Engineering Journal of University of Qatar, Vol. 8, 1995, p. 83 - 100

ON IMPROVING COMMUNICATION SOFTWARE DEVELOPMENT PROCESS

Magdi M. Said El-Soudani
Electrical Engineering Department, Faculty of Engineering,
University of Qatar, Doha, Qatar.

ABSTRACT

This paper aims to standardize the software development process by making use of the well-established hardware development process. The procedure can be applied to any engineering software development process, but we consider the case of communication systems. We propose a model for software development process that has a hierarchical structure. We examine this hierarchical structure together with the development process at each level. Each process has several stages, and we use the same definitions at all levels. The development process consists of the following stages: requirements and specification, flow detailed structure, source coding, and testing stages. Integration of the whole system is performed in steps through different testing levels. During the development process we measure the performance or the quality of each stage. We consider the factors that have great impact on software quality and that may be the cause of defects. These includes complexity, supporting tools and methods, and system ability, and from these factors we define some quality attributes to measure the system quality.

INTRODUCTION

Software is a major component of any large or complex communication system. The software development process for communication systems differs from other application software development processes. A small fault in the software may lead to a complete failure of the communication system. For example, in January 1990, AT&T experienced a software fault that impaired the ability of users to complete calls across the AT&T network. Also in June 1991, a software fault caused in signaling system No.7 (SS7) led to major outages in several regions in the United States [1]. Apart from the large number of users in communication systems who would be affected, the hardware has to be reset after

such a failure and a part of the information may be lost. That is why the communication software should be solid and robust.

The software development of communication systems normally takes long time compared with other systems. Moreover, we sometimes need to develop both software and hardware in parallel. In most of the cases, the hardware is located behind the software and the software interfaces with the user's needs. Therefore the external specifications of the service are clear for the user, but for the developer or designer the specifications of the required hardware should be clear and feasible. This puts a great burden on the software developer who should also be knowledgeable in hardware in order to consider the interactions between software and hardware. Also, in some communication systems, rapid prototyping is need. This means modeling some functional capability of the actual system in order to reduce uncertainty regarding requirements. This method helps the system developer as well as the user to know the exact output for the given requirement. This also gives them the chance to modify the system at early stages. Also in communication software there is a strong push to reuse software developed for one service, for example, within another one. In this case we have to assure that the reliability of reused software will be sufficient to satisfy the needs of the new service. On the other hand, careful inspection and revision before putting software into operation or service does not prevent some operational faults. Some of these faults or " bugs " may lead to severe failure of the communication system. Communication systems software should be able to identify and isolate any fault that may lead to such a failure. Even if this failure occurs, the system should be able to recover from it. Therefore communication software should be designed and built with these types of system failures in mind [2],[3].

The following sections are all related to software development processes and quality measurement. Whenever software is mentioned it means software for communication systems. A software hierarchical structure is described in the next section together with the proposed development process model. Then, we explain the integration process and define several testing stages. To evaluate the developed software we define a set of quality attributes and show how they are used to measure the software quality. We conclude the paper with comments on tradeoffs among cost and quality in the proposed software development process.

SOFTWARE DEVELOPMENT PROCESS

In this paper, we make use of similarities between hardware and software to develop a model for software development. From this model we can determine the factors that will improve the development process. Improving the working process

is possible if we standardize the software development process. Therefore, it is necessary to put the software itself in a tree or hierarchical structure. By definition, software is a kind of transformation process of a discrete set of inputs into a set of outputs taking into consideration the system resources. In switching systems a set of input parameters from the hardware circuitry indicating the subscribers states will make the software generate different output signals to the hardware circuitry such as connect tone, disconnect ringing signal, or release a connection. The software has to consider the availability of free paths between the calling and called subscribers. Therefore, this flow of information has a hierarchical structure by nature. We can decompose the software into levels as shown in Fig.1. These levels from the top are the system, module, function, and unit levels. It is well understood that the decomposition is *a posteriori* to a complete study of the system requirements and functionality. On the other hand, the software development process is a chain of design processes. As the design goes on, the work expands. In order to simplify the design and to standardize the development process, we decompose the process into levels to follow the same software hierarchical structure. As the development progresses, the work is divided

