UNIT 2 QUALITY CONCEPTS

Structure **Page Nos.**

2.0 INTRODUCTION

The goal of any engineering activity is to build a product. For example, the aerospace engineer builds an aeroplane. The product of the software engineer is a software system. It is possible to modify software. This quality makes software quite different from other products such as cars. In this unit, we will first examine the important qualities of software and then discuss software engineering principles.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- learn different qualities of a software product, and
- know the principles of software engineering.

2.2 IMPORTANT QUALITIES OF SOFTWARE PRODUCT AND PROCESS

There are many important qualities of software products. Some of these qualities are applicable both to product and to the process used to produce the product. The user wants the software product to be reliable and user-friendly. The designer of the software wanted it to be maintainable, portable and extensible. In this unit, we will consider all these qualities.

2.2.1 Correctness

A program is functionally correct if it behaves according to the specification of the functions it should provide (called functional requirements specifications). It is common to use the term correct rather *functionally correct*; similarly, in this context, the term specification implies *functional requirements specifications.* We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is a desirable property for software systems.

Correctness is a mathematical property that establishes the equivalence between the software and its specification. Obviously, we can be more systematic and precise in assessing correctness depending on how rigorous we are in specifying functional requirements. Correctness can be assessed through a variety of methods, some stressing an experimental approach (e.g., testing), others stressing an analytic approach (e.g., formal verification of correctness). Correctness can also be enhanced by using appropriate tools such as high-level languages. Likewise, it can be improved by using standard algorithms or by using libraries of standard modules, rather than inventing new ones.

2.2.2 Reliability

Software Reliability may be defined as the ability of it to perform its required functions under stated conditions for a specified period of time.

Informally, software is reliable if the user can depend on it. The specialised literature on software reliability defines reliability in terms of statistical behaviour – the probability that the software will operate as expected over a specified time.

Correctness is an absolute quality; any deviation from the requirements makes the systems incorrect, regardless of how minor or serious are the consequences of the deviation. The notion of reliability is, on the other hand, relative; if the consequence of a software error is not serious, the incorrect software may still be reliable.

2.2.3 Robustness

Software robustness is the degree to which the software can function correctly in the presence of invalid inputs or stressful environmental conditions.

Informally, a program is robust if it behaves reasonably even in circumstances that were not anticipated in the requirements specification – for example, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its reasonable behaviour completely. Thus, robustness would become equivalent to correctness.

2.2.4 User Friendliness, Verifiability and Maintainability

A software is said to be user friendly if it was designed with that primary objective and the same was met.

The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase is known as Software Verification and if it is possible to do the same, then we say that the software is verifiable.

The ease with which a software system can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment, is known as Software Maintainability.

2.2.5 Reusability

Reusability may be defined as the degree to which a software module can be used in more than one computer program or a software system.

Reusability is akin to evolvability. In product evolution, we modify a product to build a new version of that same product. In product reuse, we use it – perhaps with minor changes – to build another product. Reusability appears to be more applicable to software components than to whole products but it certainly seems possible to build products that are reusable.

A good example of a reusable product is the UNIX shell. The UNIX shell is a command language interpreter; that is, it accepts user commands and then executes. But it is designed to be used both interactively and in batch. The ability to start a new shell with a file containing a list of shell commands allows us to write programs – scripts – in the shell command language. We can view the program as a new product that uses the shell as a component. By encouraging standard interfaces, the UNIX environment in fact supports the reuse of any of its commands, as well as the shell, in building powerful utilities.

Scientific libraries are best known reusable components. Several large FORTRAN libraries have existed for many years. Users can buy these and use them to build their own product, without having to reinvent or recode well-known algorithms. Indeed, several companies are devoted to producing just such libraries.

Another successful example of reusable packages is the development of systems based on windows such as X-windows or Motif, for the development of user interfaces. Unfortunately, while reusability is clearly an important tool for reducing software production costs, examples of software reuse in practice are becoming more and more.

Reusability is difficult to achieve a posteriori, therefore, one should strive for reusability when software components are developed. One of the more promising techniques is the use of object-oriented design, which can unify the qualities of evolvability and reusability. So far, we have discussed reusability in the framework of reusable components, but the concept has broader applicability: It may occur at different levels and may affect both product and process.

