UNIT 1 SOFTWARE PERFORMANCE

Structure **Page Nos.**

1.0 INTRODUCTION

The Software development team works towards the goals of producing high quality software within the target dates. Software quality Assurance (SQA) is a set of activities designed to evaluate the process by which products are developed or manufactured.

The Software Requirements Specification (SRS) captures all the desirable properties of the application and no undesirable properties should be specified in it. The set of all reliable programs include the set of correct programs, but not vice versa. Unfortunately, things are different in practice. In fact, the specification is a model of what the user wants, but the model may or may not be an accurate statement of the user's needs and actual requirements. All that the software can do is to meet the specified requirements of the model. It cannot ensure the accuracy of the model.

Software Reliability may be defined as the ability of software to perform its required functions under stated conditions for a specified period of time. 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, 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. If we could state precisely what we should do to make an application robust, we would be able to specify its behavior completely. Thus robustness would become equivalent to correctness of reliability.

An analogy with bridges is instructive. Two bridges connecting two sides of the same river are both "correct" if they each satisfy the requirements. I however, during an unexpected, unprecedented, torrential rain, one collapses and the other one does not, we can call the latter more robust than the former. Notice that the lesson learned from the collapse of the resistance to torrential rains is a correctness requirement. In other words as the phenomenon under study becomes more and more known, we will approach the ideal case where specifications capture expected requirements.

The amount of code devoted to robustness depends on the application. For example, a system written to be used by novice computer users must be more prepared to deal with ill-formatted input than an embedded system that receives its input from a sensor.

If the embedded system is controlling the space shuttle or some life critical devices, then extra robustness is advisable.

In conclusion, we can see that robustness and correctness 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 an issue of robustness. The border line between the two qualities is the specification of the system. Finally, reliability comes in because not all incorrect behaviors signify equally serious problems; some incorrect behaviors may actually be absorbed.

Correctness, robustness, and Reliability also apply to the software production process. A process is robust; for example, if it can accommodate unanticipated changes 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 engineering discipline, considerable research is devoted to the discovery of reliable processes.

Any engineering product is expected to meet a certain level of performance. Unlike other disciplines, in software engineering, we often equate performance with efficiency. A software system is efficient if it uses computing resources economically.

Performance is important because t affects the usability of the system. If a software system is too slow, it reduces the productivity of users, possibly to the point of not meeting their needs. If a software system uses too much disk space, it may be too expensive to run.

Underlying all of these statements and also what makes the efficiency issue difficulty are the changing limits of efficiency as technology changes. Our view of what “too expensive” is constantly changing as advances in technology extend the limits. The computers of today cost orders of magnitude less than computers of a few years are yet they provide order of magnitude of more power.

Performance is also important because it affects the scalability of a software system. An algorithm that is quadratic may work on small inputs but not work at all for larger inputs. For example, a compiler that uses a register allocation algorithm whose running time is the square of the number of program variables will run slower and slower as the length of the program being complied increases.

There are several ways to evaluate the performance of a system. One method is to measure efficiency by analyzing the complexity of algorithms. An extensive theory exists for characterizing the average or worst case behavior of algorithms, in terms of significant resource requirements such as time and space, or-less traditionally in terms of number of message exchanges (in the case of distributed system).

Analysis of the complexity of algorithms provides only aver average or worst case information, rather than specific information, about a particular implementation. For more specific information, we can use techniques of performance evaluation. The three basic approaches to evaluating the performance of a system are measurement, analysis, and simulation. We can measure the actual performance of a system by means, of monitors that collect data while the system is working and allow us to discover bottlenecks in the system. Or we can built a model of the product and analyze it. Or, finally, we can even build a model that simulates the product. Analytic models-often based on queuing theory-are usually easier to build but are less accurate while simulation models r more expensive to build but are more accurate. We can combine the two techniques as follows: At the start of a larger project, an analytic model can provide a general understanding of the performance-critical areas of the product pointing out areas where more thorough study is required. Then, we can build simulation models of these particular areas.

In many software development projects, performance is addressed only after the initial version of the product is implemented. It is very difficult. Sometimes, it is impossible to achieve significant improvement in performance without redesigning the software. Even, a simple model is useful for predicting system performance and guiding design choices so as to minimize the need for redesign.

