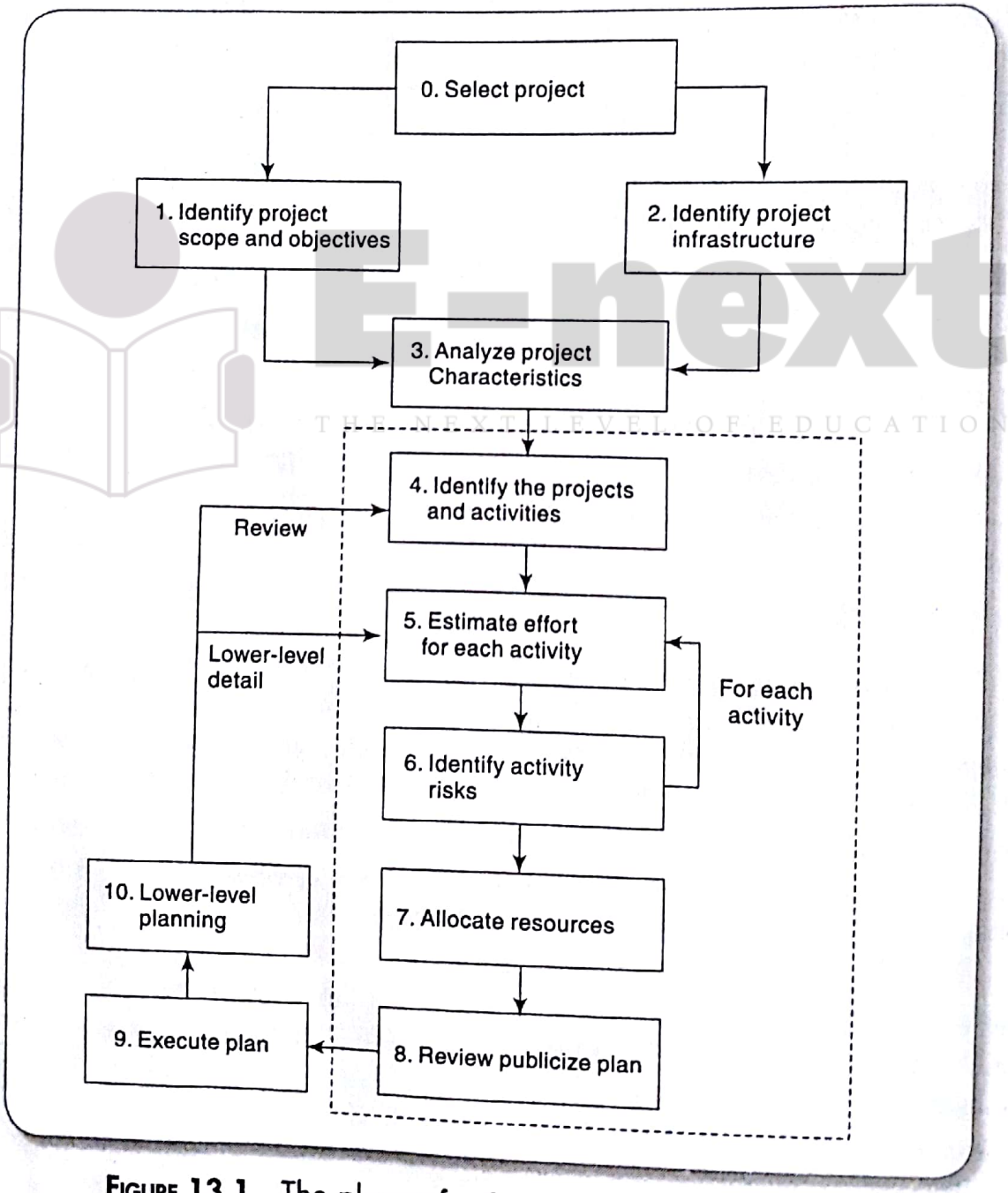

FIGURE 13.1 The place of software quality in Step Wise

- *Step 4: Identify the products and activities of the project* It is at this point that the entry, exit and process requirements are identified for each activity. This is described later in this chapter.
- *Step 8: Review and publicize plan* At this stage the overall quality aspects of the project plan are reviewed.

13.3 Importance of Software Quality

We would expect quality to be a concern of all producers of goods and services. However, the special characteristics of software create special demands.

- *Increasing criticality of software* The final customer or user is naturally anxious about the general quality of software, especially its reliability. This is increasingly so as organizations rely more on their computer systems and software is used in more safety-critical applications, for example to control aircraft.
- *The intangibility of software* can make it difficult to know that a project task was completed satisfactorily. Task outcomes can be made tangible by demanding that the developer produce 'deliverables' that can be examined for quality.
- *Accumulating errors during software development* As computer system development comprises steps where the output from one step is the input to the next, the errors in the later deliverables will be added to those in the earlier steps, leading to an accumulating detrimental effect. In general, the later in a project that an error is found the more expensive it will be to fix. In addition, because the number of errors in the system is unknown, the debugging phases of a project are particularly difficult to control.

For these reasons quality management is an essential part of effective overall project management.

13.4 Defining Software Quality

In Chapter 1 we noted that a system has functional, quality and resource requirements. Functional requirements define what the system is to do, the resource requirements specify allowable costs and the quality requirements state how well this system is to operate.

Exercise 13.1

At Brightmouth College, Brigette has to select the best off-the-shelf payroll package for the college. How should she go about this in a methodical manner?

One element of the approach could be the identification of criteria against which payroll packages are to be judged. What might these criteria be? How could you check the extent to which packages match these criteria?

Some qualities of a software product reflect the external view of software held by *users*, as in the case of usability. These *external qualities* have to be mapped to *internal factors* of which the *developers* would be aware. It could be argued, for example, that well-structured code is likely to have fewer errors and thus improve reliability.

Defining quality is not enough. If we are to judge whether a system meets our requirements we need to be able to measure its qualities.

The BS ISO/IEC 15939:2007 standard *Systems and software engineering – measurement process* has codified many of the practices discussed in this section.

A good measure must relate the number of units to the maximum possible. The maximum number of faults in a program, for example, is related to the size of the program, so a measure of *faults per thousand lines of code* is more helpful than *total faults in a program*.

Trying to find measures for a particular quality helps to clarify and communicate what that quality really is. What is being asked is, in effect, 'how do we know when we have been successful?'

The measures may be *direct*, where we can measure the quality directly, or *indirect*, where the thing being measured is not the quality itself but an indicator that the quality is present. For example, the number of enquiries by users received by a help desk about how one operates a particular software application might be an indirect measurement of its usability.

When project managers identify quality measurements they effectively set targets for project team members, so care has to be taken that an improvement in the measured quality is always meaningful. For example, the number of errors found in program inspections could be counted, on the grounds that the more thorough the inspection process, the more errors will be discovered. This count could, of course, be improved by allowing more errors to go through to the inspection stage rather than eradicating them earlier – which is not quite the point.

When there is concern about the need for a specific quality characteristic in a software product then a quality specification with the following minimum details should be drafted:

- *Definition/description:* definition of the quality characteristic
- *Scale:* the unit of measurement
- *Test:* the practical test of the extent to which the attribute quality exists
- *Minimally acceptable:* the worst value which might be acceptable if other characteristics compensated for it, and below which the product would have to be rejected out of hand
- *Target range:* the range of values within which it is planned the quality measurement value should lie
- *Now:* the value that applies currently

Exercise 13.2

Suggest quality specifications for a word processing package. Give particular attention to the way that practical tests of these attributes could be conducted.

There could be several measurements applicable to a quality characteristic. For example, in the case of reliability, this might be measured in terms of:

- *Availability:* the percentage of a particular time interval that a system is usable
- *Mean time between failures:* the total service time divided by the number of failures
- *Failure on demand:* the probability that a system will not be available at the time required or the probability that a transaction will fail
- *Support activity:* the number of fault reports that are generated and processed

Exercise 13.3

The enhanced IOE maintenance jobs system has been installed, and is normally available to users from 8.00 a.m. until 6.00 p.m. from Monday to Friday. Over a four-week period the system was unavailable for one whole day because of problems with a disk drive and was not available on two other days until 10.00 in the morning because of problems with overnight batch processing runs.

What were the availability and the mean time between failures of the service?

Associated with reliability is *maintainability*, which is how quickly a fault, once detected, can be corrected. A key component of this is *changeability*, which is the ease with which the software can be modified. However, before an amendment can be made, the fault has to be diagnosed. Maintainability can therefore be seen as changeability plus a new quality, *analysability*, which is the ease with which causes of failure can be identified.

Maintainability can be seen from two different perspectives. The user will be concerned with the *elapsed time* between a fault being detected and it being corrected, while the software development managers will be concerned about the *effort* involved.

13.5 Software Quality Models

The need to be able to quantitatively measure the quality of a software is often felt. For example, one may want to set quantitative quality requirements for a software, or to verify whether a software meets the quality requirements set of it. Unfortunately, it is hard to directly measure the quality of a software. However, it can be expressed in terms of several attributes of a software that can be directly measured. The quality models give a characterization (often hierarchical) of software quality in terms of a set of characteristics of the software. The bottom level of the hierarchy can be directly measured, thereby, enabling a quantitative assessment of the quality of the software. There are several well-established quality models, including McCall's, Dromey's and Boehm's. Since there was no standardization among the large number of quality models that became available, the ISO 9126 model of quality was developed. We briefly discuss Garvin's, McCall's, Dromey's and Boehm's quality models in this section and discuss ISO 9126 in the next section.

Garvin's quality dimensions: David Garvin, a professor of Harvard Business school in his book, *Total Quality Management*, defined the quality of any product in terms of eight general attributes of the product, some of these are measurable and some are not. Garvin reasoned that sometimes users have subjective judgment of the quality of a program (perceived quality) that must be taken into account to judge its quality.

- **Performance:** How well it performs the jobs.
- **Features:** How well it supports the required features.
- **Reliability:** Probability of a product working satisfactorily within a specific period of time.
- **Conformance:** Degree to which the product meets the requirements.
- **Durability:** Measure of the product life
- **Serviceability:** Speed and effectiveness maintenance.
- **Aesthetics:** The look and feel of the product.
- **Perceived quality:** User's opinion about the product quality.

McCall' model: McCall defined the quality of a software in terms of three broad parameters: its operational characteristics, how easy it is to fix defects and how easy it is to port it to different platforms. These three high-level quality attributes are defined based on the following eleven attributes of the software:

- **Correctness:** The extent to which a software product satisfies its specifications.
- **Reliability:** The probability of the software product working satisfactorily over a given duration.
- **Efficiency:** The amount of computing resources required to perform the required functions.
- **Integrity:** The extent to which the data of the software product remains valid.
- **Usability:** The effort required to operate the software product.
- **Maintainability:** The ease with which it is possible to locate and fix bugs in the software product.
- **Flexibility:** The effort required to adapt the software product to changing requirements.
- **Testability:** The effort required to test a software product to ensure that it performs its intended function.
- **Portability:** The effort required to transfer the software product from one hardware or software system environment to another.
- **Reusability:** The extent to which a software can be reused in other applications.
- **Interoperability:** The effort required to integrate the software with other software.

Dromey's model: Dromey proposed that software product quality depends on four major high-level properties of the software: Correctness, internal characteristics, contextual characteristics and certain descriptive properties. Each of these high-level properties of a software product, in turn, depends on several lower-level quality attributes of the software. Dromey's hierarchical quality model is shown in Figure 13.2.

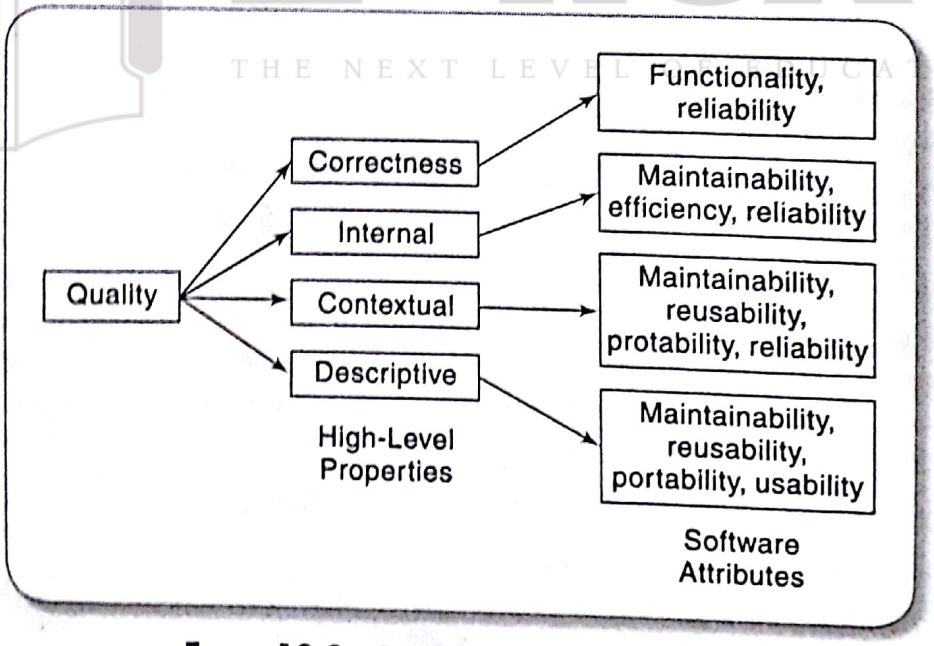

FIGURE 13.2 Dromey's quality model

Boehm's model: Boehm postulated that the quality of a software can be defined based on three high-level characteristics that are important for the users of the software. These three high-level characteristics are the following:

- **As-is utility:** How well (easily, reliably and efficiently) can it be used?

- **Maintainability:** How easy is it to understand, modify and then retest the software?
- **Portability:** How difficult would it be to make the software in a changed environment?

