QUALITIES OF SOFTWARE

**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-fiendly. The designer of the software want it to use maintainable protable and extensible. In this unit, we will consider all these qualities.**

2.2.1 Correctness

**A program isn functionally correct if it behaves according to the specification of the functions it should provide (called functional requiremnts specifications). It is common simply to use the term correct rather functionally correct; similarly, in this context, the term specificationsB 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 avaiilable and that it is possible to determine unambigiously whether or not a programm meets the specifications does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguties. Regardlessof these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is 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 regorous we are in specifying functional requirments. Correctness can be assessed through a variety of methods, some stressing an experimenal approach (e.g. testing), others stressing an analytic approach (e.g. formal verificatin of correctness). Correstness can also be enhances by using appropriatetools such as high-level languages, particularlt thosee supporting extensive static analysis. Likewise, it can be improved by using stadard algorithms or using libraris of standard modules, rather than inventing new ones.**

2.2.2 Reliability

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

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

**Engineering products are expected to be reliable. Unreliable products, in general, disappear quickly from the marketplace. Unfortunately, software products have not achieved this enviable status, yet software products are commonly releasted along with a list  known bugs. Users of software take it for granted that Release 1 of a product us buggy. This is one of the most strking symptoms of the immaturity of the software engineering field as an engineering discipline.**

**In calsiis engineering disciplines, a product is not released if it hs bugs. You do not expect to take delivery of an automobile along with alist of shortcomings or abridge with a warning not to use the railing. Design errors are extremely rate and worthy of news headlines. A bridge that coollapses may even the designers to be prosecuted in court.**

**On the contrary, software design errors are generally ttreated as unavoidable. Far from being suprised with the occurrence of software errors, we expect them. Whereas with all other porducts the customer receives a gurantee of reliability, with woftware we get a disclaimer that the software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.**

2.2.3 Robustness

**A programm is robust if it brhaves reasonably, even in circumstances that were not anticipated in the requirements specification -f or examplee, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes pefect 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, through, 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 applicaion robust, we would be ableto specify it reasonalbe behaviour completely. Thus, rpbustness pwuld become equivalent to correctness.**

**The amount of code devoted to robustnesss dependd on the application area. For example, a systtem written to be used by novice computer users must be more prepared to deal with ill - formatted input than an embedded system that received its input from a sensor - althouh, if the embeded system is controlling the space shuttle or some life-critical devices, then extra robustness is advisable.**

**In conclusion, we can see that robustness and corrctness are strongly related, without a sharp dividing line between them. If we put a requirement in the specification, its accomplishment becomes an issue of correctness; if we leave it out of the specification, it may become as issue of robustness. The border line between the two qualities is the specification of the system. Finally, re.liability comes in because not all incorrenct behaviours signfy equally serioud problems; some incorrect behaviours may actually be tolerated.**

**Correctness, robustness, and reliability also apply to the software production process. A process is robust, for example, if it can accommodate unanticipated dchanges in the environment, such as a new release of the operating system or the sudden transfer of half the employees to another location. A process is reliable if it consistently leads to the production of high - quality products. In many rengineering desciplines, considerable research is devoted to the discovery of reliable processes.**

2.2.4 User Friendlines

**A software system is user friendly if its human users find it eacy to use. This definition reflects the subjective nature of user friendless. An application that is used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example, a novice user may appreciate bvervbose messages, while an expperience user grows to detest and ignore them. Similarly, a nonprogrammer may apprecaite the use of menus, while a programmer may be more confortable with typing a command.**

**The user interface is an important component of user friendliness. A software system that presents the novice with a window interface and a mouse is friendlier than one that requires the user to use a set of one-letter commands. On the other hand, an experienced user might prefer a set of commands that minimize the number of keystrokers rather than a fancy windows interface through which he has to navigate to get to the command that he knew all along he wanted to ececute.**

**There is more to user friendliness, however, than the user interface. Fow example, an embedded software system does not have a human user interface. Instead, it interacts with hardware an perhaps other software systems. In this case, the user Friendliness is reflected in the case with which the system can be configures and adpated to the hardware environment.**

**In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above such as correctness and performance - also affect user friendliness. A software system that produces wrong answers is not friendly, regardless of how fancy iits user interface is. Also a software system that produces answers more slowly than the user required is not fiendly even if the answerx are displayed in colour.**