In some complex projects, where the feasibility of the performance requirements is not clear, performance models are built. Such projects start with a performance model and use it initially to answer feasibility questions and later in making design decisions. These models can help resolve issues such as whether a function should be provided by software or a special-purpose hardware device, etc.

The notion of performance also applies to a process, in which case we call it productivity. Productivity is important enough to be treated as an independent quality factor.

1.1 OBJECTIVES

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

- Know the definitions of some important terms associated with the performance of software;
- Know different software tools, and
- Know various development tools.

1.2 CUSTOMER FRIENDLINESS

A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. 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 any appreciate verbose messages. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command (though this may not be the case always).

The user interface is an important component of user friendliness. A software system that presents a novice user with a window interface and a mouse is friendlier than the 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 keystrokes rather than a fancy window interface through which he was to navigate to get to the command that he knew all along he wanted to execute. There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts with hardware and perhaps other software systems. In this case, the user friendliness is evaluated based from the ease with which the system can be configured and adapted 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 its user interface is. Also, a software system that products answers more slowly than the user requires is not friendly even if the answers are displayed correctly and in different fonts.

User friendliness is also discussed under the subject “human factors”. Human faction or human engineering lays a major role in many engineering disciplines. For example, automobiles manufactures devote significant effort in deciding the position of the various control knobs on the dashboard. Television manufactures and microwave oven makers also try to make their products easy to user. User-interface decision in these classical engineering fields are made, not randomly by engineers, but only after extensive study of user needs and attitudes by specialists in fields such as industrial design or psychology.

Interestingly, ease of use in many of these engineering disciplines is achieved through standardization of the human interface. Once a user knows how to use one television set, s/he operates almost any other television set.

1.3 SOFTWARE RELIABILITY

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

If an organization depends on a software for its functions then it is reliable software. Reliability of software is an important factor of its overall quality. Software Reliability factor can be measured and estimated. In statistical terms, the probability of failure free operation of software in a particular environment is defined as Software Reliability. Software Reviews form an important part of Software quality Assurance Activity. The Quality Assurance team must collect data about Software Engineering Process, evaluate the data and disseminate. The ability to ensure quality is the measurement of a mature engineering discipline.

Correctness is an absolute quality: A deviation from the requirements makes the system incorrect, regardless of how minor or serious is the consequence of the deviation. The notion of reliability is, on the other hand, relative. If the consequence of a software error is not serious, then 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 achieve this enviable status yet. Software products are commonly released along with a list of "Known Bugs". Users of software take it for granted that "Release 1" of a product is "buggy".

In classic engineering disciplines, a product is not released if it has "bugs". You do not expect to take deliver of an automobile along with a list of shortcomings. Current research and development activity in the area of standard user interfaces for software systems will lead to more user-friendly systems in the future.

☞ Check Your Progress 1

1) In statistical terms, the probability of failure free operation of software in a particular environment is defined as _____________.
2) Software Reviews form an important part of ______________.

1.4 SOFTWARE REVIEWS

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. For example, the correctness or the performance of a software system are properties we would be interested in verifying. Verification can be performed either by formal analysis methods or through testing. A common technique for doing verifiability is the use of "software monitors".

Modular design disciplined coding practices, and use of an appropriate programming language contribute to verifiability.

Verifiability is usually an internal quality.For example, in many security-critical applications, the customer requires the verifiability of certain properties. The highest level of security standards for a "trusted computer system" requires the verifiability of the operating system kernel.

Software maintenance is commonly used to refer to the modifications that are made in a software system after its initial release. Maintenance used to be viewed as merely "bug fixing," and it was distressing to discover that so much effort was being spend on fixing defects. Studies have shown, however, that the majority of time spend on maintenance is in fact spend on enhancing the product with features that were not in the original specifications or were stated incorrectly.

"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 o do with modifying an existing piece of software in order to make improvement. A term that perhaps captures the essence of this process better is "software evaluation". Secondly, in other engineering products, such as computer hardware or automobiles or washing machines, "maintenance" refers to the upkeep of the product in response to the gradual deterioration of parts due to extended use of the product.