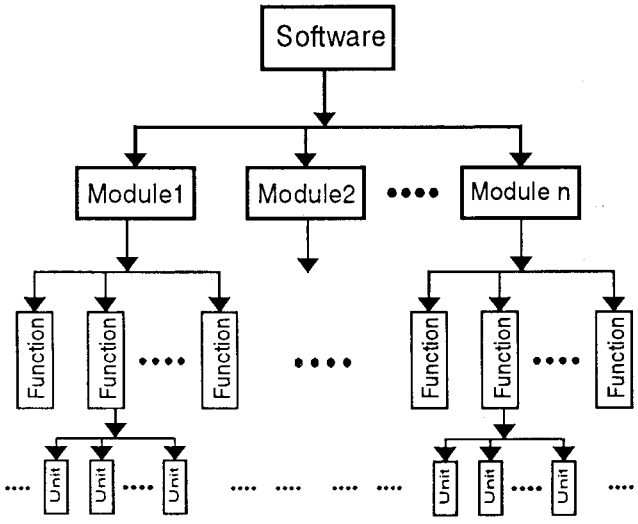

Fig.1: Software hierarchy

into steps that are smaller, clearer, and more solid. Finally the work reaches elementary operations. For example, dialing process can be divided into units with

simple functions such as read the dialed digit and send the proper output to the tone encoder. A higher level function will repeat this read/write procedure for all the dialed digits. The development processes at the different levels are nearly identical.

Figure 2 shows the elements of a software development process. We have three main elements; input, process, and output. The input sets the concepts and definitions of the system. Process means design, coding, and compilation. It also means verification and testing of different developing stages. Figure 3 shows the main stages in the process itself that must be considered at each developing level. These stages form the so-called software life cycle. More stages can be added to the life cycle if more detailed structure is required such as installation, and maintenance stages [4]. Other elements are included in Fig. 2 such as resources that mean the necessary tools, and equipment. The schedule means the time planning for the process. Personnel are simply the people necessary to develop, analyze, and test the software. Methodology here means the developing methods and engineering techniques. The process is surrounded by environmental factors that influence the development process.