**User friendliness is also discussed under the subject humand factors. Human factors or human engineering plays a major role in many engineeering disciplines. For example, automibile manufacturers devote significant effort to deciding the position of the various control knobs on the dashboard. Televison manufacturers and microwave oven makers also try to make their procuts easy to use. User-interface decisions in these classical engineering fields are made, not randomly by enfgineers, but only after extensive study of user needs and attitude by specialists in fields suc has indistrual design or psychology.**

**Interestingly, ease of use in many of these engineering disciplines is achieved through standardisation of the human interface. Once a user knows how to use one television set, he or she can operate almost any other television set. The significant current research and development activity in the area of stadard user interface for software system will lead to more user - friendly systems in the future.**

**2.2.5 Verifiability**

**A software system is verifiable if its properties can be verified easily. For example, the correctness or the performance of a software system are properties we would be interested in verfying. Verification can be performed either by formal analysis methods or thorugh testing . A common t echnique for improving verfiability is the use of software monitors, that is, code inserted in the software to monitor various qualities such as performance or correctness.**

**Modular design, disciplined coding practices, and the user of an appropriate programming language all contribute to verifiability.**

**Verifiability is usually an internal quality, although it sometimes becomes an external quality also. For example, in many security-Critical applications, the customer requires the verifiability of certain properties. The highest level of the security standard for a trusted computer system requires the verfiability of the operating system kernel.**

2.2.6 Maintainability

**The term software maintenane is commonly used to refer to the modification that are made to a software system its initial release. Maintenance used to be viewed as merely bug fixing, and it was distressing to disvcover that so much effort was being spent on fixing defects. Studies have shown, hoever, that the majority of time spent on maintenane is in fact spent on enhancing the product iwth features that were not in the original specifications or were stated incorrectly there.**

**Maintenance is indeed not the proper word to use with software. First, as it is used today, the term covers a wide range of activities, all having to do with modifying an existing piece of software in order to make an improvement. A term that perhaps captures the essence of this proces better is software evolution. Second in other engineering products, such as computer hardware or automobiles or washing machince, maintenance refers to the upkeep of the product in response to the gradual deterioraion of parts due to extended use of the product. For example, transmisisions are olied and air filters are susted and periodically changes. To use the word maintenance with software gives the wrong connotation becuase software does not wear out. Unfortunately, however, the term is used so widely that we will continue using it.**

**There is evidence that maintenane costs exceed 60% of the total costs of software. To analyse the factors that affect such costs, it is customary to divide software maintenance into three categories; corrective, adaptive and perfectiv maintenance.**

2.2.7 Reusability

**Reusability is akin to evolvability. In product evolution, we modify a product to build a new version of that same product; in product resue, we use-oit -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 acceptes uer commands and then executes. But it is desinged to be used both interactivelyy and in batch The ability to start a new shell with a file contaning a list of shell commands allows us to write programs - scripts - in the shell command language. We can view the program as a new pdoeuct that uses the shell as a component. By encouraging standatrd 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 the best known reusable components. Several large FORTRAN libraries have existed for many years. Users can buy these and use them to build their own products, 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 recent development of windowing systems such as X windows or Motif, for the development of user interface.**

**Unfortunately, while reusability is clearly an important tool for recucing software production costs, examples of software reuse in practice are rather rare.**

**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 componets, but the concept has broader applicability; it may occur at different levels and may affect both prodcut and prcess. A simple and widly practived type of reusability consists of the reuse of peopple, i.e. reusing their specific knowledge of an application domain, of a developmetn or target enviroment, and so on. This level of reuse is unsatisfactory, partiallyy due to the turnover of software engineers; knowledge goes way with people and nver becomes a permanent asset.**

**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 desinged, or even at the code level. In the latter case, we might be procvided with software components that are reused from a previous application. Some software expers claim that in thefuture new applications will be produced by assembling together a set of ready-made, 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 on the marketplace for use by other software products.**

**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 are also attempts at reusing higher level processes. Another example of reusability in a process is the r replay approach to software maintenance. In this approach,. the entire process is repeated when making a modification. That is, first the requiremtns are modified, and then the subsequent steps are flollowed as in the intial product development**

**Reusability is a key factor that charactterises, the maturity of an industrial field. We see high degrees of reusability in such mature areas as the automobile industry and consumer electrnics. For example, in the automibile industry, the engine is ofter reused from model to model. Moreover, a car is constucted by assembling together many components that are highly standardised and used across many models produced by the same industry. Finally, the manufacuring process is often reused. The low degree of reusability in software is a clear indication that the field must oevolve to chieve the status of a well-established discipline.**