There is evidence that maintenance costs exceed 60% of the total costs of software. To analyze the factors that affect such costs, it is customary to divide software maintenance into three categories: corrective, adaptive and perfective maintenance.

Maintenance performed to correct faults in hardware or software is known as Corrective Maintenance. Corrective maintenance accounts for about 20 percent of maintenance cost.

Software maintenance performed to make a computer program usable in a changed environment is known as Adaptive maintenance. Adaptive and perfective maintenance are the real sources of changes in software. They motivate the introduction of evolvability as a fundamental software quality and anticipation of change as a general principle that should guide the software engineer. Adaptive maintenance accounts for nearly another 20 percent of maintenance costs while over 50 percent is absorbed by perfective maintenance.

Adaptive maintenance involves adjusting the application to changes in the environment, e.g., a new release of hardware or the operating system or a new data-base system. In other words, in adaptive maintenance, the need for software changes cannot be attributed to a feature in the software itself, such as the presence of residual errors or the inability to provide some functionality required by the user. Rather, the software must change because the environment in which it works changes.

Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program is known as Perfective maintenance. Here, changes may be due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The requests to perform perfective maintenance may come directly from the software engineer, in order to improve the status of the product on the market, or they may come from the customer, to meet some new requirements.

We will view maintainability as two separate qualities: reparability and evolvability. Software is repairable if it allows the fixing of defects; it is evolvable if it allow changes that enable it to satisfy new requirements.

The distinction between reparability and evolvability is not always clear. For example, if the requirements specifications are vague, then it may not be clear whether we are fixing defect or satisfying a new requirement.

A software system is repairable if it allows the correction of its defects with a limited amount of work. In many engineering products, reparability is a major design goal. In computer hardware engineering, there is a subsection called reparability, availability, and serviceability (RAS).

In other engineering fields, as the cost of a product decreases and the product assumes the status of a commodity, the need for reparability decreases. It is cheaper to replace he whole thing, or at least major arts of it, than to repair it. For example, in early television sets, you could replace a single vacuum tube. Today, a whole board has to be replaced.

In fact, a common technique for achieving reparability in such products is to use standards parts that can be replaced easily. But software parts do not deteriorate. Thus, while the use of standard parts can reduce the cost of software production, the concepts of replaceable parts does not seem to apply to software reparability. Software is also different in this regard because the cost of software. Software is also different in this regard as the cost of software is determined, not by tangible parts, but by human design activity.

Reparability is also affected by the number of parts in a product. For example, it is harder to repair a defect in a monolithic automobile body than If the body were made of several regularly shaped parts. In the latter case, we could replace a single part more easily than the whole body. Of course, if the body consisted of too many parts, it would require too many connections among the parts, leading to the probability that the connections among the parts, leading to the probability that the connections might need repair.

An analogous situation is applied to software. A software product that consists of well-designed modules is much easier to analyze and repair than a monolithic one. Merely, increasing the number of modules, however, does not make a more repairable product. We have to choose the right module structure with the right module interfaces to reduce the need for module interconnections. The right modularization promotes reparability by allowing errors to be confined to few modules, making it easier to locate and remove them. We can examine several modularization techniques, including information hiding and abstract data types.

Reparability can be improved through the use of proper tools. For example, using a high-level language rather than an assembly language leads to better reparability. Also, tools such as debuggers can help in isolating and repairing errors.

A product's reparability affects its reliability. On the other hand, the need for reparability decreases as reliability increases.

1.5 SOFTWARE UPGRADATION

Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modification extremely easy to apply to an implementation. There is, however, a major difference between software modification and modification of other engineering products. In the case of other engineering products, modification starts at the design level and then proceeds to the implementation of the product. For example, if one decides to add a second storey to a house, first, one must do a feasibility study to check whether this can be done safely. Then, one must develop design, based on the original design of the house. Then the design must be approved, ensuring that it does not violate the existing regulations. And, finally, the construction of the new part may be commissioned.

In the case of software, unfortunately, people seldom proceed in such an organized fashion. Although the change might be a radical change in the application too often the implementation is started without doing any feasibility study. Let alone a change in the original design. Still worse, after the change is accomplished, the modification is not even documented a posteriori; i.e., the specifications are not updated to reflect the change. This makes future changes more and more difficult to apply.