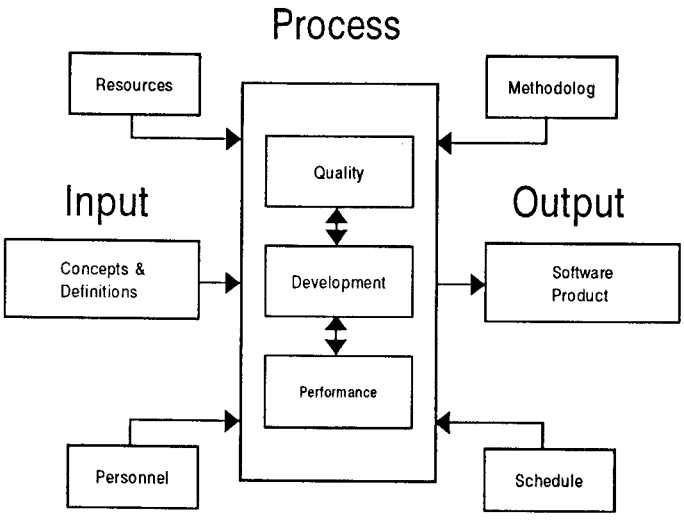

Fig.2 Elements of software development process

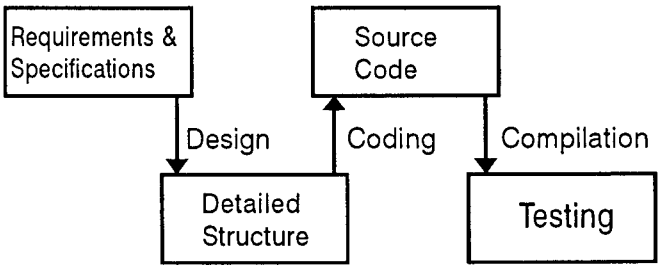

Fig. 3: Stages of software development process

Usually, software specifications must come from the analysis of user's concepts and definitions. In communication systems, software specifications are given exactly or in the form of a finite state machine (such as the call handling procedure, dialing process and switching procedure) and the user has little say in the specifications in this case. In the design stage, the requirements and specifications are translated into description of software system that can be coded. From communication network perspectives, it is important to design software that is robust. Even when hardware or software is encountered in any sort of troubles such as overloading of a switch, repeated fault messages, or non-acknowledgment of messages, the software should be able to identify and isolate the fault and the system should be able to recover. In the design of development process we take into considerations the hardware design process. Therefore software and hardware design processes will have some similarities. The hardware design process will be the same as that for software. For example the detailed flow chart will be replaced by detailed logic diagram, the coding stage will be replaced by detailed circuit diagram, and the compilation in software process is equivalent to the implementation of printed circuit board (PCB) in the hardware process.

SYSTEM INTEGRATION AND TESTING

At the testing stage the development process converges again according to the system hierarchical structure and finally results in the complete software program. There are several testing stages in the software developing process. These are unit, function and module integration, and system testing stages as shown in Fig.4. Other testing stages may be added such as product, customer, system verification, and regression testing [5]. Unit testing occurs when programmer tests

the run programs of each unit separately, while integration testing is the testing of previously separate units of the software when they are put together. The integration test is performed at the function level and at the module level when several software parts are grouped together to perform a specific function. System testing is the testing of the functional part of the software to determine that it performs its expected function. Product testing aims to test the functionality of the final system while customer testing is often a product testing performed by the intended user of the system. If the new software is a revised version of already existing software, regression testing is needed to assure that the new version of the software faithfully reproduces the desirable behavior of the previous one.

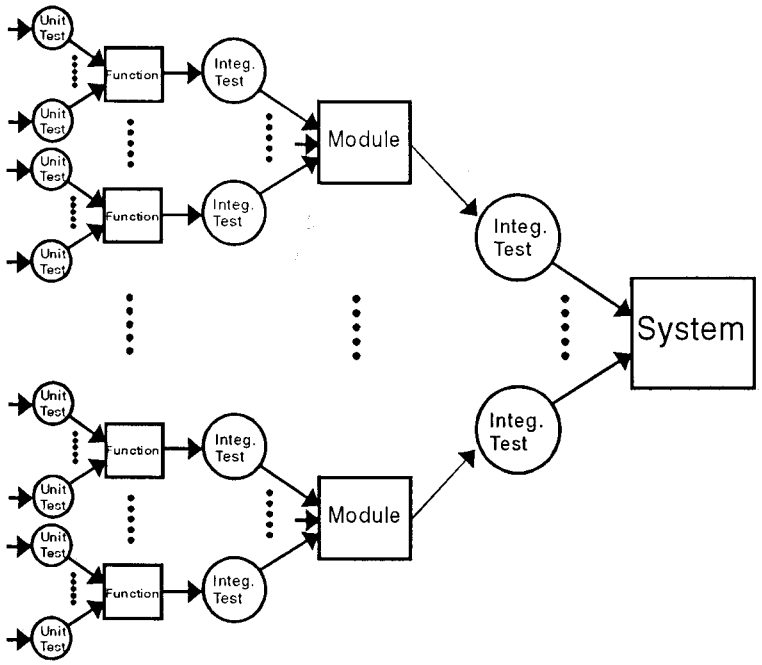

Fig.4 Testing stages

To make sure that those jobs controlling the hardware are operating properly we conduct system verification tests. In this case, the previously tested portions of the software that passed the several testing levels, from the software point of view, are tested with their related hardware circuitry. For example, system verification test is needed to check if the proper addressing is used to select a cross point switch in an array of switching matrices. Sometimes another testing stage is needed to identify those parts of the software that have not been exercised during

testing. This type of testing is called coverage test. All these testing stages aim to assure that program's specifications are met by exercising the features described in the specifications. This depends on the specifications of the program and is independent of its coding.

