

## **Software Engineering**

### **Third class**

### **Lecture 15**

#### **15.1 System Testing**

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. The types of system tests that are worthwhile for software-based systems are:

##### **1-Recovery Testing**

Many computer based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

*Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

##### **2-Security Testing**

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration.

*Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

##### **3-Stress Testing**

Stress tests are designed to confront programs with abnormal situations.

*Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,(1) special tests may be designed that

generate ten interrupts per second, when one or two is the average rate, (2) test cases that require maximum memory or other resources are executed. Essentially, the tester attempts to break the program.

#### 4- Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable.

*Performance testing* is designed to test the run-time performance of software within the context of an integrated system.

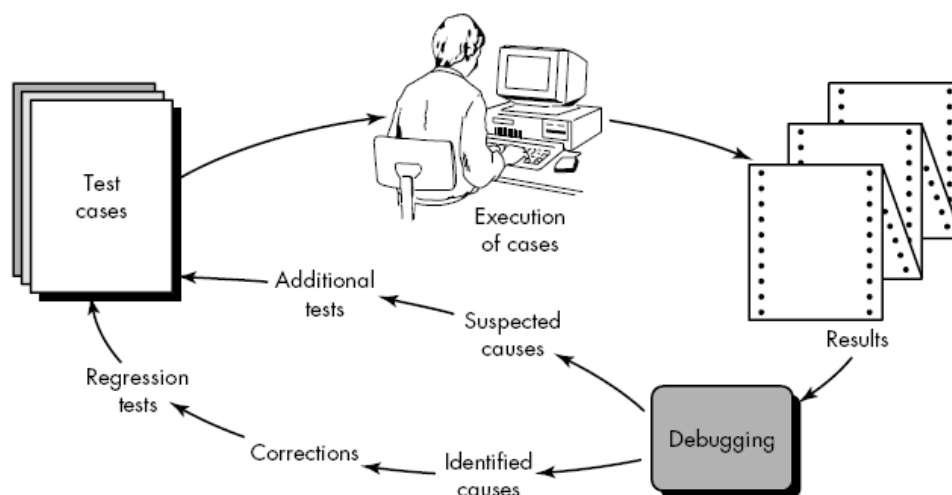
Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

#### 15.2 Debugging

*Debugging* occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. The poorly understood mental process that connects a symptom to a cause is debugging.

##### 15.2.1 The Debugging Process

Debugging is not testing but always occurs as a consequence of testing. Referring to Figure 1,



**Figure 1:** The debugging process

The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

### 15.2.2 Debugging Approaches

Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck.

In general, three categories for debugging approaches may be proposed:

(1) brute force, (2) backtracking, and (3) cause elimination.

The **brute force category** of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time.

**Backtracking** is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found.

Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—**cause elimination**—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis.

Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

### **15.3 Maintaining the System**

#### **15.3.1 The changing system**

System development is complete when the system is operational, that is, when the system is being used by users in an actual production environment. Any work done to change the system after it is in operation is considered to be **maintenance**. Many people think of software system maintenance as they do hardware maintenance: repair or prevention of broken or improperly working parts. However, software maintenance cannot be viewed in the same way.

One goal of software engineering is developing techniques that define a problem exactly, design a system as a solution, and implement a correct and efficient set of programs. And test the system for faults. This goal is similar for hardware developers. Producing a reliable fault free product that works according to specification.

Hardware maintenance in such a system concentrates on replacing parts that wear out or using techniques that prolong the system's life. However, while do constructs do not wear out after 10,000 loops, and semicolons do not fall off the ends of statements. Unlike hardware, software does not degrade or require periodic maintenance.

#### **15.3.2 The nature of Maintenance**

When we develop systems, our main focus is on producing code that implements the requirements and works correctly. At each stage of development, our team continually refers to work produced at earlier stages. The design components are tied to the requirements specification. The code components are cross-referenced and reviewed for compliance with the design, and the tests are based on finding out whether functions and constraints are working according to the requirements and design. Thus development involves looking back in a careful, controlled way.

Maintenance is different. As maintainers, we look back at development products, but also at the present by establishing a working relationship with users and operators to find out how satisfied they are with the way the system works. We look forward, too, to anticipate things that might go wrong, to consider functional changes required by a changing business need, and to consider system changes required by changing

hardware, software, or interfaces. Thus, maintenance has a broader scope, with more to track and control.

### **15.3.3 Maintenance Activities And Roles**

Maintenance activities are similar to those of development: analyzing requirements, evaluating system and program design, writing and reviewing code, testing changes, and updating documents. So the people who perform maintenance—analysts, programmers, and designers—have similar roles. However, because changes often require an intimate knowledge of code's structure and content, programmers play a much larger role in maintenance than they did in development.

Maintenance focuses on four major aspects of system evolution simultaneously:

- 1-maintaining control the day-to-day functions
- 2-maintaining control over system modification.
- 3-perfecting existing acceptable functions.'
- 4-preventing system performance from degrading to unacceptable level.

**Corrective Maintenance.** To control the day-to-day system functions, we on the maintenance team respond to problems resulting from faults. Addressing these problems is known as corrective maintenance. As failures occur, they are brought to our team's attention; we then find the failure's cause and make corrections and changes to requirements, design, code, test suites and documentation, as necessary. Often, the initial repair is temporary; something to keep the system running, but not the best fix. Long range changes may be implemented later to correct more general problems with design or code.