On the other hand, successful software products are quite lived. Their first release is the beginning of a long lifetime and each successive release is the next step in the evolution of the system. If the software is designed with care, and if each modification is thought out carefully, then it can evolve gracefully.

☞ **Check Your Progress 2**

1) ________________ is commonly used to refer to the modifications that are made in a software system after its initial release.

2) Software maintenance performed to make a computer program usable in a changed environment is known as ________________.

1.6 SOFTWARE TOOLS AND ENVIRONMENTS

Up to this point we have reviewed the various phases in the software life cycle, and we have seen how to apply some principles of program design and modularity in th actual writing of programs. A natural next step is to consider how some of this work can be streamlined or automated through the use of computers. Any program or other automated aids for the development of software are called tools, and it is appropriate to consider what kind of software tools might increase the productivity of developers or the effectiveness of the final product.

Some of the most obvious areas for computer support of software development arise within the programming phase. At an elementary level, programming requires code to be written, compiled, run, modified, and corrected. This suggests access to an editor that is easy to use, a compiler that runs quickly and identifies errors in an understandable and helpful way, and a means to run programs conveniently with a variety of data sets.

More generally, provisions for editing, compiling, and running programs are part of a general programming environment, which includes all the features available on a particular computer to aid the programming process. A description of such an environment includes statements about what capabilities are available for the programmer and considerations of what the programmer must do to take advantages of these capabilities. For example, on small microcomputers, a programming environment might contain an editor that can be stored on one floppy disk, a compiler on another, and programs themselves on a third disk. If the microcomputer has only one or two disk drives, then programmers must swap disks each time they move from editing to compiling or back.

In contrast, another environment might include an editor and compiler in our package so that programmer can compile and run a program at any time within an editing environment. In this case, the programmer need not spend any extra time moving from editing to compiling environment or running a program.

Beyond this basic integration of programming operator’s more sophisticated environments may assist further with some of these task. For example, some modern syntax-directed editor scan a program as it is begin entered or revised, and this may help automate program indenting and format. These editors may add the keyword End on a following line whenever a programmer types Begin. Similarly, these editors may insert the general form for procedures (including braces, { }, for initial comments and a Begin-End block) whenever the programmer starts a new procedure.

Other programming environments may include incremental compilers, which allow a programmer to work on one part of a program at a time. These compilers recognize what section of code has been revised recently and recompile only these procedures that have been changed. While developing large programs, these capabilities can speed up compilation considerably, since only a few lines may need to be compiled from one test to the next if the overall program contains thousands of lines of code.

Once code is first written, its tracing and debugging can be assisted by the use of a debugger, which controls the execution of the program. With debuggers, program execution may be stopped at designated points, values of variables printed or changed, and the order of procedure calls clarified. Individual procedures or groups of procedures also may be written and tested by themselves, before they are placed within an overall program. A programmer may supply initial values for parameters and then run a module to test the results before these small pieces of code are combined into larger units.

When available, such capabilities may eliminate the need to write separate driver programs for testing.

Other advances in automating the software development process extend beyond the programming phase. Some tools assist specific phases of the process, and other tools automate parts of several phases. This gives rise to general software development environments, which may include a collection of automated aids to assist in the development process.

Both the specification and design phases of software development have the characteristics that data must be collected organized, checked for completeness and consistency, and presented clearly for review. Several computer packages with these common features have been developed to help store information about a problem so that it can be stored and manipulated with relatively little overhead. More sophisticated systems may allow some automated checking of pieces for consistency and some possible omissions. The systems also may format parts of the specifications of designs in appropriate data flow diagrams or structure charts.

In any of these environments, automation can help streamline a particular part of the software development process, and time may be saved in the transition from one phase of the work to the next. Information and decisions within one phase can be checked for completeness and consistency. In addition, this recording of information and decisions through the development of software can assist in the maintenance of the software since maintenance staff will know how software is structured, what each module does, and how potential modifications will affect a module.

1.7 SOFTWARE LIBRARIES AND TOOLKITS

Our discussion of automated aids for software development has focused on the development of new specifications, designs and programs. Another helpful technique involves cataloging existing modules and placing tested modules in libraries. With careful planning, it may not be necessary to spend time and effort on the development of new systems if one can reuse materials that already perform certain tasks.