The number of faults during testing decreases with time assuming that the source of fault is detected and recovered. Poisson distribution is the most adequate method to predict the number of fault. Let x be the number of faults in time t, the probability to get x faults will be given by

$$\Pr\{x\} = \frac{[\lambda]^x}{x!} e^{-\lambda} \tag{1}$$

where λ is the expected fault rate in time t. Since the number of faults varies with the testing time, the fault rate will be a function of testing time, and the system will follow the nonhomogeneous Poisson process (NHPP). This case is called the Goel-Okumoto (GO) model [6]. The fault rate itself can be expressed as a function of cumulative number of faults during time t. From the observation of faults in different software development processes, we can describe the cumulative number of faults $m_1(t)$ by the exponential relation

$$m_1(t) = a\,(1 - e^{-bt}) \tag{2}$$

where a is the expected number of faults to be observed initially and b is the fault detection rate. The corresponding fault rate $\lambda_1(t)$ will be given by:

$$\lambda_1(t) = m_1'(t) = ab\, e^{-bt} \tag{3}$$

This fault prediction process cannot distinguish between hardware and software faults because symptoms of hardware faults are often similar. Other forms of the function $m(t)$ can be used depending on the observations [6]. Sometimes when the system is tested in normal working conditions the fault rate increases then decreases. The cumulative number of faults in this case may take the form:

$$m_2(t) = a\,(1 - e^{-bt^c}) \tag{4}$$

where c is a constant selected upon the type of testing and required quality. This a generalization of the GO model. The corresponding error rate will be given by:

$$\lambda_2(t) = m_2'(t) = abe^{-bt^c} t^{c-1} \quad (5)$$

Figure 5 shows the cumulative number of faults using equation (2) and (4) assuming that the expected number of faults is 100 and the fault detection rate is 0.075. The constant c is chosen to be 1.25. The corresponding fault rates are shown in Fig.6.