Boehm expressed these high-level product quality attributes in terms of several measurable product attributes. As compared to McCall's and Dromey's quality models, Boehm's quality model is based on a wider range of software attributes and with greater focus on software maintainability. Boehm's hierarchical quality model is shown schematically in Figure 13.3.

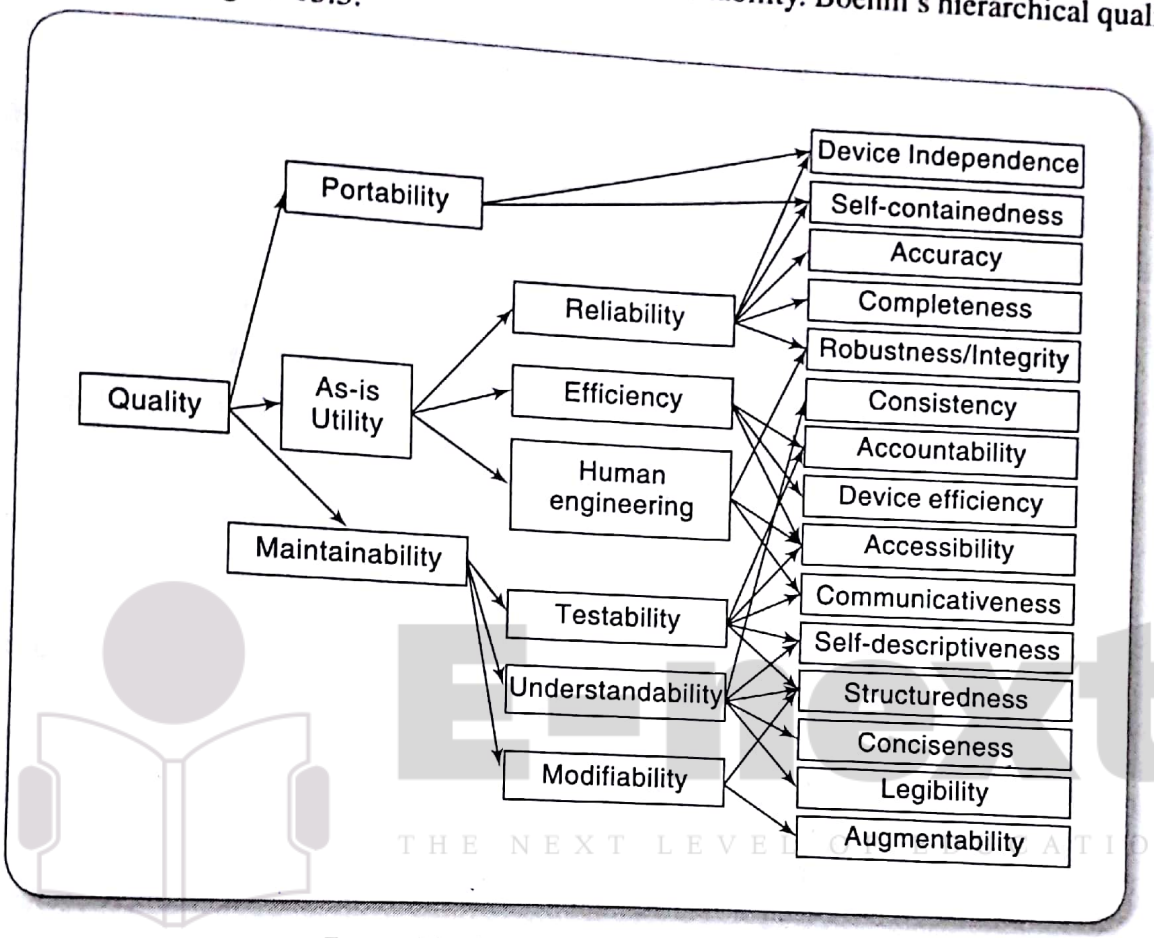

FIGURE 13.3 Boehm's quality model

13.6 ISO 9126

Over the years, various lists of software quality characteristics have been put forward, such as those of James McCall and of Barry Boehm. A difficulty has been the lack of agreed definitions of the qualities of good software. The term 'maintainability' has been used, for example, to refer to the ease with which an error can be located and corrected in a piece of software, and also in a wider sense to include the ease of making any changes. For some, 'robustness' has meant the software's tolerance of incorrect input, while for others it has meant the ability to change program code without introducing errors. The ISO 9126 standard was first introduced in 1991 to tackle the question of the definition of software quality. The original 13-page document was designed as a foundation upon which further, more detailed, standards could be built. The ISO 9126 standards documents are now very lengthy. Partly this is because people with differing motivations might be interested in software quality, namely:

Currently, in the UK, the main ISO 9126 standard is known as BS ISO/IEC 9126-1:2001. This is supplemented by some 'technical reports' (TRs), published in 2003, which are provisional standards. At the time of writing, a new standard in this area, ISO 25000, is being developed.

- *Acquirers* who are obtaining software from external suppliers
- *Developers* who are building a software product
- *Independent evaluators* who are assessing the quality of a software product, not for themselves but for a community of users – for example, those who might use a particular type of software tool as part of their professional practice

ISO 9126 has separate documents to cater for these three sets of needs. Despite the size of this set of documentation, it relates only to the definition of software quality attributes. A separate standard, ISO 14598, describes the procedures that should be carried out when assessing the degree to which a software product conforms to the selected ISO 9126 quality characteristics. This might seem unnecessary, but it is argued that ISO 14598 could be used to carry out an assessment using a different set of quality characteristics from those in ISO 9126 if circumstances required it.

The difference between internal and external quality attributes has already been noted. ISO 9126 also introduces another type of quality – *quality in use* – for which the following elements have been identified:

- *Effectiveness:* the ability to achieve user goals with accuracy and completeness
- *Productivity:* avoiding the excessive use of resources, such as staff effort, in achieving user goals
- *Safety:* within reasonable levels of risk of harm to people and other entities such as business, software, property and the environment
- *Satisfaction:* smiling users

'Users' in this context includes not just those who operate the system containing the software, but also those who maintain and enhance the software. The idea of quality in use underlines how the required quality of the software is an attribute not just of the software but also of the context of use. For instance, in the IOE scenario, suppose the maintenance job reporting procedure varies considerably, depending on the type of equipment being serviced, because different inputs are needed to calculate the cost to IOE. Say that 95% of jobs currently involve maintaining photocopiers and 5% concern maintenance of printers. If the software is written for this application, then despite good testing, some errors might still get into the operational system. As these are reported and corrected, the software would become more 'mature' as faults become rarer. If there were a rapid switch so that more printer maintenance jobs were being processed, there could be an increase in reported faults as coding bugs in previously less heavily used parts of the software code for printer maintenance were flushed out by the larger number of printer maintenance transactions. Thus, changes to software use involve changes to quality requirements.

ISO 9126 identifies six major external software quality characteristics:

- *Functionality*, which covers the functions that a software product provides to satisfy user needs
- *Reliability*, which relates to the capability of the software to maintain its level of performance
- *Usability*, which relates to the effort needed to use the software
- *Efficiency*, which relates to the physical resources used when the software is executed
- *Maintainability*, which relates to the effort needed to the make changes to the software
- *Portability*, which relates to the ability of the software to be transferred to a different environment

ISO 9126 suggests sub-characteristics for each of the primary characteristics. They are useful as they clarify what is meant by each of the main characteristics.

Characteristic	Sub-characteristics
Functionality	Suitability
	Accuracy
	Interoperability
	Functionality compliance
	Security

'Functionality compliance' refers to the degree to which the software adheres to application-related standards or legal requirements. Typically these could be auditing requirements. Since the original 1999 draft, a sub-characteristic called 'compliance' has been added to all six ISO external characteristics. In each case, this refers to any specific standards that might apply to the particular quality attribute.

'Interoperability' is a good illustration of the efforts of ISO 9126 to clarify terminology. 'Interoperability' refers to the ability of the software to interact with other systems. The framers of ISO 9126 have chosen this word rather than 'compatibility' because the latter causes confusion with the characteristic referred to by ISO 9126 as 'replaceability' (see below).

Characteristic	Sub-characteristics
Reliability	Maturity
	Fault tolerance
	Recoverability
	Reliability compliance

'Maturity' refers to the frequency of failure due to faults in a software product, the implication being that the more the software has been used, the more faults will have been uncovered and removed. It is also interesting to note that 'recoverability' has been clearly distinguished from 'security' which describes the control of access to a system.

Characteristic	Sub-characteristics
Usability	Understandability
	Learnability
	Operability
	Attractiveness
	Usability compliance

Note how 'learnability' is distinguished from 'operability'. A software tool could be easy to learn but time-consuming to use because, say, it uses a large number of nested menus. This might be fine for a package used intermittently, but not where the system is used for many hours each day. In this case 'learnability' has been incorporated at the expense of 'operability'.

'Attractiveness' is a recent addition to the sub-characteristics of usability and is especially important when users are not compelled to use a particular software product, as in the case of games and other entertainment products.

Characteristic	Sub-characteristics
Efficiency	Time behaviour
	Resource utilization
	Efficiency compliance
Maintainability	Analysability
	Changeability
	Stability
	Testability
	Maintainability compliance

'Analysability' is the ease with which the cause of a failure can be determined. 'Changeability' is the quality that others call 'flexibility': the latter name is a better one as 'changeability' has a different connotation in plain English – it might imply that the suppliers of the software are always changing it!

'Stability', on the other hand, does not refer to software never changing: it means that there is a low risk of a modification to the software having unexpected effects.

Characteristic	Sub-characteristics
Portability	Adaptability
	Installability
	Coexistence
	Replaceability
	Portability compliance

'Portability compliance' relates to those standards that have a bearing on portability. The use of a standard programming language common to many software/hardware environments would be an example of this.

A new version of a word processing package might read the documents produced by previous versions and thus be able to replace them, but previous versions might not be able to read all documents created by the new version.

'Replaceability' refers to the factors that give 'upwards compatibility' between old software components and the new ones. 'Downwards' compatibility is not implied by the definition.

'Coexistence' refers to the ability of the software to share resources with other software components; unlike 'interoperability', no direct data passing is necessarily involved.

ISO 9126 provides guidelines for the use of the quality characteristics. Variation in the importance of different quality characteristics depending on the type of product is stressed. Once the requirements for the software product have been established, the following steps are suggested:

1. *Judge the importance of each quality characteristic for the application* Thus reliability will be of particular concern with safety-critical systems while efficiency will be important for some real-time systems.
2. *Select the external quality measurements within the ISO 9126 framework relevant to the qualities prioritized above* Thus for reliability mean time between failures would be an important measurement, while for efficiency, and more specifically 'time behaviour', response time would be an obvious measurement.
3. *Map measurements onto ratings that reflect user satisfaction* For response time, for example, the mappings might be as in Table 13.1.
4. *Identify the relevant internal measurements and the intermediate products in which they appear* This would only be important where software was being developed, rather than existing software being evaluated. For new software, the likely quality of the final product would need to be assessed during development. For example, where the external quality in question was time behaviour, at the software design stage an estimated execution time for a transaction could be produced by examining the software code and calculating the time for each instruction in a typical execution of the transaction. In our view the mappings between internal and external quality characteristics and measurements suggested in the ISO 9126 standard are the least convincing elements in the approach. The part of the standard that provides guidance at this point is a 'technical report' which is less authoritative than a full standard. It concedes that mapping external and internal measurements can be difficult and that validation to check that there is a meaningful correlation between the two in a specific environment needs to be done. This reflects a real problem in the practical world of software development of examining code structure and from that attempting to predict accurately external qualities such as reliability.

TABLE 13.1 Mapping measurements to user satisfaction

Response time (seconds)	Rating
<2	Exceeds expectation
2–5	Within the target range
6–10	Minimally acceptable
>10	Unacceptable

According to ISO 9126, measurements that might act as indicators of the final quality of the software can be taken at different stages of the development life cycle. For products at the early stages these indicators might be qualitative. They could, for example, be based on checklists where compliance with predefined criteria is assessed by expert judgement. As the product nears completion, objective, quantitative, measurements would increasingly be taken.

5. *Overall assessment of product quality* To what extent is it possible to combine ratings for different quality characteristics into a single overall rating for the software? A factor which discourages attempts at combining the assessments of different quality characteristics is that they can, in practice, be measured in very different ways, which makes comparison and combination difficult. Sometimes the presence of one quality could be to the detriment of another. For example, the efficiency characteristics of time behaviour and resource utilization could be enhanced by exploiting the particular characteristics of the operating system and hardware environments within which the software will perform. This, however, would probably be at the expense of portability.

It was noted above that quality assessment could be carried out for a number of different reasons: to assist software development, acquisition or independent assessment.

During the development of a software product, the assessment would be driven by the need to focus the minds of the developers on key quality requirements. The aim would be to identify possible weaknesses early on and there would be no need for an overall quality rating.

The problem here is to map an objective measurement onto an indicator of customer satisfaction which is subjective.

TABLE 13.2 Mapping response times onto user satisfaction

Response time (seconds)	Quality score
<2	5
2–3	4
4–5	3
6–7	2
8–9	1
>9	0

Where potential users are assessing a number of different software products in order to choose the best one, the outcome will be along the lines that product A is more satisfactory than product B or C. Here some idea of relative satisfaction exists and there is a justification in trying to model how this satisfaction might be formed. One approach recognizes some *mandatory* quality rating levels which a product must reach or be rejected, regardless of how good it is otherwise. Other characteristics might be *desirable* but not essential. For these a user satisfaction rating could be allocated in the range, say, 0–5. This could be based on having an objective measurement of some function and then relating different measurement values to different levels of user satisfaction – see Table 13.2.

Along with the rating for satisfaction, a rating in the range 1–5, say, could be assigned to reflect how important each quality characteristic was. The scores for each quality could be given due weight by multiplying it by its importance weighting. These weighted scores can then be summed to obtain an overall score for the product. The scores for various products are then put in the order of preference. For example, two products might be compared as to usability, efficiency and maintainability. The importance of each of these qualities might be rated as 3, 4 and 2, respectively, out of a possible maximum of 5. Quality tests might result in the situation shown in Table 13.3.