Any system to take advantage of earlier work relies on at least three conditions. First, we must have a clear idea of what is needed in the current project so that we can identify the appropriate capabilities. Second, there must be a well-organized catalogue of existing software so that we can locate any modules that might available to do pieces of the current project. Third, each existing module must be clearly documented with carefully stated pre and post-conditions.

Although these conditions may seem obvious, it is essential that each of these points is satisfied if existing software is to be reused. For example, the existence of even one extra undocumented, Write statement in a module may change what appears on a user’s terminal.

1.8 SOFTWARE MODULES

Since the purpose of problem solving is to find solutions to specific problems, it is good practice e to survey existing software before planning for new code. For example, many statistical applications can often use general purpose packages, such as SPSS for entering and editing data and running appropriate tests. In such instances, the use of an existing software package may save significant amount of development effort.

In other cases, an existing package may do most of the work need in an applications but some adjustment may be necessary to complete the required task. Here, it may be possible to change or enhance the existing software to meet the current needs.

Alternatively, one might construct a revised flow of data for which most of the work is done with existing software but some additional work needs to be done before or after this main step. Such a data flow is shown at a high level in Figure 1.1. In this figure, the data as initially presented in the problem may not be in a form that can be used by an package. Thus, we may write a (Short) program or pre-processor to modify the form of the raw data so that the revised form will be appropriate for the existing package. Then, after that packages has performed much of the required processing, another (short) post-processor may be needed to complete the job. In this case, solving the initial problem may require the writing of two short programs to handle some details of processing. However, this work still may be substantially shorter and less error prone than ignoring the existing package and writing the entire application.

More generally, much work can be saved by dividing a larger processing task into several subtasks, as shown in *Figure 1.2*. Here, each phases performs some transformations on the data which contribute to the desired results. This approach is particularly powerful if a computing environment allows such steps to be combined in an easy way. For example, the UNIX operating system supports a capability called piping, in which the output of one program can be fed directly into the input of the text. This allows long sequences of programs to be placed together easily. In such an environment, many application may require little programming at all; rather solutions ay be developed by combing various modules in the same way that jigsaw puzzles are completed by putting individual pieces together.

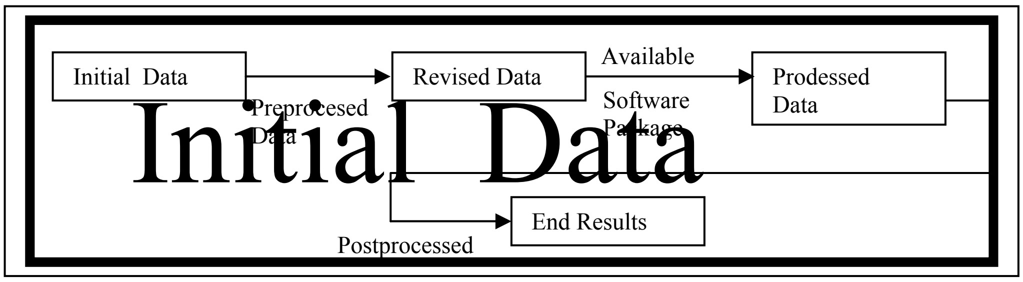

Figure 1.1: Modifying Data to use Existing Software

1.9 REAPPLICATION OF SOFTWARE MODULES

A second way to reduce the amount of code that must be written in any application is to identify tasks that must be done in several phases and write one low-level module to perform the common tasks. As a simple example, in designing a wagon, we might start from the general concept of "wagon". Once we decide that a wagon must be supported in four places, we will not try to invent the wheel four times! In the programming context, we might determine the details of a tasks just once, document pre-conditions and post-conditions for this task.Then, refer to that common task from several places.