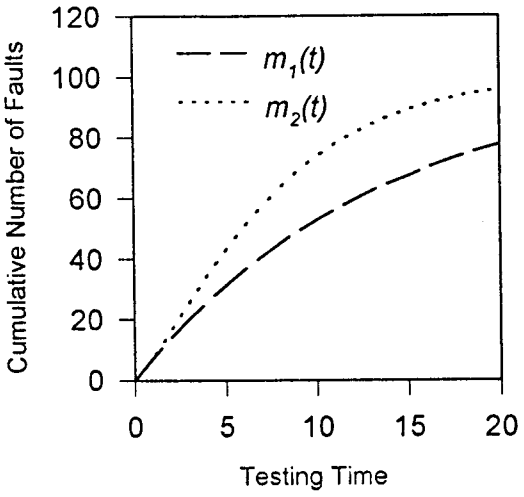

Fig. 5: Cumulative number of faults, $m_1(t)$ and $m_2(t)$

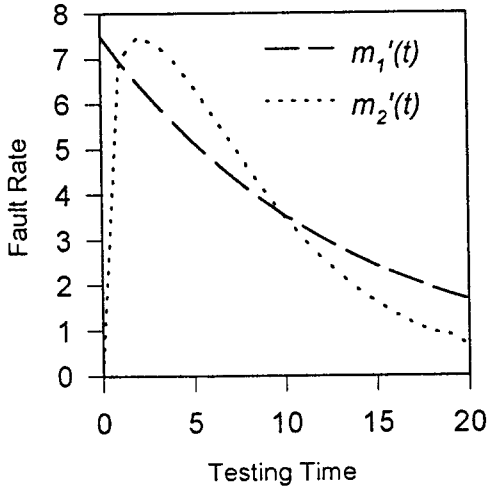

Fig.6: Fault rate $\lambda_1(t) = m_1'(t)$ and $\lambda_2(t) = m_2'(t)$

On the other hand, without knowing the value of testing it is hard to decide when to stop testing, or how to trade testing time against code inspection time, or to assess the effect of testing on quality, or even to decide whether testing is worthwhile. Testing cost may be used as a guide to determine when to stop. Testing cost is proportional to the number of detected faults and the effort necessary to fix the faults. The total testing cost $C(t)$ till time t will be given by:

$$C(t) = An_d(t) + B(1-\delta)\, n_d(t) + E$$

$$= D\, n_d(t) + E \qquad (6)$$

and $D = [A+B(1-\delta)]$

where $n_d(t)$ is the number of faults detected in time t, δ is the portion of unfixed faults. A and B are constants that represent the average effort done to detect and fix the fault and E represents other expenses in the testing procedure. If we consider the NHPP model, $n_d(t) = m_1(t)$. It remains to determine the quality of the developed system. Software quality measurement will help us to answer this question.

FACTORS AFFECTING SOFTWARE QUALITY

Classically, the term quality means " bug-free ". In software development, various defects are caused by imperfections in the process leading to differences between what should be and what actually is. As software development moves towards its final target and completion, the cost to repair a fault in the application soars. Development economists thus argue strongly for testing and tuning software as early as possible in its life cycle, preferable before the integration phase. The cost of correcting a code fault during the coding stage is a matter of changing the code, recompiling, and testing. On the other hand, the cost of fixing a design fault found in the system integration test for example will include the cost of correcting the design, changing the code, passing through all testing stages. The cost of fixing faults in operation phase, i.e., after the release of the software, may be several times the cost required to fix a fault in coding phase. So it is cost-effective to avoid, if possible, faults and errors in the final stages of the software life cycle. It is always said that where quality cannot be defined, it cannot be measured; where it cannot be measured it cannot be controlled. It is much better

to find what people think about factors that have great influence or impact on quality, and from these factors we can define the quality measurement methods. Among these factors are complexity, methodology, and system ability. Complexity considers, in general, the size of the program, the required memory, the control flow, the link between different modules, and the number of input/output operations between software and hardware. Methodology is concerned with the engineering techniques and supporting tools as well as the language adopted for the development. System ability is the capability of the developed system to satisfy all the requirements as well as its ability to be modified or expanded. The capability of personnel to achieve the final objectives within the constraints of the time schedule is also considered as part of system ability.

From the hierarchical structure we measure the software complexity as function of the number of variables in each module and each subprogram. We also consider the common variables among the subprograms. Let $\alpha(i)$ denote the number of variables in subprogram i, and $\beta(i)$ be the number of common variables used by the subprogram i. The structural complexity $\chi(j)$ of module j with n subprograms will be given by:

$$\chi(j) = \left(\sum_{i=1}^{n} \alpha(i)\beta(i) \right) (1 - \delta(j)) \tag{7}$$

where $\delta(j)$ is a constant depends on the amount of comments in jth module and $0 \leq \delta(j) \leq 1$. The overall program structural complexity χ will be given by:

$$\chi = \sum_{j=1}^{k} \chi(j) \tag{8}$$

where k is the number of modules in the program. Another measure of software complexity is the operational complexity. We define the operational complexity $\chi_{p(j)}$ of module j as:

$$\chi_{p(j)} = \frac{\mu_j}{1+\varepsilon_j} N_a \tag{9}$$

where μ_j is the utilization factor that reflects the access frequency of module j during program operation. N_a denotes the total number of access of module j during testing and ε_j is the backtracking degree of module j during testing. This

means that if the main program scans all modules to initiate the required service routines we can find the utilization factor of the jth module by counting the number of calls or accesses during one cycle. The utilization μ_j is defined as the average number of accesses of module j divided by the minimum number of accesses. In the same way we can find the operational complexity of each unit or routine in any module. The total program operational complexity will be the sum of the modules complexity. Other parameters can be used to measure the system complexity such as the average size of module's procedures. Also the size of modules and the density of branching and transfer of control commands.