Another level of reuse may occur at the requirements level. When a new application is conceived, we may try to identify parts that are similar to parts used in a previous application. Thus, we may reuse parts of the previous requirements specification instead of developing an entirely new one.

As discussed above, further levels of reuse may occur when the application is designed, or at the code level. In the latter case, we might be provided with software components that are reused from a previous application. Some software experts claim that in the future, new applications will be produced by assembling together a set of readymade, off-the-shell components. Software companies will invest in the development of their own catalogues of reusable components so that the knowledge acquired in developing applications will not disappear as people leave, but will progressively accumulate in the catalogues. Other companies will invest their efforts

in the production of generalised reusable components to be put in the marketplace for use by other software producers.

Reusability applies to the software process as well. Indeed, the various software methodologies can be viewed as attempts to reuse the same process for building different products. The various life cycle models also attempt at reusing higher level processes. Another example of reusability in a process is the replay approach to software maintenance. In this approach, the entire process is repeated when making a modification. That is, at first the requirements are modified, and then the subsequent steps are followed as in the initial product development.

Reusability is a key factor that characterizes the maturity of an industrial field. We see a high degree of reusability in such mature areas as the automobile industry and consumer electronics. For example, in the automobile industry, the design of engine is often reused from model to model. Moreover, a car is constructed by assembling together many components that are highly standardised and used across many models produced by the same industry. Finally, the manufacturing process is often reused.

2.2.6 Portability

The ease with which a software can be transferred from one hardware or software environment to another is known as Portability.

Software is portable if it can run in different environments. The term environment can refer to a hardware platform or a software environment such as a particular operating system. With the proliferation of different processors and operating systems, portability has become an important issue for software engineers.

More generally, portability refers to the ability to run a system on different hardware platforms. As the ratio of money spent on software versus hardware increases, portability gains more importance. Some software systems are inherently machine specific. For example, an operating system is written to control a specific computer, and a compiler produces code for a specific machine. Even in these cases, however, it is possible to achieve some level of portability. Again, UNIX is an example of an operating system that has been ported to many different hardware systems. Of course, the porting effort requires months of work. Still, we can call software as portable because writing the system from scratch for the new environment would require much more effort than porting it.

For many applications, it is important to be portable across operating systems. Or, looked at another way, the operating system provides portability across hardware platforms.

2.2.7 Data Abstraction

Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the unimportant details.

The programming languages that we use are abstractions built on top of the hardware. They provide us with useful and powerful constructs so that we can write (most) programs ignoring such details as the number of bits that are used to represent numbers or the addressing mechanism. This helps us concentrate on the problem to solve rather than the way to instruct the machine on how to solve it. The programs we write are themselves abstractions. For example, a computerised payroll procedure is an abstraction over the manual procedure it replaces. It provides the essence of the manual procedure, not its exact details.

Data abstraction is a concept which encapsulates (collects) data structure and well defined procedure/function in a single unit. This encapsulation forms a wall which is

intended to shield the data representation from computer used. There are two requirements for data abstraction facilities in programming language.

(i) Data structure and operations as operations as described is a single semantic unit.

(ii) Data structure and internal representation of the data abstractions are not visible to the programmer. Rather, the programmer is presented with a well defined procedural interface. Today, most of the object oriented programming languages support this feature.

2.2.8 Modularity

Modularity may be defined as the degree to which a computer program is composed of discrete components such that a change to one component has minimal impact on other components.

A complex system may be divided into similar pieces called modules. A system that is composed of modules is called modular. The main benefit of modularity is that is allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring details of other modules); and when dealing with the overall characteristics of all modules and their relationship in order to integrate them into a coherent system. If the two phases are temporarily executed in the order mentioned, then we say that the system is designed bottom up; the converse denotes top-down design.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in a simple way (sequentially or overlapping) to achieve the desired result.

We will emphasise modularity in the context of software design in the next chapter. Modularity, however, is not only a desirable design principle, but permeates the whole of software production. In particular, there are three goals that modularity tries to achieve in practice: capability of decomposing a complex system, composing it from existing modules, and of understanding the system in pieces.

The decomposability of a system is based on dividing the original problem top down into sub-problems and then applying the decomposition to each sub-problem. Recursively. This is known as “Divide and Conquer” technique.