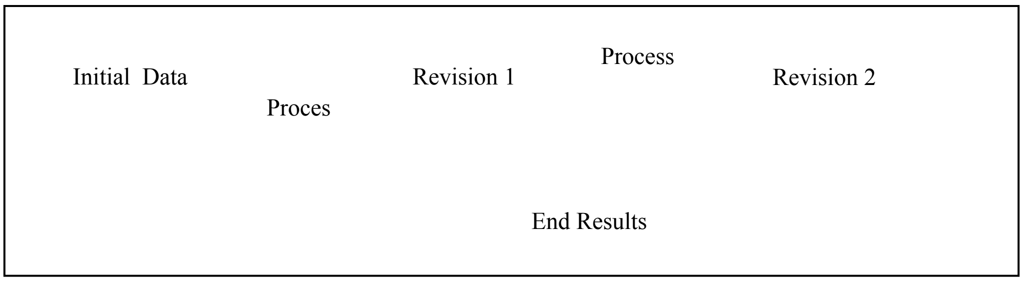

Figure 1.2: Completing a task using several processors

For example, the same type of input or output may be needed in several places within a program, and we may write some procedures to perform standard input and output tasks. In Pascal Language, we regularly rely on the Read and Write procedures in virtually every program. A similar approach can be applied to many programs when we want to read data and place them in a standard form in a particular field of a data record on array. One data entry procedure can ensure that all terminal or file input will be handled in a consistent manner throughout the program.

Between the high level use of all programs and the low-level definition of common procedures for a specific application, we can develop a collection of these commonly used procedures and functions that we can use without additional effort in a variety of application. Such a collection of procedures is known as library, and we had experiences using such libraries.

1.10 DEVELOPMENT TOOLS

There are many development tools for the development environment which include editors. Since software is ultimately a more or les complex collection of documents requirements specifications, module architecture descriptions, programs, etc., editors are a fundamental software development tool.

With respect to the classification of the previous section, we can place editors in different categories as mentioned below.

- They can be either textual or graphical.
- They can follow a formal syntax, op they can be used for writing informal text or drawing free-from pictures. For instance, a general purpose graphical editor such as Apple's MacDraw could be used to produce any kind of diagrams, including formal diagrams such as DFDs or Petri nets, but cannot perform any check on their syntactic correctness. Similarly, we may use a word processor to write programs in any programming language, but the tool cannot help in finding missing keywords, ill formed expressions, undeclared variables, etc. In order to perform such checks one should use tools that are sensitive to the syntax and, if possible, the semantics of the language.
- They can be either monolingual (e.g. and Ada syntax-directed editor) or polylingual (e.g., a general syntax-directed editor that is driven by the specific syntax of a programming language). A conventional world processor is intrinsically polylingual.

They may be used not only to produce, but also to correct or updata documents. Thus, editors should be flexible (e.g., able to be run interactively or in batch) and easy to integrate with other tools. We can integrate an editor with a debugger, in order to support program correction during debugging.

Linkers are tools that are used to combine object code fragments into a larger system. Thus, they can be both monolingual (when they are language specific) or polylingual (when they can accept modules written in different languages).

Basically, a linker binds names appearing in a module to external names exported by other modules. In the case of language-specific linkers, this may also imply a kind of intermodule type checking, depending on the nature of the language. A polylingual linker may perform only binding resolution, leaving all language-specific activities to other tools.

The concept of a linker has broader applicability than just to programming language Typically, if one deals with a modular specification language, then the linker for the language would be able to perform checking and binding across various specification modules.

An interpreter executes actions specified in a formal notation-its input code. At one extreme, an interpreter is the hardware that executes machine code. However, it is possible to interpret even specifications, if they are written in a formal language. In this case, interpreter behaves as a simulator or a prototype of the end product and can help detect mistakes in the specifications at the early stages of the software process.

We have already observed that requirements specifications often occur incrementally hand in hand with the analysis of the application domain. For example, initially, one might decide to specify only the sequence of screen panels through which the end user will interact with the system, leaving out the exact specification of the functions that will be invoked in response to the user input. This decision might be dictated by the fact that, in the application under development, user interfaces are the most critical factor affecting the requirements. Thus, we would decide to check with the end user whether the interaction style we intend to provide corresponds to his/her expectations, before starting any development. This implies that the interpreter of the specifications should be able to generate screen panels and should allow us to display sequences of screen panels in order to demonstrate the interactive sessions with the application. The interpreter should tolerate the incompleteness of specifications. For example, when no functions are provided in response to various commands that might be entered in the fields of the screen panels. In essence, the interpreter operates like a partial prototype, allowing experimentation with the look and feel of the end product.