Programming language and techniques are among the factors that affect the software quality. There is no direct way to evaluate the language or the technique. We can say that the program size and the execution time are among the factors used to evaluate the system methodology. The program size is usually measured in terms of the number of lines of code (LOC) or in thousands of LOC, i.e., KLOC. There is no unique definition of LOC since it is a language dependent. Therefore it is not possible to compare systems developed using different language. We prefer to use the volume of the executable file rather than that of the source file. The execution time can either be measured in terms of the number of processing cycles or in CPU- time if possible. The execution time per line of code will be a measure of the efficiency of the adopted methodology, and it will be given by:

$$M = \frac{\tau_E}{v} \tag{10}$$

where τ_E is the execution time (or CPU- time) and v is the size of the executable file. Other methods are suggested in [7].

QUALITY MEASUREMENT

Today's quality concept has been widened to include many factors such as usability, functionality, understandability, etc. Software quality could be evaluated using various quality attributes [8]. However, no single set of quality attributes has been adopted for use as a standard set. The attributes selected mainly depend on the software project's objectives, developing environment, and on the evaluator. The fact that individuals disagree about what is quality is an argument in itself. From the factors that affect the software development process, we define a set of quality attributes to measure the software quality. Each quality attribute in itself can be divided into subattributes. For example, the quality attributes used to

measure the effects of system complexity are understandability, masterability, and usability. We refer to this group of attributes as the clarity group. The operability, modifiability, and expandability attributes measure the adopted programming language and developing tools. These attributes form what we call capability group. The impact of methodology and supporting tools on the developed software is evaluated using the functionality, the suitability, and the performability attributes. We refer to these set of attributes as the technicality group. Each group measures a set of quality factors, and from these measurements we can conclude the quality of the developed software. Some of the above mentioned attributes can be measured quantitatively while others are difficult. Special metrics have to be used with the latter type of attributes. Sometimes, different attributes are used to measure the same factor but from different point of views. Some of these attributes are used to evaluate the developing tools themselves such as suitability and operability [8]. In what follows, we describe briefly the functions of the attributes in the above mentioned groups.

Clarity Group

- Understandability determines how well the software is structured so that the flow diagram can be easily followed. Also it shows whether the language is easy to understand or not.
- Masterability measures how much one can master the software development tools and the language used.
- Usability measures the ease of use of the software, sometimes called "friendliness".

Capability Group

- Operability measures the ease of using and mastering the system operating rules and functions.
- Modifiability determines the simplicity of system modifications in case of changing requirements or specifications.
- Expandability measures to what limit the system can be expanded to include any new functions.

Technicality Group

- Functionality determines how the functions developed or used are easy to realize. It also measures the completeness.
- Suitability determines the degree of ease for implementation of the software requirements using the adopted technique.
- Performability measures the system efficiency. It also measures the system communicativeness and interoperability between the software and hardware.

Table-1 summarizes the functions of the different quality attributes and shows the level at which these attributes can be measured as well as the methods used to measure them.

Table 1: Quality Attributes

GROUP	ATTRIBUTE	MEASURED FACTORS	MENT LEVEL	MEASURED USING
Clarity	Understandability Masterability Usability	S/w structure. Language and tools. User's effort.	Overall system	Training hours. Document size. Number of modules. Test parameters.
Capability	Operability Modifiability Expandability	S/w rules and functions. Ease of changing parameters. Ease of adding new modules.	Overall system. Module, function and unit levels.	Man-hours and development volume (KLOC).
Technicality	Functionality Suitability Performability	Realization Methodology Efficiency	Overall system	Number of variables. Development volume. Number of detected bugs.