TABLE 13.3 Weighted quality scores

Product quality	Importance rating (a)	Product A		Product B	
		Quality score (b)	Weighted score (a × b)	Quality score (c)	Weighted score (a × c)
Usability	3	1	3	3	9
Efficiency	4	2	8	2	8
Maintainability	2	3	6	1	2
Overall			17		19

Finally, a quality assessment can be made on behalf of a user community as a whole. For example, a professional body might assess software tools that support the working practices of its members. Unlike the selection by an individual user/purchaser, this is an attempt to produce an objective assessment of the software independently of a particular user environment. It is clear that the result of such an exercise would vary considerably depending on the weightings given to each software characteristic, and different users could have different requirements. Caution would be needed here.

13.7 Product and Process Metrics

We have already discussed in Section 13.4 that the users assess the quality of a software product based on its external attributes, whereas during development, the developers assess the product's quality based on various internal attributes. We can also say that during development, the developers can ensure the quality of a software product based on a measurement of the relevant internal attributes. The internal attributes may measure either some aspects of the product (called product or of the development process (called process metrics). Let us understand the basic differences between product and process metrics.

- Product metrics help measure the characteristics of a product being developed. A few examples of product metrics and the specific product characteristics that they measure are the following: the LOC and function point metrics are used to measure size, the PM (person-month) metric is used to measure the effort required to develop a product, and the time required to develop the product is measured in months.
- Process metrics help measure how a development process is performing. Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, and the number of latent defects per line of code in the developed product.

13.8 Product versus Process Quality Management

The measurements described above relate to *products*. With a product-based approach to planning and control, as advocated by the PRINCE2 project management method, this focus on products is convenient. However, we saw that it is often easier to measure these product qualities in a completed computer application rather than during its development. Trying to use the attributes of intermediate products created at earlier stages to predict the quality of the final application is difficult. An alternative approach is to scrutinize the quality of the *processes* used to develop software product.

The system development process comprises a number of activities linked so that the output from one activity is the input to the next (Figure 13.4). Errors can enter the process at any stage. They can be caused either by defects in a process, as when software developers make mistakes in the logic of their software, or by information not passing clearly and accurately between development stages.

Errors not removed at early stages become more expensive to correct at later stages. Each development step that passes before the error is found increases the amount of rework needed. An error in the specification found in testing will mean rework at all the stages between specification and testing. Each successive step of development is also more detailed and less able to absorb change.

Note that Extreme Programming advocates suggest that the extra effort needed to amend software at later stages can be exaggerated and is, in any case, often justified as adding value to the software.

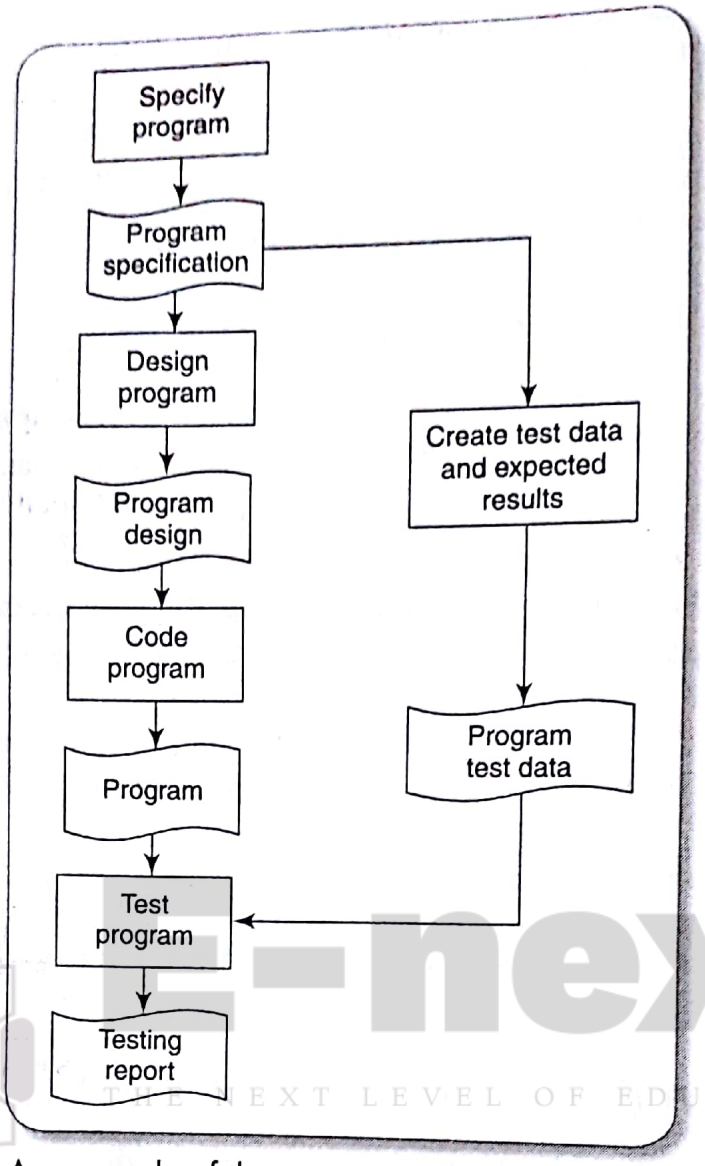

FIGURE 13.4 An example of the sequence of processes and deliverables

Errors should therefore be eradicated by careful examination of the deliverables of each step before they are passed on. One way of doing this is by having the following *process requirements* for each step.

- *Entry requirements*, which have to be in place before an activity can start. An example would be that a comprehensive set of test data and expected results be prepared and approved before program testing can commence.
- *Implementation requirements*, which define how the process is to be conducted. In the testing phase, for example, it could be laid down that whenever an error is found and corrected, all test runs must be repeated, even those that have previously been found to run correctly.
- *Exit requirements*, which have to be fulfilled before an activity is deemed to have been completed. For example, for the testing phase to be recognized as being completed, all tests will have to have been run successfully with no outstanding errors.

These requirements may be laid out in installation standards, or a *Software Quality Plan* may be drawn up for the specific project if it is a major one.

Exercise 13.4

In what cases might the entry conditions for one activity be different from the exit conditions for another activity that immediately precedes it?

Exercise 13.5

What might be the entry and exit requirements for the process *code program* shown in Figure 13.2?

13.9 Quality Management Systems

BS EN ISO 9001:2000

At IOE, a decision might be made to use an outside contractor to produce the annual maintenance contracts subsystem. A natural concern would be the standard of the contractor's deliverables. *Quality control* would involve the rigorous testing of all the software produced by the contractor, insisting on rework where defects are found. This would be very time-consuming. An alternative approach would focus on *quality assurance*. In this case IOE would check that the contractors themselves were carrying out effective quality control. A key element of this would be ensuring that the contractor had the right quality management system in place.

Various national and international standards bodies, including the British Standards Institution (BSI), have engaged in the creation of standards for quality management systems. The British Standard is now called BS EN ISO 9001:2000, which is identical to the international standard, ISO 9001:2000. Standards such as the ISO 9000 series try to ensure that a monitoring and control system to check quality is in place. They are concerned with the certification of the development process, not of the end-product as in the case of crash helmets and electrical appliances with their familiar CE marks. Standards in the ISO 9000 series relate to quality systems in general and not just those in software development.

ISO 9000 describes the fundamental features of a quality management system (QMS) and its terminology. ISO 9001 describes how a QMS can be applied to the creation of products and the provision of services. ISO 9004 applies to process improvement.

There has been some controversy over the value of these standards. Stephen Halliday, writing in *The Observer*, had misgivings that these standards are taken by many customers to imply that the final product is of a certified standard although, as Halliday says, '*It has nothing to do with the quality of the product going out of the gate. You set down your own specifications and just have to maintain them, however low they may be.*' Obtaining certification can be expensive and time-consuming which can put smaller, but still well-run, businesses at a disadvantage. Finally, there has been a concern that a preoccupation with certification might distract attention from the real problems of producing quality products. This would be another example of means–ends inversion, discussed in Chapter 4.

Putting aside these reservations, let us examine how the standard works. First, we identify those things to be the subject of quality requirements. We then put a system in place which checks that the requirements are being fulfilled and corrective action taken when necessary.

An overview of BS EN ISO 9001:2000 QMS requirements

The standard is built on a foundation of the following principles:

Remember that these standards are designed for all kinds of production – not just software development.

- Understanding the requirements of customers so that they can be met, or even exceeded
- Leadership to provide the unity of purpose and direction needed to achieve quality objectives
- Involvement of staff at all levels
- A focus on the individual processes which create intermediate or deliverable products and services
- A focus on the systems of interrelated processes that create delivered products and services
- Continuous improvement of processes
- Decision making based on factual evidence
- Building mutually beneficial relationships with suppliers

These principles are applied through cycles which involve the following activities:

1. Determining the needs and expectations of the customer
2. Establishing a *quality policy*, that is, a framework which allows the organization's objectives in relation to quality to be defined
3. Design of the *processes* which will create the products (or deliver the services) which will have the qualities implied in the organization's quality objectives
4. Allocation of the responsibilities for meeting these requirements for each element of each process
5. Ensuring that resources are available to execute these processes properly
6. Design of methods for measuring the effectiveness and efficiency of each process in contributing to the organization's quality objectives
7. Gathering of measurements
8. Identification of any discrepancies between the actual measurements and the target values
9. Analysis and elimination of the causes of discrepancies

The procedures above should be designed and executed so that there is continual improvement. They should, if carried out properly, lead to an effective QMS. More detailed ISO 9001 requirements include:

- *Documentation* of objectives, procedures (in the form of a *quality manual*), plans, and records relating to the actual operation of processes. The documentation must be subject to a change control system that ensures that it is current. Essentially one needs to be able to demonstrate to an outsider that the QMS exists and is actually adhered to.
- *Management responsibility* – the organization needs to show that the QMS and the processes that produce goods and services conforming to the quality objectives are actively and properly managed.
- *Resources* – an organization must ensure that adequate resources, including appropriately trained staff and appropriate infrastructure, are applied to the processes.
- *Production* should be characterized by:
  - Planning
  - Determination and review of customer requirements

- Effective communications between the customer and supplier
- Design and development being subject to planning, control and review
- Requirements and other information used in design being adequately and clearly recorded
- Design outcomes being verified, validated and documented in a way that provides sufficient information for those who have to use the designs
- Changes to the designs should be properly controlled
- Adequate measures to specify and evaluate the quality of purchased components
- Production of goods and the provision of services should be under controlled conditions to ensure adequate provision of information, work instruction, equipment, measurement devices, and post-delivery activities
- Measurement – to demonstrate that products conform to standards, and the QMS is effective, and to improve the effectiveness of processes that create products or services

Exercise 13.6

One of the processes involved in developing software is system testing and subsequent modification to the application in the light of errors found. If a software development organization were to attempt to conform to BS EN ISO 9001:2000, how might this affect system testing?

Exercise 13.7

Bearing in mind the criticisms of BS EN ISO 9001 that have been mentioned, what precautionary steps could a project manager take where some work for which quality is important is to be contracted out to an organization which has BS EN ISO 9001 accreditation?

13.10 Process Capability Models

As compared to the product metrics, the process metrics are more meaningfully measured during product development. Consequently, to manage quality during development, process-based techniques are very important. In this section, we discuss SEI CMM, CMMI, ISO 15504, and Six Sigma, which are popular process capability models.

A historical perspective

Before the 1950s, the primary means of realizing quality products was by undertaking extensive testing of the finished products. However, the emphasis of the quality paradigms later shifted from product assurance (extensive testing of the finished product) to process assurance (ensuring that a good quality process is used for product development). In this context, it needs to be emphasized that a basic assumption made by all modern quality paradigms is that if an organization's processes are good and are followed rigorously, then the products developed by using it would certainly be of good quality. Therefore, all modern quality assurance

techniques focus on providing sufficient guidance for recognizing, defining, analysing, and improving the process.

A good documented process enables setting up of a good quality system. However, to reach the next quality level, it is necessary to improve the process whenever any shortcomings in it are noticed. It is also necessary to incorporate into the development process any new tools or techniques that may become available. This forms the essential idea behind Total Quality Management (TQM). In a nutshell, TQM advocates that the process followed by an organization must continuously be improved through process measurements. Continuous process improvement is achieved through process redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization.

SEI capability maturity model (CMM)

The United States Department of Defence (US DoD) is one of the largest buyers of software products in the world. It has often faced difficulties dealing with the quality of performance of vendors, to whom it assigned contracts. The department had to live with recurring problems of delivery of low quality products, late delivery, and cost escalations. In this context, DoD worked with the Software Engineering Institute (SEI) of the Carnegie Mellon University to develop CMM. Originally, the objective of CMM was to assist DoD in developing an effective software acquisition method by predicting the likely contractor performance through an evaluation of their development practices.

Most of the DoD contractors began to undertake CMM-based process improvement initiatives as they vied for DoD contracts. It was soon observed that the SEI CMM model helped organizations to actually improve the quality of the software they developed. These organizations were quickly convinced that adoption of SEI CMM model had significant business benefits even when they were developing software for clients other than the DoD. Gradually many other commercial organizations began to adopt CMM in their own internal improvement initiatives.

In simple words, CMM is a reference model for appraising a software development organization into one of five process maturity levels. The maturity level of an organization is a ranking of the quality of the development process used by the organization. This information can be used to predict the most likely outcome of a project that the organization undertakes.

It should be remembered that SEI CMM can be used in two different ways, viz., capability evaluation and process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation essentially concerns assessing the software process capability of an organization. Capability evaluation is administered by the contract awarding authority, and therefore the results are indicative of the likely contractor performance if the contractor is awarded a work. On the other hand, process assessment is used by an organization with the objective of improving its own process capability. Thus, the result of the latter type of assessment is purely for internal use by a company.