In other cases, the result of interpreting the requirements is more properly called requirements animation; what we provide on the screen is a view of the dynamic evolution of the model, which corresponds to the physical behaviour of the specified system. For example, one might easily animate a finite state machine that is used to model the evolution of a state changing dynamic system. If the state changing system is a plan controlled by a computer, and a finite state machine displayed on the terminal describes the states entered by the plant as a consequence of commands issued by the computer, then we may achieve animation by blinking the states of the finite state machine as the corresponding state of the plant is entered. The control signals may be simulated by pressing any key on the terminal.

Usually, interpreters operate on actual input values. It is possible, however, to design symbolic interpreters, which operate on symbolic input values. A symbolic execution corresponds to a whole class of executions on input data. Thus, a symbolic interpreter can be a useful verification fool and can be used as an intermediate step in the derivation of test data that causes execution to follow certain paths.

1.10.1 Code Generators

The software development process is a sequence of steps that transform a given problem description called a specification into another description called an implementation. In general, the later description is more formal, more detailed, and of lower level than the former. It is also more efficient. The transformation process eventually end in machine- executable code. As mentioned in the previous section, intermediate steps may also be executable.

Derivation steps may require creativity and may be supported by tools to varying degrees. A simple and fully automatic step is the translation from source code into object code. This is performed by suitable software tools. The transformation may be recorded, and even controlled, by a suitable tool, but the choice of which lower level modules to use to implement a given higher level module is the designer's responsibility and cannot be automated fully.

With reference to the transformation based life cycle model, the optimizer tool is essentially a translator supporting the stepwise transformation of specifications into an implementation. As we discussed, the optimizer is only partially automated. The clerical job of recording the transformation steps is automated in order to support later modifications. We also envisioned the case where the optimizer plays the role of an intelligent assistant. The difficult and critical steps, however, cannot be automated; thus, even is this case, most of the creative tasks are the software engineer's responsibility.

Moving from a formal specification of a module to an implementation may also be viewed as a transformation that involves creative activities such as designing data structure of algorithm. Again, clerical parts of such a transformation can be supported by automatic tools. Examples of generalized code generators are provided by several so-called fourth-generation tools, which automatically generate code for higher level language. Also, reports may be automatically generated from the database definition. In this case, the user can choose among several options to define the report formats.

1.10.2 Debuggers

Debugging may be viewed as a kind of interpreter. In fact, debuggers execute a program with the purpose of helping to localize and fix errors Modern debuggers give the following major capabilities to the user.

- To inspect the execution state in a symbolic way. (Here, “symbolic” means “referring to symbolic identifiers of program object”. But, not that the debugger is a symbolic interpreter).
- To select the execution mode, such as initiating step-by-step execution or setting breakpoints symbolically.
- To select the portion for the execution state by the execution points to inspect without manually modifying the source code. This not only makes debugging simpler, but also avoids the risk of introducing foreign code that one may forget to remove after debugging.
- To check intermediate assertions.
- A debugger can also be used for other reasons than just locating and removing defects from a program. A good symbolic debugger can be used to observe the dynamic behavior of a program. By animating the programs in this way, a debugger can be a useful aid in understanding programs written by another programmer, thus supporting program modifications and reengineering.

☞ Check Your Progress 3

1) UNIX operating system supports a capability called _________ , in which the output of one program can be fed directly into the input of the text.

2) A __________ binds names appearing in a module to external names exported by other modules.

1.11 SUMMARY

Software Reliability may be defined as the ability of software to perform its required functions under stated conditions for a specified period of time. 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. Software maintenance is commonly used to refer to the modifications that are made in a software system after its initial release. Maintenance performed to correct faults in hardware or software is known as Corrective Maintenance. Software maintenance performed to make a computer program usable in a changed environment is known as Adaptive maintenance. Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program is known as Perfective maintenance.

1.12 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Software Reliability

2) SQA

Check Your Progress 2

1) Software Maintenance

2) Adaptive Maintenance

Check Your Progress 3

1) Piping

2) Linker

1.13 FURTHER READINGS

1. *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

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

Reference Websites

http://www.rspa.com
http://www.ieee.org