The composability of a system is based on starting bottom up from elementary components and proceeding to the finished system. As an example, a system for office automation may be designed by assembling together existing hardware components such as personal workstations, a network, and peripherals; system software such as the operating system; and productivity tools such as document processors, data bases and spreadsheets. A car is another obvious example of a system that is built by assembling components. First, consider the main subsystems into which a car may be decomposed, namely, the body, the electrical system, the power system, the transmission system, etc. Each of them, in turn, is made out of standard parts; for example, the battery, fuses, cables, etc. form the electrical system. When something goes wrong, defective components may be replaced by new ones.

Ideally, in software production we would like to be able to assemble new applications by taking modules from a library and combining them to form the required product. Such modules should be designed with the express goal of being reusable. By using reusable components, we may speed up both the initial system construction and its fine-tuning. For example, it would be possible to replace a component by another that performs the same function but differs in computational resource requirements.

The capability of understanding each part of a system separately aids in modifying a system. The evolutionary nature of software is such that the software engineer is often required to go back to a previous step to modify it. If the system cannot be understood in its entirety, then, modifications are likely to be difficult to apply, and the results are unreliable. When the need for repair arises, proper modularity helps confine the search for the source of malfunction to single components.

A module has high cohesion if all the elements of it are related strongly. Elements of a module (e.g., statement, procedures, and declarations) are grouped together in the same module for a logical reason, not just by chance; they cooperate to achieve a common goal, which is the function of the module.

Whereas cohesion is an internal property of a module, coupling characterises a module's relationship to other modules. Coupling measures the interdependence of two modules (e.g., module A calls a routine provided by module B or accesses a variable declared by Module B). If two modules depend on each other heavily, they have high coupling. Ideally, we would like modules in a system to exhibit low coupling, because if two modules are highly coupled, it will be difficult to analyse, understand, modify, test, or reuse them separately.

Module structures with high cohesion and low coupling allow us to see modules as black boxes when the overall structure of a system is described and then deal with each module separately when the module's functionality is described or analysed. This is just another example of the principle of separation of concerns.

☞ **Check Your Progress 1**

1) _____ may be defined as the degree to which a software module can be used in more than one computer program or a software system.

2) _____ may be defined as the degree to which a computer program is composed of discrete components such that a change to one component has minimal impact on other components.

2.3 PRINCIPLES OF SOFTWARE ENGINEERING

Engineering disciplines have principles based on the laws of physics, biology, chemistry or mathematics. Principles are rules to live by; they represent the collected wisdom of many dozens of people who have learned through experience.

Because the product of software engineering is not physical, physical laws do not form a suitable foundation. Instead, software engineering has had to evolve its principles the more important ones. A customer will not tolerate a poor-quality product, regardless of how you define quality. Quality must be quantified and a mechanism put into place to motivate and reward its achievement. It may seem politically correct to deliver a product on time, even though its quality is poor, but this is correct only in the short term. There is no trade-off to be made here. The first requirement must be quality.

The degree to which a software meets specified requirements is known as Software Quality. To developers, it might be an elegant design or an elegant code. To users, it might be good response time or high capacity. For cost-conscious managers, it might be low development cost. For some customers, it might be satisfying all their perceived and not-yet-perceived needs. The dilemma is that these definitions may not be compatible.

2.3.1 High-quality Software is Possible

Although our industry is saturated with examples of software systems that perform poorly, are full of bugs, or otherwise fail to satisfy user needs, there are counter

examples. Large software systems can be built with very high quality, but they carry a steep price tag. One example is IBM's on-board flight software for the space shuttle: three million lines of code with less than one error per 10,000 lines.

Techniques that have been demonstrated to increase quality considerably include involving the customer, prototyping (to verify requirements before full-scale development), simplifying design, conducting inspections, and hiring the best people.

2.3.2 Give Products to Customers Early

No matter how hard you try to learn user's needs during the requirements phase, the most effective way to ascertain real needs is to give users a product and let them play with it. The conventional waterfall model delivers the first product after 99 per cent of the development resources have been expanded. Thus, the majority of customer feedback on need occurs after resources are expended. Contrast this with an approach that you deliver a quick-and-dirty prototype early in development, gather feedback, write a requirements specification, and then proceed with full-scale development. In this scenario, only five to twenty per cent of development resources have been expended when the customer first sees the product.