In process assessment, the quality level is assessed by a team of assessors coming into an organization and interviewing the key staff about their practices, using a standard questionnaire to capture information. It needs to be remembered that in this case, a key objective is not just to assess, but to recommend specific actions to bring the organization up to a higher process maturity level. The different levels of SEI CMM have been designed so that it is easy for an organization to slowly ramp up its quality system starting from scratch. SEI CMM classifies software development organizations into the following five maturity levels.

Level 1: Initial A software development organization at this level is characterized by haphazard activities by the members of project teams. The chaotic activities are primarily brought about by the lack of any definition of the development and management processes. Each developer feels free to follow any process that he or she may like. Due to the chaotic development process practised, when a developer leaves the organization, the new incumbent usually faces great difficulty in understanding the process that was followed for the portion of the work that has been completed. Besides the lack of any agreed development processes in the organization, no systematic project management process is prevalent. Consequently, time pressure builds up towards the product delivery time. To cope up with the time pressure, many short cuts are tried out leading to low quality products. Though project failures and project completion delays are commonplace in these level 1 organizations, yet it is possible that some projects may get successfully completed. But an analysis of any incidence of successful completion of a project would reveal the heroic efforts put in by some members of the project team. Thus, it can be said that the chances of a successful project execution by a level 1 organization depends to a large extent on who exactly are the members of the development team.

Level 2: Repeatable Organizations at this level usually practise some basic project management practices such as planning and tracking cost and schedule. Further, these organizations make use of configuration management tools to keep the deliverable items under configuration control. As level 1 organizations, level 2 organizations are characterized by any documented process. However, the developers usually have a rough understanding of the process being followed in the organization. As a result, such an organization can usually repeat its success on one project on other similar projects.

Level 3: Defined At this level, the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. At this level, the organization builds up the capabilities of its employees through periodic training programs. Also, systematic reviews are practised to achieve phase containment of errors.

Level 4: Managed Organizations at this level focus on effectively managing development tasks by collecting appropriate process and product metrics. Quantitative quality goals are set for the products and processes. At the time of project completion, it is checked whether the quantitative quality goals for these have been met. The process metrics are used to check if project activities were performed satisfactorily. In other words, the collected metrics are used to measure and track project performance rather than improve the process.

Level 5: Optimizing Organizations operating at this level not only collect process and product metrics, but analyze them to identify scopes for improving and optimizing the various development and management activities. In other words, these organizations strive for *continuous process improvement*. As an example of a process optimization that may be made, consider that from an analysis of the process measurement results, it is observed that the code reviews are not very effective and a large number of errors are detected only during the unit testing. In this case, the review process would be fine-tuned to make it more effective. In a level 5 organization, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by careful analysis of the process measurement results and assimilation of innovative ideas and technologies. Level 5 organizations usually have a department whose sole responsibility is to assimilate latest tools and technologies and propagate them across the organization. Since the processes change continuously in these organizations, it becomes necessary to effectively manage these changing processes. To effectively manage process changes, level 5 organizations use configuration management techniques.

Key process areas

Except for level 1, each maturity level is characterized by several Key Process Areas (KPAs). The KPAs of a level indicate the areas that an organization at the lower maturity level needs to focus to reach this level. The KPAs for the different process maturity levels are shown in Table 13.4. Note that level 1 has no KPAs associated with it, since by default all organizations are in level 1.

KPAs provide a way for an organization to gradually improve its quality of over several stages. In other words, at each stage of process maturity, KPAs identify the key areas on which an organization needs to focus to take it to the next level of maturity. Each stage has been carefully designed such that one stage enhances the capability already built up. For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures. In other words, trying to focus on some higher level KPAs without achieving the lower level KPAs would be counterproductive.

CMMI (Capability Maturity Model Integration)

CMMI is the successor of the Capability Maturity Model (CMM). In 2002, CMMI Version 1.1 was released. Version 1.2 followed in 2006. The genesis of CMMI is the following. After CMMI was first released in 1990, it was adopted and used in many domains other than software development, such as human resource management (HRM). CMMs were developed for disciplines such as systems engineering (SE-CMM), people management (PCMM), software acquisition (SA-CMM), and others. Although many organizations found these models to be useful, they faced difficulties arising from overlap, inconsistencies, as well as integration of the models. In this context, CMMI is generalized to be applicable to many domains using a single framework. However, this unification has resulted in making CMMI much more abstract than its predecessors such as CMM. For example, all the terminologies that are used are very generic in nature and even the word software does not appear in the definition documentation of CMMI. However, CMMI has much in common with CMM, and also describes the five distinct levels of process maturity of CMM.

TABLE 13.4 CMMI key process areas

Level	Key process areas
1. Initial	Not applicable
2. Managed	Requirements management, project planning and monitoring and control, supplier agreement management, measurement and analysis, process and product quality assurance, configuration management
3. Defined	Requirements development, technical solution, product integration, verification, validation, organizational process focus and definition, training, integrated project management, risk management, integrated teaming, integrated supplier management, decision analysis and resolution, organizational environment for integration
4. Quantitatively managed	Organizational process performance, quantitative project management
5. Optimizing	Organizational innovation and deployment, causal analysis and resolution

ISO 15504 process assessment

ISO/IEC 15504 is a standard for process assessment that shares many concepts with CMMI. The two standards should be compatible. Like CMMI the standard is designed to provide guidance on the assessment of software development processes. To do this there must be some benchmark or *process reference model* which represents the ideal development life cycle against which the actual processes can be compared. Various process reference models could be used but the default is the one described in ISO 12207, which has been briefly discussed in Chapter 1 and which describes the main processes – such as requirements analysis and architectural design – in the classic software development life cycle.

The main reference in the UK for this standard is BS ISO/IEC 15504-1:2004.

Processes are assessed on the basis of nine process attributes – see Table 13.5.

TABLE 13.5 ISO 15504 framework for process capability

For a process to be judged to be at Level 3, for example, Levels 1 and 2 must also have been achieved.

Level	Attribute	Comments
0. Incomplete		The process is not implemented or is unsuccessful
1. Performed process	1.1 Process performance	The process produces its defined outcomes
2. Managed process	2.1 Performance management	The process is properly planned and monitored
	2.2 Work product management	Work products are properly defined and reviewed to ensure they meet requirements
3. Established process	3.1 Process definition	The processes to be carried out are carefully defined
	3.2 Process deployment	The processes defined above are properly executed by properly trained staff
4. Predictable process	4.1 Process measurement	Quantitatively measurable targets are set for each sub-process and data collected to monitor performance
	4.2 Process control	On the basis of the data collected by 4.1 corrective action is taken if there is unacceptable variation from the targets
5. Optimizing	5.1 Process innovation	As a result of the data collected by 4.1, opportunities for improving processes are identified
	5.2 Process optimization	The opportunities for process improvement are properly evaluated and where appropriate are effectively implemented

When assessors are judging the degree to which a process attribute is being fulfilled they allocate one of the following scores:

Level	Interpretation
N – not achieved	0 to 15% achievement
P – partially achieved	15% to 50% achievement
L – largely achieved	50% to 85% achievement
F – fully achieved	85% achievement

In order to assess the process attribute of a process as being at a certain level of achievement, indicators have to be found that provide evidence for the assessment. For example, say the requirement analysis processes of an organization were being assessed. Assessors might wish to test whether the organization is at Level 3, which relates to there being an established process. The assessor might find a section in a procedures manual relating to the conduct of requirements. This could be evidence of the process being defined (3.1 in Table 13.5). They might also come across control documents which have been signed off as each step of the requirements analysis process has been completed. This would indicate that the defined process is actually deployed (3.2).

Implementing process improvement

The CMMI standard has now grown to over 500 pages. Without getting bogged down in detail, this section explores how the general approach might usefully be employed. To do this we will take a scenario from industry.

UVW is a company that builds machine tool equipment containing sophisticated control software. This equipment also produces log files of fault and other performance data in electronic format. UVW produces software that can read these log files and produce analysis reports and execute queries.

Both the control and analysis software is produced and maintained by the Software Engineering department. Within this department there are separate teams who deal with the software for different types of equipment. Lisa is a Software Team Leader in the Software Engineering department with a team of six systems designers reporting to her.

Her group is responsible for new control systems and the maintenance of existing systems for a particular product line. The dividing line between new development and maintenance is sometimes blurred as a new control system often makes use of existing software components which are modified to create the new software.

A separate Systems Testing Group test software for new control systems, but not fault correction and adaptive maintenance of released systems.

A project for a new control system is controlled by a Project Engineer with overall responsibility for managing both the hardware and software sides of the project. The Project Engineer is not primarily a software specialist and would make heavy demands on the Software Team Leader, such as Lisa, in an advisory capacity. Lisa may, as a Software Team Leader, work for a number of different Project Engineers in respect of different projects, but in the UVW organizational chart she is shown as reporting to the Head of Software Engineering.

A new control system starts with the Project Engineer writing a software requirement document which is reviewed by a Software Team Leader, who will then agree to the document, usually after some amendment. A copy of the requirements document will pass to the Systems Testing Group so that they can create system test cases and a systems test environment.

Lisa, if she were the designated Software Team Leader, would then write an Architecture Design document mapping the requirements to actual software components. These would be allocated to Work Packages carried out by individual members of Lisa's team.

UVW teams get the software quickly written and uploaded onto the newly developed hardware platform for initial debugging. The hardware and software engineers will then invariably have to alter the requirement and consequently the software as they find inconsistencies, faults and missing functions. The Systems Testing Group should be notified of these changes, but this can be patchy. Once the system seems to be satisfactory to the developers, it is released to the Systems Testing Group for final testing before shipping to customers.

Lisa's work problems mainly relate to late deliveries of software by her group because:

(i) The Head of Software Engineering and the Project Leaders may not liaise properly, leading to the over-commitment of resources to both new systems and maintenance jobs at the same time

(ii) The initial testing of the prototype often leads to major new requirements being identified

(iii) There is no proper control over change requests – the volume of these can sometimes increase the demand for software development well beyond that originally planned

(iv) Completion of system testing can be delayed because of the number of bug fixes

We can see that there is plenty of scope for improvements. One problem is knowing where to start. However, approaches like that of CMMI can help us identify the order in which steps in improvement have to take place. Some steps need to build on the completion of others. An immediate step would be to introduce more formal planning and control. This would at least enable us to assess the size of the problems even if we are not yet able to solve them all. Given a software requirement, formal plans enable staff workloads to be distributed more carefully. The monitoring of plans would also allow managers to identify emerging problems with particular projects. Effective change control procedures would make managers more aware of how changes in the system's functionality can force project deadlines to be breached. These process developments would help an organization move from Level 1 to Level 2. Figure 13.5 illustrates how a project control system could be envisaged at the level of maturity.

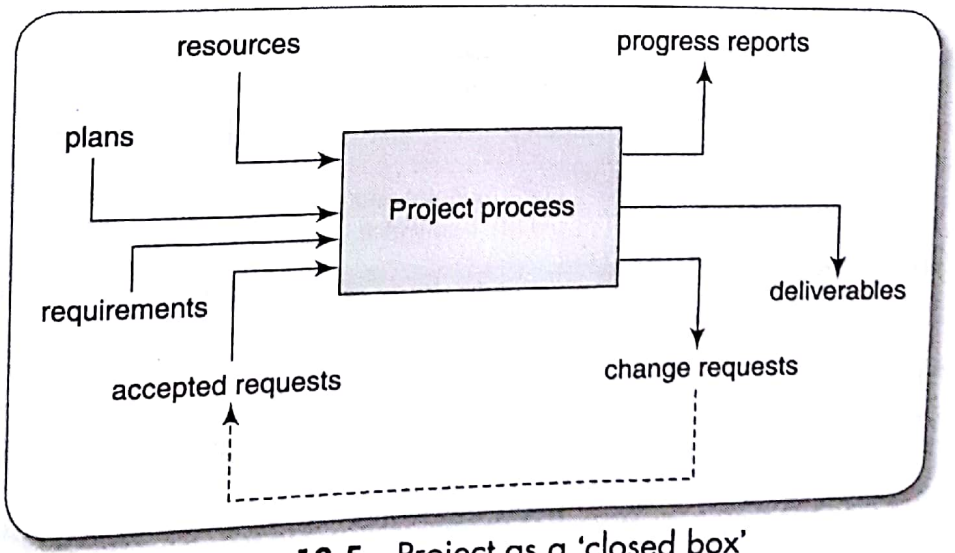

FIGURE 13.5 Project as a 'closed box'

The next step would be to define carefully the processes involved in each stage of the development life cycle – see Figure 13.6. The steps of defining procedures for each development task and ensuring that they are actually carried out help to bring an organization up to Level 3.

Exercise 13.8

The diagram in Figure 13.6 does not show the flows of information needed to indicate how managers could ensure that the correct amount of staff time is allocated to development activities. Amend the diagram to show these flows.