2.2.8 Portability

**Software is portable if it can run if different environemtns. The term environment can refer to a hardware platform or a software environment such as a particulat operating system. With the proliferation of different processor s and operating systems, protability has become an important issue for software engineres.**

**Mote generally,, portability refers to the ability to run a system on differnt hardware platforms. As the ratio of money spent on software verus hardwae increases, portability gains more importance. Some softtware systems are inherently machince specific. For example, an operating system is written to control a specific computer, and a compiler produces code fodr a specific machine. Even in these cases, however, it is possible to achieve some level of portabilty. 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 the software portable because writing the system from scratch for the new environment would require much more effort than porting it.**

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

2.2.9 Data Abstration

**Abstration is a process whereby we identify the important aspects of a phenomenon an d ignore its details. Thus absttraction is a special case of separation of concerns wherein we separtee the concern of the important aspects from the concern of the unimportant detials.**

**The programming languges that we use are abstractions built on top of the hardware: they provide us iwth useful and powerful constuctions so that we can write (most) programs ignoring such details as the number of nits that are used to represent numbers or the addressng mechanism. This helps us concentrate on the problem to solve rather than the way to instuct the machine on how to solve it. The programms 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 eact details.**

**Data abstraction is a concept which encapasulate (Collect) data structure and well defined procedure /function in a single unit. This encapsulation forms a wall which ins intended to shield the data representation from computer uses. There are two requirements for data abstraction facilities in programming language.**

**(i) Data structure and operations an discribed is a single syntacfic unit.**

**(ii) Data srructure and internal representation of the data abstraction are not visible to the programmer, rather the programemr is presented with a well defined procdural interface. Today most of the object oriented programming oanguage support this feature.,**

2.2.10 Modularity

**A complex system may be divided into similar pieces called modiuules. A system that is composed to modules is called modular. The main benefit of modularity is that it allows the principle of separation of oncerns 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 characterstics of all modules and their reltionships in order to integregate them into a coherent system. If the two phases are temporally executed in the ordr mentioned, then we say 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 constuction off cars proceeds by assembling building blocks that re desinged and built separately. Futhermore, parts are ofen reused from model to mdoel, perhaps after minor changes. Most industrial procsses are essentially modular, made out of work packages that are combines in simple wasy (sequenctlly or overlapping) to achieve the desired result.**

**We will emphasise modulaity in the context of software design in the next chapter. Modularity, however, not only in a desirable design principle, but permeates the whole of software peoduction. In particular, there are three foals that modularity tries to achieve in practice: capability of decomposing a complex system of 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 subproblems and then applying the decompostiong to each suproblem recursively. This procedure reflects the well known Latin motto divide et impera (divide and conquer), which descrivbes the philosop followed by the ancient Romans to dominate other nations: divide and isolate them first and conquer them individually.**

**The composability of a system is based on staring bottom up from elementary components and proceedng to the finished system. As an example, aa system for office automation may be desinged by assembling together existing hardware components such as personal workstations, w 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 aobious example of a system that is built by assembling compoents. Consider first the main subsystems into which a car may be decomposed; the body the electical 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. from the electrical system. When something goes wrongg, defective components may be replaced ny new ones.**

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

**The capability of undersanding each part of a system separately aid in modifying a system. the evolutionary nature of software is such that the software enginner is often required to go back to previos work to modify it. If the entire system can be understood only in its entirely, modifications are likely to be difficult to apply, and the result unreliable. When the need for repair arises, proper modularity helps confine the search for the source of malfuncion to single compoents.**

**To acheve modular composability, decomposability, and understanding, modules must have high cohesion and low coupling.**

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

**Whereas cohesion is an internal proeprty of a module, coupling characterises a module’s relationship to other modules. Coupling measures the intrdependcne of two modules (e.g. module A calls a routine provided by module B or accesses a variable decalred by module B). If two modules depend on each other heavily, they have high coupling, because if two modules are highly couplied, it will be difficult to analyse, understand, modify, test or reuse the m separately.**

**Modular structures with gigh cohesion and low coupling loow us to see modules as black module separetely when the module’s functioality is described or analysed. This is just another example of the principle of separation of concerns.**