2.3.3 Determine the Problem Before Writing the Requirements

When faced with what they believe is a problem, most engineers rush to offer a solution. If the engineer's perception of the problem is accurate, the solution may work. However, problems are often elusive. The occupants in high-rise buildings always complain of long waits for an elevator. Is this really the problem? And whose problem is it? From the occupants' perspective, the problem might be that the wait is a waste of time. From the building owner's perspective, the problem might be that long waits will reduce occupancy (and thus rental income). The obvious solution is to increase the speed of elevators. But you could also add elevators, stagger working hours, reserve some elevators for express service, increase the rent, or refine the homing algorithm so that elevators go to high-demand floors when they are idle. The range of costs, risks, and time associated with these solutions is enormous. Yet any one could work, depending on the situation. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.

2.3.4 Evaluate Design Alternatives

After the requirements are agreed upon you must examine a variety of architectures and algorithms. You certainly do not want to use an architecture simply because it was used in the requirements specification. After all, that architecture was selected to optimize the understandability of the system's external behaviour. The architecture you want is the one that optimizes conformance with the requirements.

For example, architectures are generally selected to optimize constructability, throughput, response time, modifiability, portability, interoperability, safety, functional requirements. The best way to do this is to enumerate a variety of software architectures, analyze each with respect to the goals, and select the best alternative. Some design methods result in specific architectures. So, one way to generate a variety of architectures is to use a variety of methods.

2.3.5 Use an Appropriate Process Model

There are dozens of process models: waterfall, prototyping, incremental, spiral, operational prototyping, and so on. There is no such thing as a process model that works for every project. Each project must select a process that makes the most sense for that project, on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well-understood.

Study your project's characteristics and select a process model that makes the most sense. When building a prototype for example, choose a process that minimizes protocol, facilitates rapid development and does not worry about checks and balances. Choose the opposite when building a life-critical product.

2.3.5 Minimize Intellectual Distance

Edsger Dijkstra defined intellectual distance as the distance between the real-world problem and the computerized solution to the problem. Richard Fairley has argued that the smaller the intellectual distance, the easier it is to maintain the software. To minimize intellectual distance, the software structure should be as close as possible to the real-world structure. This is the primary motivation for approaches such as object-oriented design and Jackson System Development. But, you can minimize intellectual distance using any design approach. Different humans perceive different structures when they examine the same real world and thus construct quite different realities.

2.3.7 Good Management is more Important than Good Technology

The best technology will not compensate for poor management, and a good manager can produce great results even with meagre resources. Successful software start-ups are not successful because they have great process or great tools (or great products for that matter!), most are successful because of great management and great marketing. Good management motivates people to do their best, but there are no universal right styles of management. Management style must be adapted to the situation. It is not uncommon for a successful leader to be an autocrat in one situation and a consensus-based leader in another. Some styles are innate, others can be learnt.

2.3.8 People are the Key to Success

Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and processes will succeed. The wrong people with appropriate tools, languages and processes will probably fail (as will be right people with insufficient training or experience). When interviewing prospective employees, remember that there is no substitute for quality.

2.3.9 Follow with Care

Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping are some factors that may increase quality, decrease cost, and increase user satisfaction.

However, only those organizations that can take advantage of them will reap the rewards. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal. You can't afford to ignore a new technology. But don't believe the inevitable hype associated with it. Read carefully. Be realistic with respect to payoffs and risks. And run experiments before you make a major commitment.

2.3.10 Take Responsibility

When a bridge collapses we ask, what did the engineers do wrong? When software fails we rarely ask this. When we do, the response is, I was just following the 15 steps of this method.The fact is that in any engineering discipline the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant designs.

☞ Check Your Progress 2

1) Waterfall model is a _____ model.

2) The degree to which a software meets specified requirements is known as _____.

2.4 SUMMARY

In this unit, we learned about the different important qualities of a software product and the process. Also, their definitions were given. The qualities discussed in this unit are correctness, reliability, robustness, user friendliness, verifiability, maintainability, reusability, portability, data abstraction and modularity. Also, various principles of software engineering are discussed.

2.5 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Reusability

2) Modularity

Check Your Progress 2

1) Software Process

2) Software Quality

2.6 FURTHER READINGS

1) *Software Quality, 1997*, Mordechai Ben-Menachem and Garry S.Marliss; Thomson Learning.

2) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.

Reference Websites

- **http://standards.ieee.org**
- **http://www.rspa.com**