Each quality attribute can be divided into subattributes if there many parameters to be considered in the evaluation process. Also it is not necessary to select all the defined attributes in the evaluation process. We assign a weighting factor W_i to each attribute depending on its impact on the overall system quality. An evaluating rate r_i will be assigned to each subattribute. The maximum rate is R_i. The total rate for a particular attribute α_j will be given by:

$$\alpha_j = \frac{\sum_{i=1}^{n} r_{ij}}{\sum_{i=1}^{n} R_{ij}} \qquad (11)$$

where n is the number of subattributes in the jth attribute. The overall quality rate for a k selected attributes will be given by:

$$Q = \sum_{j=1}^{k} \alpha_j W_j \tag{12}$$

On the other hand, reliability measures the system non-deficiency, error tolerance, and availability. We can consider software reliability as a probabilistic measure of software quality. It is defined as the probability that software faults do not cause a fault during a specific period in a specified environment. However, the use of fault rate is sometimes misleading [8]. Some faults do not occur if small portions of the software are tested separately. Therefore covering test is necessary to exercise those portions of the software that have not been tested. A number of analytical models exist to find the reliability. All these models depend on the fault history of the software and differ only in the nature of the fault process they consider. In communication systems another quality factor has to be considered. This is the software service-quality that used to quantify customer perceptions and expectations of delivered services during the trial state of the developed software. Another important factor to determine the system quality is the development cost or what is called quality cost. The cost includes all the costs that have been spent to ensure that the software satisfies its requirements and specifications. It is not easy to list all types of cost in the development process. Accurate measurements of development costs are needed through the whole development process. However, it is important to have a cost estimation before hands at the beginning of the development process as we have done to estimate the cost of testing. This is an important factor to improve the development process and this requires a good understanding of the details of the process. This means that a cost estimation model that is closely related to the development process model should be developed. This cost model must follow the same life cycle as the development process (requirements, design, test, etc.). It is not our concern in this work to develop such a model, and different models are given in the literature [9]. What is important is to show that the development process model is the milestone in any software development project.

We summarize the evaluation procedure of the developed software in the following steps:

I- Requirements and Specifications

- Determine if the requirements and specifications can really be implemented with the available resources.
- Identify the redundancy , if any, in the requirements to avoid overdoing the job.

- Determine the final objectives for each possible requirement.
- Provide an assessment of the suitability of the development tools (design, methodology, programming language, ..etc.).

II- Code Inspection

- Inspect the program code for each developed unit.
- Determine the number of functions per module and the number of units per function.

III-Software Analysis

- Determine the number of variable per function as well as per module.
- Calculate the structural complexity for each module using equation (7) and then the overall system complexity using equation (8).
- From the transfer of control commands find the utilization factor μ for each module.
- Find the operational complexity using equation (9).
- Estimate the average CPU-time (or number of execution cycles) per LOC using equation (10).
- List all these in table-II for each module.

Table 2: Code Inspection and Software Analysis

Number of LOC.	
Number of Comment lines	
Number of variables	
Number of common variables	
Structural Complexity χ	
Utilization facto μ	
Number of access N_a of Module j	
Operational complexity χ_p	

IV-Fault Observations

- Find the time between faults and the cumulative number of errors during the test period.
- Estimate the expected number of faults a and the detection rate b and use these parameters to predict the expected fault rate during the testing period using equations (2) or (4). For equation (4) select c according to the required quality rate (large values of c for low fault rates).

V-Quality Measurements

- Choose a suitable rating scheme (for example; 3-point system for discard, fair, and good) to measure each quality attribute.
- Assign a weighting factor to each attribute depending on its importance in the developed system using a table similar to Table-III.
- Determine the pass criteria (for example, small structural complexity with moderate quality rate).
- Find the percentage of the achieved score for the overall system using equations (11) and (12).
- If the quality rate is beyond requirement and cost of testing is less than permissible limits, readjust the system parameters (measured factors from table-III with low rating), otherwise discard the process.