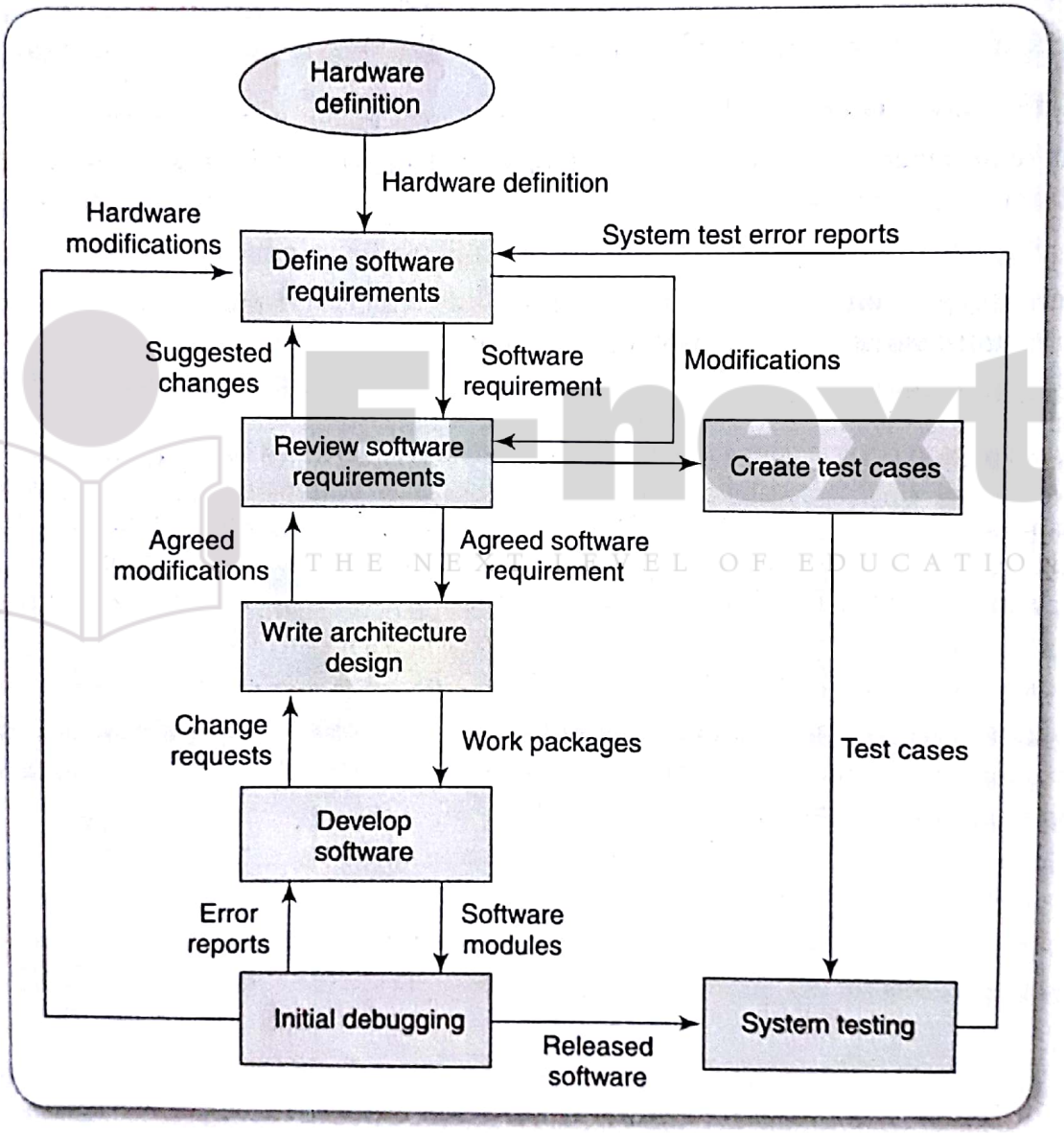

FIGURE 13.6 Process diagram

When more formalized processes exist, the behaviour of component processes can be monitored. We can see, for example, the numbers of change reports generated and system defects detected at the system testing phase. Apart from information about the products passing between processes, we can also collect effort information about each process itself. This enables effective remedial action to be taken speedily when problems are found. The development processes are now properly managed, bringing the organization up to Level 4.

Finally, at Level 5 of process management, the information collected is used to improve the process model itself. It might, for example, become apparent that the changes to software requirements are a major source of defects. Steps could therefore be taken to improve this process. For example, the hardware component of the system could be simulated using software tools. This could help the hardware engineers to produce more realistic designs and reduce changes. It might even be possible to build control software and test it against a simulated hardware system. This could enable earlier and cheaper resolution of technical problems.

Personal Software Process (PSP)

PSP is based on the work of Watts Humphrey. Unlike CMMI that is intended for companies, PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyze, design, code, test or document software products, but assumes that engineers use effective personal practices. PSP recognizes that the process for individual use is different from that necessary for a team. The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating, planning and tracking performance against plans, and provides a defined process which can be tuned by individuals.

See Watts Humphrey, 'The Personal Software Process, Technical report, SEI-CMU, November, 2000

Time measurement. PSP advocates that developers should rack the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break, etc. An engineer should measure the time he spends for various development activities such as designing, writing code, testing, etc.

PSP Planning. Individuals must plan their project. Unless an individual properly plans his activities, disproportionately high effort may be spent on trivial activities and important activities may be compromised, leading to poor quality results. The developers must estimate the maximum, minimum and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in Figure 13.7. As has been shown in Figure 13.7, an individual developer must plan the personal activities and make the basic plans before starting the development work. While carrying out the activities of different phases of software development, the individual developer must record the log data using time measurement. During post implementation project review, the developer can compare

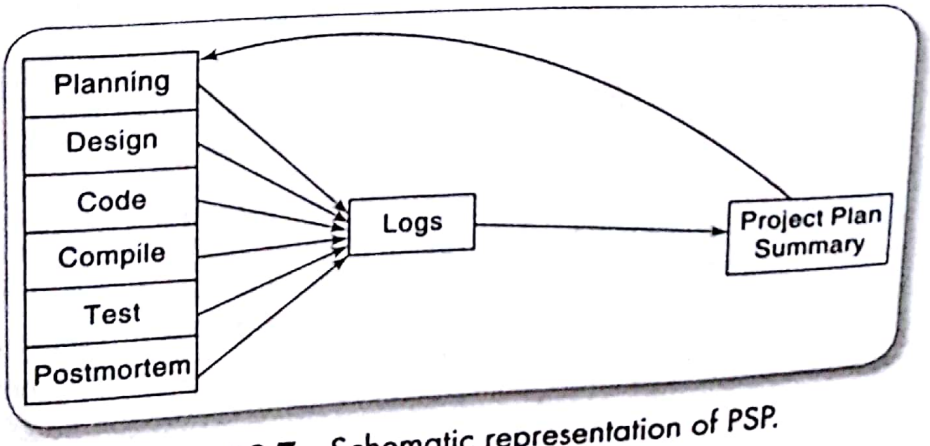

FIGURE 13.7 Schematic representation of PSP.

the log data with the initial plan to achieve better planning in the future projects, to improve his process, etc. The four maturity levels of PSP have schematically been shown in Figure 13.8. The activities that the developer must perform for achieving a higher level of maturity have also been annotated on the diagram. PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed by analysing the defect data gathered from earlier projects.

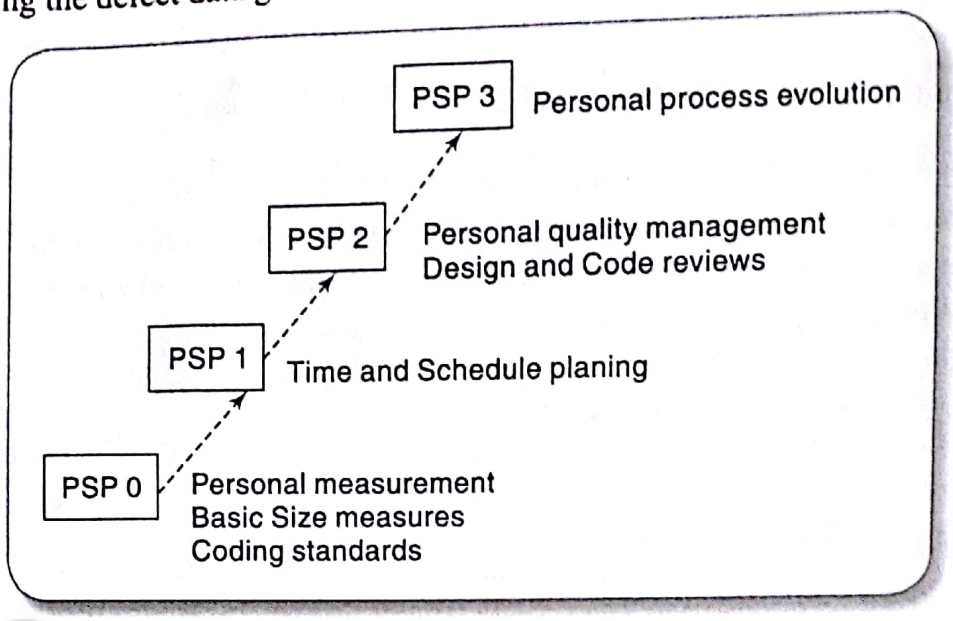

FIGURE 13.8 PSP levels

Six Sigma

Motorola, USA, initially developed the six sigma method in the early 1980s. Since then, thousands of companies around the world have discovered the benefits of adopting six sigma methodologies. The purpose of six sigma is to improve processes to do things better, faster, and at a lower cost. It can in fact, be used to improve every facet of business, i.e., production, human resources, order entry, and technical support areas. Six sigma becomes applicable to any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry. Six sigma seeks to improve the quality of process outputs by identifying and removing the causes of defects and minimizing variability in the use of process. It uses many quality management methods, including statistical methods, and requires presence of six sigma experts within the organization (black belts, green belts, etc.).

Six sigma is essentially a disciplined, data-driven approach to eliminate defects in any process. The statistical representation of six sigma describes quantitatively how a process is performing. To achieve six sigma, a process must not produce more than 3.4 defects per million defect opportunities. A six sigma defect is defined as any system behaviour that is not as per customer specifications. Total number of six sigma defect opportunities is then the total number of chances for committing an error. Sigma of a process can easily be calculated using a six sigma calculator.

As already mentioned, a basic objective of the six sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of six sigma improvement methodologies. This is accomplished through the use of two six sigma sub-methodologies: DMAIC and DMADV. The six sigma DMAIC process (define, measure, analyze, improve, control) is an improvement system for existing processes falling below specification and looking for incremental improvement. The six sigma DMADV process (define, measure, analyze, design, verify) is an improvement

system used to develop new processes or products at six sigma quality levels. Both six sigma processes are executed by six sigma green belts and six sigma black belts, and are overseen by six sigma master black belts.

Many frameworks exist for implementing the six sigma methodology. Six sigma consultants all over the world have also developed proprietary methodologies for implementing six sigma quality that is based on various philosophies and tools.

13.11 Techniques to Help Enhance Software Quality

Three main themes emerge in this discussion of software quality:

- *Increasing visibility* A landmark in the movement towards a focus on software quality was Gerald Weinberg's advocacy of 'egoless programming'. Weinberg encouraged the simple practice of programmers looking at each other's code.
- *Procedural structure* At first, programmers were left to get on with writing programs as best they could. Over the years there has been the growth of methodologies where every process in the software development cycle has carefully laid down steps.
- *Checking intermediate stages* It is tempting to push forward quickly with the development of any engineered object until a 'working' model, however imperfect, has been produced which can then be 'debugged'. The move towards quality practices has emphasized checking the correctness of work at its earlier, conceptual, stages.

Gerald Weinberg (1998) *The Psychology of Computer Programming*, Silver Anniversary Edition, Dorset House.

The creation of an early working model of a system may still be useful, as the creation of prototypes shows.

However, recently focus has shifted from relying solely on checking the products of intermediate stages and towards building an application as a number of smaller, relatively independent components developed quickly and tested at an early stage. This can reduce some of the problems, noted earlier, of attempting to predict the external quality of the software from early design documents. It does not preclude careful checking of the design of components.

We are now going to look at some specific techniques. The push towards more visibility has been dominated by the increasing use of *walk-throughs, inspections* and *reviews*. The movement towards a more procedural structure inevitably leads to discussion of structured programming techniques and to its later manifestation in the ideas of '*clean-room*' *software development*.

The interest in the dramatic improvements made by the Japanese in product quality has led to much discussion of the quality techniques they have adopted, such as the use of *quality circles*, and these will be looked at briefly. Some of these ideas are variations on the theme of inspection and clean-room development.

Inspections

Inspections can be applied to documents produced at any development stage. For instance, test cases need to be reviewed – their production is usually not a high-profile task even though errors can get through to operational running because of their poor quality.

When a piece of work is completed, copies are distributed to co-workers who examine the work, noting defects. A meeting then discusses the work and a list of defects requiring rework is produced. The work to be examined could be, typically, a program listing that is free of compilation errors.

The main problem is maintaining the commitment of participants to a thorough examination of the work distributed to them after the novelty value of reviews has worn off a little.

Our own experience of using this technique has been that:

- it is a very effective way of removing superficial errors;
- it motivates developers to produce better structured and self-explanatory software;
- it helps spread good programming practices as the participants discuss specific pieces of code;
- it can enhance team spirit.

This is sometimes called 'peer review', where 'peers' are people who are equals.

The item will usually be reviewed by colleagues who work in the same area, so that software developers, for example, will have their work reviewed by fellow developers. To reduce the problems of communication between different stages, there may be representatives from the stages preceding and following the one which produced the work under review.

IBM made the review process more structured and formal, producing statistics to show its effectiveness. A Fagan inspection (named after the IBM employee who pioneered the technique) is led, not by the author of the work, but by a specially trained 'moderator'.

The general principles behind the Fagan method

See M. E. Fagan's (1976) article 'Design and code inspections to reduce errors in program development', *IBM Systems Journal* 15(3).

- Inspections are carried out on all major deliverables.
- All types of defect are noted – not just logic or function errors.
- Inspections can be carried out by colleagues at all levels except the very top.
- Inspections are carried out using a predefined set of steps.
- Inspection meetings do not last for more than two hours.
- The inspection is led by a *moderator* who has had specific training in the technique.
- The other participants have defined roles. For example, one person will act as a *recorder* and note all defects found, and another will act as *reader* and take the other participants through the document under inspection.
- Checklists are used to assist the fault-finding process.
- Material is inspected at an optimal rate of about 100 lines an hour.
- Statistics are maintained so that the effectiveness of the inspection process can be monitored.