Table 3: Quality Measurements

GROUP	ATTRIBUTE	MEASURED FACTOR	RATING	MAX. RATING	WEIGHT
Clarity	Understandability Masterability Usability				
Capability	Operability Modifiability Expandability				
Technicality	Functionality Suitability Performability				

CONCLUSION

In this work we have made use of the analogy between hardware and software to develop a software development process model. We have also made use of the hierarchical structure of the software to decompose the development process into small units or jobs that are easy to design, code, and test. This decomposition process of the software requires very restricted testing rules. Small units in the lower level of the hierarchical structure are first tested separately to detect code errors. When these units are integrated together to perform the required function they are tested again at the function level. The process continues till the whole software is tested. This complicated testing procedure is the major drawback in the hierarchical structure, but in this way most of the bugs or faults will be detected and filtered at early developing stages. This will considerably reduce the

development cost. The hierarchical structure does improve the system performance and reduce the development cost. Of course the different testing stages add to the development cost, but it is better to spend some money to detect faults at early developing stages rather than spending much more in order to find the cause of fault. There is always a trade-off between cost and quality. Another advantage of hierarchical structure is that the development can be run in parallel, and this is important in case of tight time schedule projects. In fact, there are few costs to reduce in software development process. The only savings lie in reducing the cost of finding defects and fixing them. In communication systems, software faults are sometimes simple and easy to detect and find either by testing or inspection, but the hardware interactions are not usually easy to analyze. Without a careful inspection and testing of every portion of the software at the various testing levels, even the detection of the software fault and supported hardware deficiency may be insufficient to predict the magnitude or the severity of the fault.

The quality attributes defined here are mainly based on the factors affecting software performance. Since most of these factors are common among software, these attributes can be applied to any software. It is not necessary to consider all these attributes. Subset of these attributes will give at least a good estimate of the software quality. The quality measure method can be further improved if statistical analyses of fault occurrence at each developing level in the hierarchical structure are considered. Finally, we have to consider that a software product is a man-made product, and people constitute an essential part of the development process. Therefore, relation between the people and their work is a key factor.

REFERENCES

1. **Bates, R.J., 1992.** "Disaster Recovery Planning, Networks, Telecommunications, and Data Communications", McGraw Hill.

2. **Chung, F., 1994.** "Software and Communication I: An Overview", IEEE JSAC, Vol-12, No.1, pp. 23-32.

3. **S. Dallas, J. Horgan, and J. Kettenring, 1994.** "Reliable Software and Communication II: Controlling The Development Process", IEEE JSAC, Vol-12, No. 1, pp.33-39.

4. **Smith, C. and L. Williams, 1993.** "Software Performance Engineering: A case Study Including Performance Comparison With Design Alternatives", IEEE Trans. Software Eng., Vol-19, No.7, pp.720-741.

5. **Chaar, J., et al, 1993.** "In Process Evaluation for Software Inspection and Test", IEEE Trans. Software Eng., Vol- 19, No. 11, pp. 1055-1070.

6. **Goel, A., 1985.** " Software Reliability Models: Assumptions, Limitations, and Applicability ", IEEE Trans. Software Eng., Vol.-11, No.12, pp.1411-1424.

7. **Song, X. and L. Osterweil, 1994.** "Experience with an Approach to Comparing Software Design Methodologies", IEEE Trans. Software Eng., Vol-20, No. 5, pp 364-384.

8. **Miyoshi, T. and M. Azuma, 1993.** "An Empirical Study of Evaluating Software Development Environment", IEEE Trans. Software Eng., Vol-19, No. 5, pp.424-435.

9. **Matson, J., J. Barrett, and J. Mellichamp, 1994.** "Software development Cost Estimation Using Function Points", IEEE Trans. Software Eng., Vol-20, No.4, pp. 275-285.