Exercise 13.9

Compare and contrast the peer review process described above with pair programming which is advocated as part of extreme programming (XP).

Structured programming and clean-room software development

In the late 1960s, software was seen to be getting more complex while the capacity of the human mind to hold detail remained limited. It was also realized that it was impossible to test any substantial piece of software

completely given the huge number of possible input combinations. Testing, at best, could prove the presence of errors, not their absence. Thus Dijkstra and others suggested that the only way to reassure ourselves about the correctness of software was by examining the code.

The way to deal with complex systems, it was contended, was to break them down into components of a size the human mind could comprehend. For a large system there would be a hierarchy of components and subcomponents. For this decomposition to work properly, each component would have to be self-contained, with only one entry and exit point.

E. W. Dijkstra in 1968 wrote a letter to a learned computing journal which was entitled 'Go To Statement Considered Harmful'. This unfortunately led to the common idea that structured programming was simply about not using GO TOs.

The ideas of structured programming have been further developed into the ideas of clean-room software development by people such as the late Harlan Mills of IBM.

With this type of development there are three separate teams:

- *a specification team*, which obtains the user requirements and also a *usage profile* estimating the volume of use for each feature in the system;
- *a development team*, which develops the code but which does no machine testing of the program code produced;
- *a certification team*, which carries out testing.

Usage profiles reflect the need to assess *quality in use* as discussed earlier in relation to ISO 9126. They will be further discussed in the Section 13.11 on testing below.

Any system is produced in increments – recall Section 4.10 – each of which should be capable of actual operation by the end-user. The development team does no debugging; instead, all software has to be verified by them using mathematical techniques. The argument is that software which is constructed by throwing up a crude program, which then has test data thrown at it and a series of hit-and-miss amendments made to it until it works, is bound to be unreliable.

The certification team carry out the testing, which is continued until a statistical model shows that the failure intensity has been reduced to an acceptable level.

Formal methods

Clean-room development, mentioned above, uses mathematical verification techniques. These techniques use unambiguous, mathematically based, specification languages of which Z and VDM are examples. They are used to define *preconditions* and *postconditions* for each procedure. Preconditions define the allowable states, before processing, of the data items upon which a procedure is to work. The postconditions define the state of those data items after processing. The mathematical notation should ensure that such a specification is precise and unambiguous. It should also be possible to prove mathematically (in much the same way that at school you learnt to prove Pythagoras' theorem) that a particular algorithm will work on the data defined by the preconditions in such a way as to produce the postconditions. Despite the claims of the effectiveness of the use of a formal notation to define software specifications for many years now, it is rarely used in mainstream software development. This is despite it being quite widely being taught in universities. A newer development that may meet with more success is the development of Object Constraint Language (OCL). It adds precise, unambiguous, detail to the UML models, for example about the ranges of values that would be valid for a named attribute. It uses an unambiguous, but non-mathematical, notation which developers who are familiar with Java-like programming languages should grasp relatively easily.

Software quality circles

Much interest has been shown in Japanese software quality practices. The aim of the 'Japanese' approach is to examine and modify the activities in the development process in order to reduce the number of errors that they have in their end-products. Testing and Fagan inspections can assist the removal of errors – but the same types of error could occur repeatedly in successive products created by a faculty process. By uncovering the source of errors, this repetition can be eliminated.

Staff are involved in the identification of sources of errors through the formation of *quality circles*. These can be set up in all departments of an organization, including those producing software where they are known as software quality circles (SWQC).

A quality circle is a group of four to ten volunteers working in the same area who meet for, say, an hour a week to identify, analyze and solve their work-related problems. One of their number is the group leader and there could be an outsider, a *facilitator*, who can advise on procedural matters. In order to make the quality circle work effectively, training needs to be given.

Together the quality group select a pressing problem that affects their work. They identify what they think are the causes of the problem and decide on a course of action to remove these causes. Often, because of resource or possible organizational constraints, they will have to present their ideas to management to obtain approval before implementing the process improvement.

Exercise 13.10

What are the important differences between a quality circle and a review group?

Associated with quality circles is the compilation of *most probable error lists*. For example, at IOE, Amanda might find that the annual maintenance contracts project is being delayed because of errors in the requirements specifications. The project team could be assembled and spend some time producing a list of the most common types of error that occur in requirements specifications. This is then used to identify measures which can reduce the occurrence of each type of error. They might suggest, for instance, that test cases be produced at the same time as the requirements specification and that these test cases should be dry run at an inspection. The result is a checklist for use when conducting inspections of requirement specifications.

Lessons learnt reports

Another way by which an organization can improve its performance is by reflecting on the performance of a project at its immediate end when the experience is still fresh. This reflection may identify lessons to be applied to future projects. Project managers are required to write a *Lessons Learnt* report at the end of the project. This should be distinguished from a *Post Implementation Review* (PIR). A PIR takes place after a significant period of operation of the new system, and focuses on the effectiveness of the new system, rather than the original project process. The PIR is often produced by someone who was not involved in the original project, in order to ensure neutrality. An outcome of the PIR will often be changes to enhance the effectiveness of the installed system.

The Lessons Learnt report, on the other hand, is written by the project manager as soon as possible after the completion of the project. This urgency is because the project team is often dispersed to new work soon after the finish of the project. One problem that is frequently voiced is that there is often very little follow-up on the recommendations of such reports, as there is often no body within the organization with the responsibility and authority to do so.

13.12 Testing

The final judgement of the quality of a software application is whether it actually works correctly when executed. This section looks at aspects of the planning and management of testing. A major headache with testing is estimating how much testing remains at any point. This estimate of the work still to be done depends on an unknown, the number of bugs left in the code. We will briefly discuss how we can deal with this problem.

In Chapter 4, the *V-process model* was introduced as an extension to the waterfall process model. Figure 13.9 gives a diagrammatic representation of this model. This stresses the necessity for validation activities that match the activities creating the products of the project.

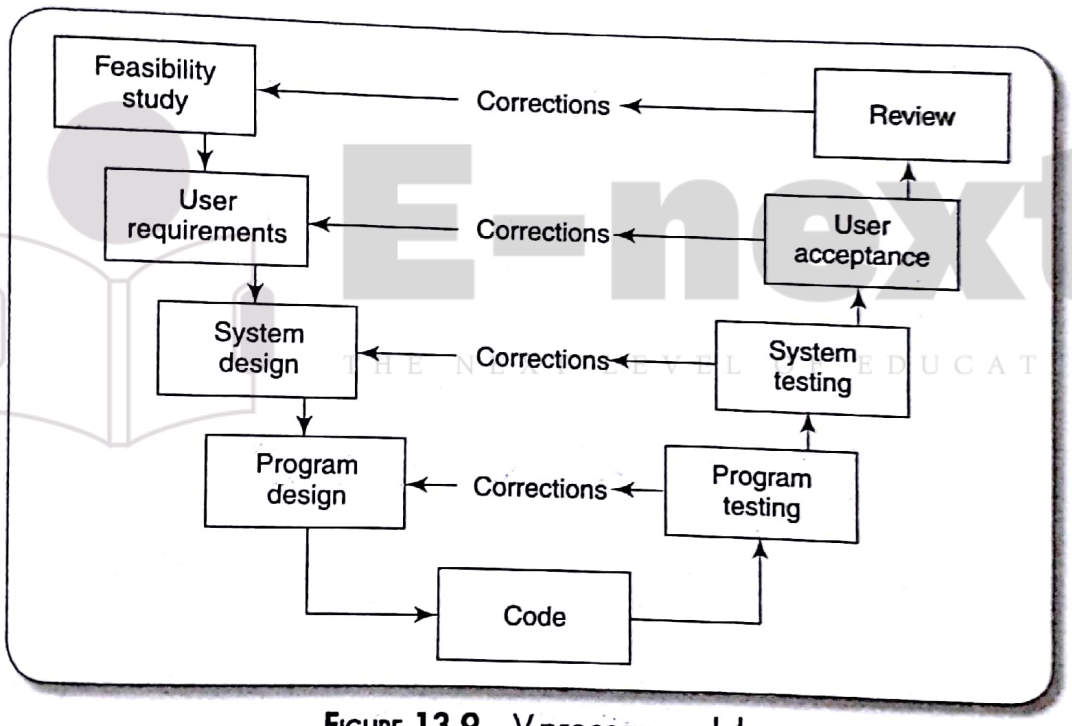

FIGURE 13.9 V-process model

The V-process model can be seen as expanding the activity box 'testing' in the waterfall model. Each step has a matching validation process which can, where defects are found, cause a loop back to the corresponding development stage and a reworking of the following steps. Ideally this feeding back should occur only where a discrepancy has been found between what was specified by a particular activity and what was actually implemented in the next lower activity on the descent of the V loop. For example, the system designer might have written that a calculation be carried out in a certain way. A developer building code to meet this design might have misunderstood what was required. At system testing stage, the original designer would be responsible for checking that the software is doing what was specified and this would discover the coder's misreading of that document.

Using the V-process model as a framework, planning decisions can be made at the outset as to the types and amounts of testing to be done. An obvious example of this would be that if the software were acquired 'off-the-shelf', the program design and code stages would not be relevant and so program testing would not be needed. User requirements would still be produced so user acceptance tests would still be valid.

Verification versus validation

The objectives of both verification and validation techniques are very similar. Both these techniques have been designed to help remove errors in software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different. The main differences between these two techniques are the following:

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase. Validation is the process of determining whether fully developed software conforms to its requirements specification. We can therefore say that the objective of verification is to check if the artifacts produced after a phase conforms to that of the previous phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements. The primary techniques used for verification include review, simulation, and formal verification. On the other hand, validation techniques are primarily based on product testing.
- Verification is carried out during the development process to check if the development activities are being carried out correctly, whereas validation is carried out towards the end of the development process to check if the right product as required by the customer has been developed. Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and accepted as an important software engineering principle.

All the boxes shown in the right hand side of the V-process model of Figure 13.9 correspond to verification activities except the system testing block which corresponds to validation activity.

Exercise 13.11

Is it at all possible to develop highly reliable software using validation techniques alone? If so, can we say that whenever thorough validation is carried out, verification is redundant?

Test case design

There are essentially two main approaches to systematically design test cases: black-box approach and white-box (or glass-box) approach.

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/output behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as *functional testing* and also as *requirements-driven testing*. Design of white-box test cases on the other hand, requires analysis of the source code. Consequently, white-box testing is also called *structural testing* or *structure-driven testing*.

Levels of testing

A software product is normally tested at three different stages or levels. These three testing stages are

- Unit testing
- Integration testing
- System testing

During unit testing, the individual components (or units) of a program are tested. For every module, unit testing is carried out as soon as the coding for it is complete. Since every module is tested separately, there is a good scope for parallel activities during unit testing. The objective of integration testing is to check whether the modules have any errors pertaining to interfacing with each other.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large. After testing all the units individually, the units are integrated over a number of steps and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing).

Exercise 13.12

Why is it necessary to test each module of a program in isolation first, then integrate these modules and re-test, and again test the integrated set of modules? Why not just test the fully integrated set of modules once thoroughly?

Testing activities

Testing involves performing the following main activities:

Test Planning Since many activities are carried out during testing, careful planning is needed. The specific test case design strategies that would be deployed are also planned. Test planning consists of determining the relevant test strategies and planning for any test bed that may be required. A suitable test bed is an especially important concern while testing embedded applications. A test bed usually includes setting up the hardware or simulator.

Test Suite Design Planned testing strategies are used to design the set of test cases (called test suite) using which a program is to be tested.

Test Case Execution and Result Checking Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for test reporting.

Test Reporting When the test cases are run, the tester may raise issues, that is, report discrepancies between the expected and the actual findings. A means of formally recording these issues and their history is needed. A review body adjudicates these issues. The outcome of this scrutiny would be one of the following:

- The issue is dismissed on the grounds that there has been a misunderstanding of a requirement by the tester.
- The issue is identified as a fault which the developers need to correct–Where development is being done by contractors, they would be expected to cover the cost of the correction.

- It is recognized that the software is behaving as specified, but the requirement originally agreed is in fact incorrect–Remedying this means adding a new requirement and a contractor could expect to receive payment for the additional work.
- The issue is identified as a fault but is treated as an *off-specification*–It is decided that the application can be made operational with the error still in place.

In a commercial project, execution of the entire test suite can take several weeks to complete. Therefore, in order to optimize the turnaround time, the test failure information is usually informally intimated to the development team as and when failures are noticed.

Debugging For each failure observed during testing, debugging is carried out to identify the statements that are in error. There are several debugging strategies, but essentially in each the failure symptoms are analyzed to locate the errors.

Error Correction After an error is located through a debugging activity; the code is appropriately changed to correct the error.

Defect Retesting Once a defect has been dealt with by the development team; the corrected code is retested by the testing team to check whether the defect has successfully been addressed. Defect retest is also popularly called *resolution testing*. The resolution tests are a subset of the complete test suite (see Figure 13.10).

Regression Testing While resolution testing checks whether the defect has been fixed, regression testing checks whether the unmodified functionalities still continue to work correctly. Thus, whenever a defect is corrected and the change is incorporated in the program code, a danger is that a change introduced to correct an error could actually introduce errors in functionalities that were previously working correctly. The regression tests check whether the unmodified functionalities continue to work correctly. As a result, after a bug-fixing session, both the resolution and regression test cases need to be run. This is where the additional effort required to create automated test scripts can pay off. As shown in Figure 13.6, some test cases may no more be valid after the change. These have been shown as invalid test case. The rest are redundant test cases, which check those parts of the program code that are not at all affected by the change.

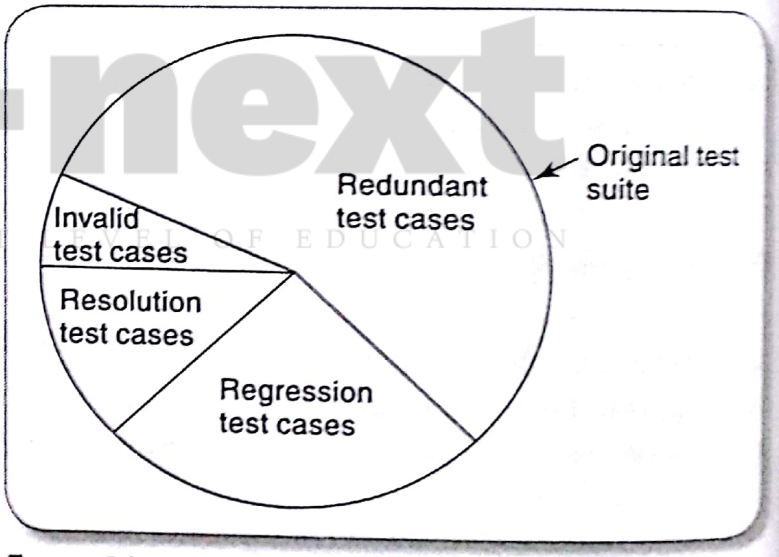

FIGURE 13.10 Types of test cases in the original test suite after a change

Test Closure Once the system successfully passes all the tests, documents related to lessons learned, test results, logs, etc., are archived for use as a reference in future projects.

Of all the above-mentioned testing activities, debugging is usually the most time-consuming activity.

Who performs testing?

A question to be settled at the planning stage is who would carry out testing. Many organizations have separate system testing groups to provide an independent assessment of the correctness of software before

release. In other organizations, staff is allocated to a purely testing role but work alongside the developers instead of a separate group. While an independent testing group can provide final quality check, it has been argued that developers may take less care of their work if they know the existence of this safety net.

Test automation

Testing is usually the most time consuming and laborious of all software development activities. This is especially true for large and complex software products that are being developed currently. In fact at present, testing cost often exceeds all other development life-cycle costs. With the growing size of programs and the increased importance being given to product quality, test automation is drawing considerable attention from both industry circles and academia. Test automation is a generic term for automating one or some activities of the test process.

Other than reducing human effort and time in this otherwise time and effort-intensive work, test automation also significantly improves the thoroughness of testing. This is because more testing can be carried out using a large number of test cases within a short period of time without any significant cost overhead.

The effectiveness of testing, to a large extent, depends on the exact test case design strategy used. Considering the large overheads that sophisticated testing techniques incur, in many industrial projects, often testing is carried out using randomly selected test values. With automation, more sophisticated test case design techniques can be deployed. Without the use of proper tools, testing large and complex software products can especially be extremely time consuming and laborious. A further advantage of using testing tools is that automated test results are much more reliable and eliminate human errors during testing. Regression testing after every change or error correction requires running several old test cases. In this situation, test automation simplifies repeated running of the test cases. Testing tools hold out the promise of substantial cost and time reduction even in the testing and maintenance phases.

Every software product undergoes significant change overtime. Each time the code changes, it needs to be tested whether the changes induce any failures in the unchanged features. Thus the originally designed test suite needs to be run repeatedly each time the code changes. Of course additional tests have to be designed and carried out on the enhanced features. Repeated running of the same set of test cases over and over after every change is monotonous, boring, and error-prone. Automated testing tools can be of considerable use in repeatedly running the same set of test cases. Testing tools can entirely or at least substantially eliminate the drudgery of running same test cases and also significantly reduce testing costs. A large number of tools are at present available both in the public domain as well as from commercial sources. It is possible to classify these tools into the following types with regard to the specific methodology on which they are based.

Capture and Playback In this type of tools, the test cases are executed manually only once. During the manual execution, the sequence and values of various inputs as well as the outputs produced are recorded. On any subsequent occasion, the test can be automatically replayed and the results are checked against the recorded output. An important advantage of the capture playback tools is that once test data are captured and the results verified, the tests can be rerun several times over easily and cheaply. Thus, these tools are very useful for regression testing. However, capture and playback tools have a few disadvantages as well. Test maintenance can be costly when the unit under test changes, since some of the captured tests may become invalid. It would require considerable effort to determine and remove the invalid test cases or modify the test input and output data. Also new test cases would have to be added for the altered code.

Automated Test Script Test scripts are used to drive an automated test tool. The scripts provide input to the unit under test and record the output. The testers employ a variety of languages to express test scripts. An

important advantage of test script-based tools is that once the test script is debugged and verified, it can be rerun a large number of times easily and cheaply. However, debugging the test script to ensure its accuracy requires significant effort. Also, every subsequent change to the unit under test entails effort to identify impacted test scripts, modify, rerun and reconfirm them.

Random Input Test In this type of an automatic testing tool, test values are randomly generated to cover the input space of the unit under test. The outputs are ignored because analysing them would be extremely expensive. The goal is usually to crash the unit under test and not to check if the produced results are correct. An advantage of random input testing tools is that it is relatively easy. This approach however can be the most cost-effective for finding some types of defects. However, random input testing is a very limited form of testing. It finds only the defects that crash the unit under test and not the majority of defects that do not crash the system but simply produce incorrect results.

Model-based Test A model is a simplified representation of program. There can be several types of models of a program. These models can either be structural models or behavioural models. Examples of behavioral models are state models and activity models. A state model-based testing generates tests that adequately cover the state space described by the model.

Estimation of latent errors

Earlier, we noted the problem of estimating the number of errors left in an application under test. At the start of testing, there is one relatively straightforward way of estimating the number of errors in code. Simply put, bigger programs are likely to have more errors. If you have collected error data from past projects, you can arrive at the historic number of errors per 1000 lines of code. This can then be used to arrive at a reasonable estimate of the number of errors likely to be found in a new system development of a known size.

This estimate could be confirmed during the actual testing. One suggestion is that known errors can be seeded in the software. This seeding could be done by having one or more people doing a desk-check of code, but then leaving any errors found in the code. Say 10 such errors are found. Then suppose that after the first set of tests 30 errors were found of which six were known errors, that is 60% of the seeded errors. This suggests that around 40% of the errors have still to be detected, that is 20 errors (of which four are already known). The method of calculating an estimate of the errors in the software is

(total errors found)/(seeded errors found) × *(total number of seeded errors)*

Tom Gilb (1977) *Software Metrics* Winthrop Publishers, Cambridge, MA.

You may be thinking that deliberately putting (or leaving) known errors in software is a bit sneaky. It might be more acceptable to use a slightly different approach originally suggested by Tom Gilb. Two different reviewers, or groups of reviewers, are asked to inspect or test the same code. They must be completely independent of one another. Three counts are collected:

- $n1$, the number of valid errors found by A
- $n2$, the number of valid errors found by B
- $n12$, the number of cases where the same error is found by both A and B.

The smaller the proportion of errors found by both A and B compared to those found by only one reviewer, the larger the total number of errors likely to be in the software. An estimate of the total number of errors (n) can be calculated by the formula:

$$n = (n1 \times n2)/n12$$

For example, A finds 30 errors and B finds 20 errors of which 15 are common to both A and B. The estimated total number of errors would be:

$$(30 \times 20)/15 = 40$$

13.13 Software Reliability

We have pointed out in Section 13.6 that reliability is an important quality attribute. In this section, we discuss some basic concepts in software reliability engineering. The reliability of a software product essentially denotes its *trustworthiness* or *dependability*. Alternatively, the reliability of a software product can be defined as the probability of its working *correctly* over a given period of time.

Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system would improve if the number of defects in it is reduced. However, it is very difficult to formulate a mathematical expression to characterize the reliability of a system in terms of the number of latent defects in it. To get an insight into this issue, consider the following. Removing errors from those parts of a software product that are infrequently executed makes little difference to the reliability of the product. It has been experimentally observed by analysing the behaviour of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. Therefore, in addition to the number of defects, the specific point in the program (core or non-core part) where the bug is located also matters. Further, reliability is observer dependent, in the sense that it depends on the relative frequency with which different users invoke the functionalities of a system. It is possible that because of different usage patterns of the available functionalities of software, a bug which frequently shows up for one user, may not show up at all for another user, or may show up very infrequently.

Reliability of a software product usually keeps on improving with time during the testing and operational phases as defects are identified and repaired. In this context, the growth of reliability over the testing and operational phases can be modelled using a mathematical expression called Reliability Growth Model (RGM). Thus, RGM models show how the reliability of a software product improves as failures are reported and bugs are corrected. A large number of RGMs have been proposed by researchers based on various failure and bug repair patterns. A few popular reliability growth models are Jelinski–Moranda model, Littlewood–Verall's model, and Goel–Okutomo's model. For a given development project, a suitable RGM can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when during the testing phase a given reliability level will be attained, so that testing can be stopped.

Based on the above discussions, we can summarize the main reasons that make software reliability more difficult to measure than hardware reliability:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

Hardware *versus* Software Reliability

A fundamental issue that sets the reliability study of software apart from hardware reliability study is the difference between their failure patterns. Hardware components fail due to very different reasons as compared

to failure of software components. Hardware components fail mostly due to wear and tear, whereas software components fail due to presence of bugs.

As an example of a hardware, consider an electronic circuit. In this circuit, a failure may occur as a logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug. For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (e.g. the inter-failure times remain constant). On the other hand, the aim of software reliability study would be reliability growth (i.e. increase in inter-failure times).

A comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product is sketched in Figure 13.11. Observe that the plot of change of reliability with time for a hardware component [Figure 13.11(a)] appears like a 'bath tub'. As shown in Figure 13.11(a), for a hardware system the failure rate is initially high, but decreases as the faulty components are identified and are either repaired or replaced.

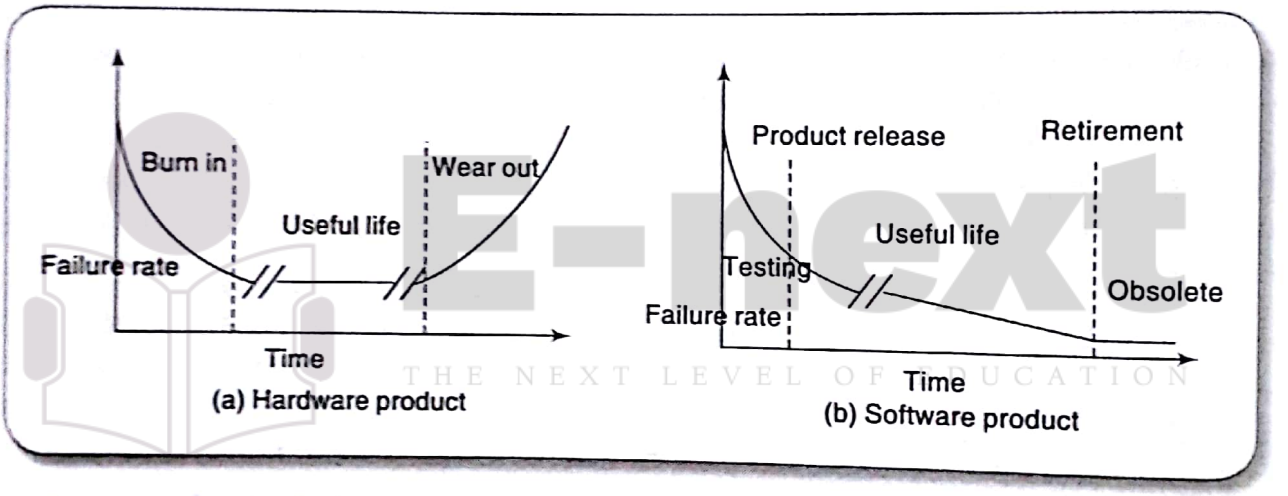

Figure 13.11 Reliability growth with time for hardware and software products

A hardware system enters its useful life, where the rate of failure is almost constant. After some time (called product life time) the major components wear out, and the failure rate increases. The initial failures are usually covered through manufacturer's warranty. A corollary of this observation (though a digression from our topic of discussion) is that it may be unwise to buy a product (even at a good discount to its face value) towards the end of its life time, That is, one need not feel happy to buy a 10-year-old car at one-tenth of the price of a new car, since it would be near the rising edge of the bath tub curve, and one would have to spend unduly large time, effort and money on repairing and end up as loser.

In contrast to the hardware products, software products show the highest failure rate just after purchase and installation [see the initial portion of the plot in Figure 13.11(b)]. As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no more error correction occurs and the failure rate remains unchanged.

Reliability Metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product.

A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very difficult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that correlate with reliability as follows.

Rate of occurrence of failure (ROCOF)

ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.

However, many software products do not run continuously (unlike a car or a mixer), but deliver certain service when a demand is placed on them. For example, a library software is idle until a book issue request is made. Therefore, for a typical software product such as a pay-roll software, applicability of ROCOF is limited.

Mean Time to Failure (MTTF)

MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2, ... and t_n. Then, MTTF can be calculated as $\sum t_i/n$. It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc., are not taken into account in the time measurements and the clock is stopped at these times.

Mean Time to Repair (MTTR)

Once failure occurs, sometime is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean Time between Failure (MTBF)

The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF.

Probability of Failure on Demand (POFOD)

Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure. We have already mentioned that the reliability of a software product should be determined through specific service invocations, rather

than making the software run continuously. Thus, POFOD metric is very appropriate for software products that are not required to run continuously.

Availability. Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, operating systems and embedded controllers, etc. which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked.

Failures which are transient and whose consequences are not serious are in practice of little concern in the operational use of a software product. These types of failures can at best be minor irritants. On the other hand, more severe types of failures may render the system totally unusable. In order to estimate the reliability of a software product more accurately, it is necessary to classify various types of failures. In the following, we give a simple classification of software failures into different types.

Transient. Transient failures occur only for certain input values while invoking a function of the system.

Permanent. Permanent failures occur for all input values while invoking a function of the system.

Recoverable. When a recoverable failure occurs, the system can recover without having to shut down and restart the system (with or without operator intervention).

Unrecoverable. In unrecoverable failures, the system may need to be restarted.

Cosmetic. These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

Reliability Growth Modelling

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired.

A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.

Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Therefore, perfect error fixing is implicit in this model. Another implicit assumption in this model is that all errors contribute equally to reliability growth (reflected in equal step size). Both the assumptions are unrealistic since different errors contribute differently to reliability growth and also the error fixed may not be perfect. Typical reliability growth predicted using this model has been shown in

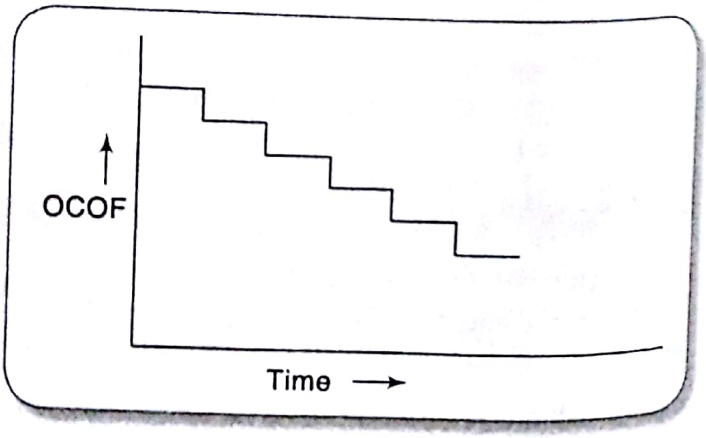

FIGURE 13.12 Jelinski–Moranda model

Figure 13.12. The instantaneous failure rate (or the hazard rate) in this model is given by $Z(t) = K(N - i)$; where K is a constant, N is the total number of errors in the program, and t is any time between the ith and $(i + 1)$th failure.

Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as the test continues.

Goel–Okutomo Model

In this model, it is assumed that the execution times between the failures are exponentially distributed. The cumulative number of failures at any time can therefore be given in terms of $\mu(t)$, the expected value of failures between time t and time $t + \Delta t$. It is assumed that it follows a non-homogeneous Poisson process (NHPP). That is, the expected number of error occurrences for any time t to $t+\Delta t$ is proportional to the expected number of undetected errors at time t. Once a failure has been detected, it is assumed that the error correction is perfect and immediate. The number of failures over time is given in Figure 13.13. The number of failures at time t can be given by, $\mu(t) = N(1 - e^{-bt})$, where N = Expected total number of defects in the code and b is the rate at which the failure rate decreases.

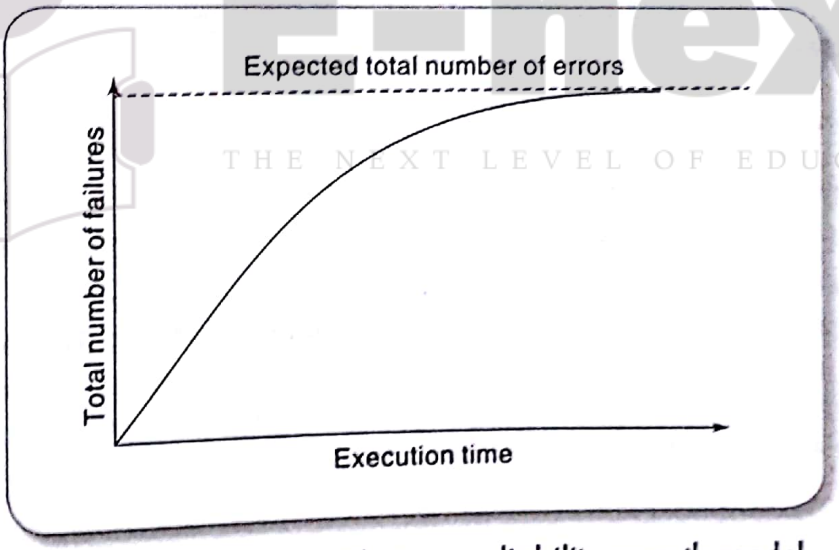

FIGURE 13.13 Goel–Okutomo reliability growth model

13.14 Quality Plans

Some organizations produce *quality plans* for each project. These show how the standard quality procedures and standards laid down in an organization's *quality manual* will actually be applied to the project. If an approach to planning such as Step Wise has been followed, quality-related activities and requirements will have been identified by the main planning process with no need for a separate quality plan. However, where software is being produced for an external client, the client's quality assurance staff might require that a

quality plan be produced to ensure the quality of the delivered products. A quality plan can be seen as a checklist that all quality issues have been dealt with by the planning process. Thus, most of the content will be references to other documents.

A quality plan might have entries for:

This contents list is based on a draft IEEE standard for software quality assurance plans.

- Purpose – scope of plan
- List of references to other documents
- Management arrangements, including organization, tasks and responsibilities
- Documentation to be produced
- Standards, practices and conventions
- Reviews and audits
- Testing
- Problem reporting and corrective action
- Tools, techniques and methodologies
- Code, media and supplier control
- Records collection, maintenance and retention
- Training
- Risk management – the methods of risk management that are to be used

Conclusion

Important points to remember about software quality include the following.

- Quality by itself is a vague concept and practical quality requirements have to be carefully defined.
- There have to be practical ways of testing for the relative presence or absence of quality.
- Most of the qualities that are apparent to the users of software can only be tested for when the system is completed.
- Therefore ways are needed of checking during development what the quality of the final system is likely to be.
- Some quality-enhancing techniques concentrate on testing the products of the development process, while others try to evaluate the quality of the development processes used.

Further Exercises

1. An organization is contemplating the purchase of a project planning software tool, such as MS Project, and has decided to draw up quality specifications for the package. The features that they are particularly concerned with are:
   - Setting up details of new projects
   - Allocating resources to project tasks, taking account of the need for resource smoothing
   - Updating the project details with information about actual tasks completed

- Effective presentation of plans

Draw up quality specifications in respect of the qualities of:

- Usability
- Reliability
- Recoverability

2. The following is an excerpt from a report generated from a help-desk logging system.

Module	Date fault reported	Fault corrected	Effort (hours)
AA247	1.4.2004	2.4.2004	5
AA247	10.4.2004	5.5.2004	4
AA247	12.4.2004	5.5.2004	3
AA247	6.5.2004	7.5.2004	2

Assess the maintainability of module AA247 from the point of view of:

- User management
- Developer management

3. Discuss how meaningful the following measurements are.
   (a) The number of error messages produced on the first compilation of a program.
   (b) The average effort to implement changes requested by users to a system.
   (c) The percentage of lines in program listings that are comments.
   (d) The number of pages in a requirements document.
4. How might you measure the effectiveness of a user manual for a software package? Consider both the measurements that might be applicable and the procedures by which the measurements might be taken.
5. What might the entry, implementation and exit requirements be for the process *design program structure*?
6. Identify a task that you do as part of your everyday work. For that task identify entry, process and exit requirements.
7. What BS EN ISO 9001 requirements have a bearing on the need for an effective configuration management system?
8. In a software development organization, identify the persons responsible for carrying out the quality assurance activities. Explain the principal tasks they perform to meet this responsibility.
9. Suppose an organization mentions in its job advertisement that it has been assessed at level 3 of SEI CMM. What can you infer about the current quality practices at the organization? What does this organization have to do to reach SEI CMM level 4?
10. Suppose as the president of a company, you have the choice to either go for ISO 9000 based quality model or SEI CMM based model, which one would you prefer? Give the reasons for your choice.
11. In a software development organization whose responsibility is it to ensure that the products are of high quality? Explain the principal tasks they perform to meet this responsibility.

12. What do you understand by repeatable software development? Organizations assessed at which level of SEI CMM maturity achieve repeatable software development?
13. What do you understand by Key Process Area (KPA), in the context of SEI CMM? Would there be any problem if an organization tries to implement higher level SEI CMM KPAs before achieving lower level KPAs? Justify your answer using a suitable example.
14. What do you understand by the six sigma quality initiative? To which category of industries is it applicable? Explain the six sigma technique with respect to its goal, the procedure followed, and the outcome expected.
15. What is the difference between process and product metrics? Give two examples of each. How does computation of process and product metrics help in developing quality products?
16. Identify the persons responsible for carrying out testing in a software development organization. Explain the principal tasks they perform.
17. Identify the factors which make the measurement of software reliability a much harder problem than the measurement of hardware reliability.
18. Through a simple plot, explain how the reliability of a software product changes over its lifetime.
19. Draw the reliability change for a hardware product over its life time and explain why the two plots look so different?
20. What do you understand by a reliability growth model? How is reliability growth modelling useful?
21. Suppose an organization mentions in its job advertisement that it has been assessed at level 3 of SEI CMM, what can you infer about the current quality practices at the organization? What does this organization have to do to reach SEI CMM level 4?
22. Suppose as the president of a company, you have the choice to either go for ISO 9000-based quality model or SEI CMM-based model, which one would you prefer? Give the reasoning behind your choice.
23. For each of the following questions, exactly one option is correct. Select the appropriate option.
    (i) Which of the following is a practical use of reliability growth modelling?
        (a) Determine the operational life of an application software.
        (b) Determine when to stop testing.
        (c) Incorporate reliability information while designing.
        (d) Incorporate reliability growth information in the code.
    (ii) Which one of the following can be considered to be a software defect?
        (a) A mistake made by a team member during any development activity
        (b) A change that the team needs to make based on the finding in code review
        (c) A configuration management plan that is too elaborate
        (d) A change request that has been launched by the customer
    (iii) Which one of the following is true of formal reviews?
        (a) Identification of faults without suggesting any solutions
        (b) Identification of faults and fixing responsibility without suggesting any solutions
        (c) Identification of faults, fixing responsibility and suggesting possible solution
        (d) Identification of faults and suggest possible solutions

(iv) Which one of the following is the basic focus of the modern quality assurance paradigms?

(a) Process assurance

(b) Product assurance

(c) Thorough testing

(d) Thorough testing and rejection of bad products

(v) Which one of the following ISO quality assurance standard applies to software development organizations?

(a) ISO 9000

(b) ISO 9001

(c) ISO 9002

(d) ISO 9003

(vi) A software organization has been assessed at SEI CMM Level 4. Which one of the following does the organization need to set up, as one of the KPAs to achieve Level 5?

(a) Defect detection

(b) Defect prevention

(c) Defect isolation

(d) Defect seeding

(vii) Which one of the following can be considered as the focus of 'quality control'?

(a) All development work products are delivered on time and under budget

(b) The development process is completely documented and adhered to during development

(c) The performance of the development work products are measured

(d) All development work products are thoroughly tested before delivery to the customer

(viii) After a software is maintained, which one of the following types of testing involves repeating execution of tests that were already performed to ensure that the code modifications that were done during maintenance have not introduced faults into previously operational software?

(a) Beta testing

(b) Functional testing

(c) Bottom-up testing

(d) Regression testing

(ix) Which of the following is indicated by the SEI CMM repeatable software development:

(a) Success in development of a software can be repeated.

(b) Success in development of a software can be repeated in related software development projects.

(c) Success in development of a software can be repeated in all the software development projects that the organization might undertake.

(d) When the same development team is chosen to develop another software, they can repeat their success.

(x) Which one of the following is a requirement for ISO 9001 certification?

(a) Demonstrate achievement of quality requirements by the developed software.

(b) Submit the reliability assessment of the developed software.

(c) Document the process being followed.

(d) Demonstrate that through stringent design and coding standards, testing is made redundant.

(xi) In CMMI, which one of the following is the name of the maturity level 5?

(a) Six Sigma
(b) Optimized
(c) Statistical
(d) Optimizing

(xii) Which one of the following statements is not an objective of software verification?

(a) Ensuring that the software development steps are carried out correctly
(b) Ensuring that the correct software has been developed
(c) Achieving phase containment of errors
(d) Ensuring that the outputs produced at a development stage conform to the outputs of the previous phase

(xiii) During which one of the following phases of the V model of software development are the integration test cases designed?

(a) Requirements specification
(b) High-level design
(c) Detailed design
(d) Coding

(xiv) Which one of the following can be said about the reliability of a typical software during its life time?

(a) Initially increases, but later decreases
(b) Initially decreases, but later increases
(c) Increasing trend throughout the life cycle
(d) Decreasing trend throughout the life cycle

(xv) The purpose of error seeding is which one of the following?

(a) Determine the origin of the bugs.
(b) Plant Trojans.
(c) Determine the number of latent bugs.
(d) Plant bugs in the code to check the efficacy of maintenance.

(xvi) Suppose in order to estimate the number of latent errors in a program, you seed it with 100 errors of different kinds. After testing the software using its full test suite, you discover only 80 of the introduced errors. You discover 16 other errors also. Estimate the number of latent errors in the software.

(a) 4
(b) 8
(c) 16
(d) 20

(xvii) Which one of the following statements is not true about the typical usage of a quality model?

(a) It forms the basis for specifying the quality requirements for a software product.
(b) It forms the basis for quantitatively measuring the quality of a software product.
(c) It helps to improve the quality of a software product by improving the attributes constituting the quality model.
(d) The quality assurance activities for a software product are defined based on the characteristics of software products that constitute the quality model.

(xviii) Which one of the following is not a valid justification for using automated tools for carrying out regression testing of a software product?

(a) Regression test cases are executed frequently

(b) Regression test cases are essentially test cases that have already been executed once

(c) At present highly effective capture and replay type of automated testing tools are available both from open source repositories as well as from commercial offerings; however, effective test case generation tools are yet to become available

(d) Some of the original test cases may require maintenance after a code change

(xix) Which one of the following is not a factor contributing to the increased difficulty of software reliability measurement, as compared to hardware reliability measurement?

(a) The reliability improvement due to fixing a single bug depends on where the bug is located in the code

(b) The perceived reliability of a software product is observer-dependent

(c) The reliability of a software product keeps changing as errors are detected and fixed

(d) Software is invisible in contrast to hardware, though the effect of execution of software can be observed