**Adaptive maintenance.** Sometimes a change introduced in one part of the system requires changes to other parts. **Adaptive maintenance** is the implementation of these secondary changes.

For example if a compiler is enhanced by the addition of debugger. We must alter the menus icons, or function key definitions to allow users to choose the debugger option.

### **Perfective Maintenance**

As we maintain a system, we examine documents, design, code and tests, looking for opportunities for improvement. Perfective maintenance involves making changes to improve some aspect of the system, even when the changes are not suggested by faults. Documentation changes to clarify items, test suite changes to improve test coverage, and code and design modifications to enhance readability are all examples of perfective maintenance.

### **Preventive Maintenance**

Similar to perfective maintenance, **preventive maintenance** involves changing some aspect of the system to prevent failures. It may include the addition of type checking, the enhancement of fault handling, or the addition of a "catch-all" statement to a case statement, to make sure the system can handle all possibilities. Preventive maintenance usually results when a programmer or code analyzer finds an actual or potential fault that has not yet become a failure and takes action to correct the fault before damage is done.

#### **15.3.4 Who perform Maintenance**

The team that develops a system is not always used to maintain the system once it is operational. Often, a separate maintenance team is employed to ensure that the system runs properly. There are positive and negative aspects to using separate maintenance team. The development team is familiar with the code, the design, and philosophy behind it, and the system's key functions. If the developers know they are building something that they will maintain, they will build the system in a way that makes maintenance easier.

However, developers sometimes feel so confident in their understanding of the system that they tend not to keep the documentation up to date. Their lack of care in writing and revising documentation may result in the need for more people or resources to tackle a problem. This situation leads to a long interval from the time a problem occurs to the time it is fixed. Many customers will not tolerate a delay.

Often a separate group of analysts, programmers and designers (sometimes including one or two members of the development team) is designed as the maintenance team. A fresh new team may be more objective than the original developers. A separate team may find it easier to distinguish how a system should work from how it does

work. If they know others will work from their documentation, developers tend to be more careful about documentation and programming standards.

#### **15.4 Maintenance Problems**

Maintaining a system is difficult. Because the system is already operational, the maintenance team balances the need for change with the need for keeping a system accessible to users. For example, updating a system may require it to be unavailable to users for several hours. However, if the system is critical to the users' business or operation, there may not be a window of several hours when users can give up the system. For instance, a life support system cannot be disconnected from a patient to maintenance can be performed on the software. The maintenance team must find a way to implement change without inconveniencing users.

##### **1-Staff Problems**

There are many staff and organizational reasons that make maintenance difficult. The staff must act as an intermediary between the problem and its solution, tinkering and tailoring the software to ensure that the solution follows the course of the problem as it changes.

##### **a- Limited understanding**

In addition to balancing used needs with software and hardware needs, the maintenance team deals with the limitations of human understanding. There is a limit to the rate at which a person can study documentation and extract material relevant to the problem being solved. Furthermore, we usually look for more clues than are really necessary for solving a problem. Adding the daily office distractions, we have a prescription for limited productivity.

47 percent of software maintenance effort is devoted to understanding the software to be modified. This high figure is understandable when we consider the number of interfaces that need to be checked whenever a component is changed. For example, if a system has  $m$  components and we need to change  $k$  of them, there are

$$k*(m-k) + k*(k-1)/2$$

Interfaces to be evaluated for impact and correctness. So, even a one-line change to a system component may require hundreds of test to be sure that the change has no direct or indirect effect on another part of the system.

User understanding also present problems. Half of the maintenance programmer's problem derived from users' lack of skill or understanding. For example, if users do not understand how the system functions, they may provide maintainers with incomplete or misleading data when reporting a problem's effects.

These results illustrate the importance of clear and complete documentation and training. The maintenance team also needs good "people skills", there are a variety of work styles. The maintenance team must understand how people with different styles think and work and team members must be flexible when communicating.

### **b- Management Priorities**

The maintenance team weighs the desires of the customer management against the system's needs. Management priorities often override technical ones: managers sometimes view maintaining and enhancing as more important than building new applications. In other words, companies must sometimes focus on business as usual. Instead of investigating new alternatives. But as management encourages maintainers to repair an old system, users are clamoring for new functions or a new system. Similarly, the rush to get product to market may encourage us. As either developers or maintainers, to implement a quick, intelligent, poorly tested change, rather than take the time to follow good software engineering practice. The result is a patched product that is difficult to understand and repair later.

### **c- Morale**

studies indicate that 11.9 percent of the problems during maintenance result from low morale and productivities. A major reason for low morale is the second class status often accorded the maintenance team. Programmers sometimes think that it takes more skill to design and develop a system than to keep it running. However, maintenance programmers handle problems that developers never seen. Maintainers are skilled not only on writing code but also in working with users, in anticipating change, and in sleuthing. Great skill and perseverance are required to track a problem to its source, to understand the inner workings of a large system, and to modify that system's structure, code and documentation.

Some groups rotate programmers among several maintenance and development projects to give the programmers a chance to do a variety of things. This rotation helps avoid the perceived stigma of maintenance programming. However, programmers are often asked to work on several projects concurrently. Demands on a programmer's time result in conflicting priorities. During maintenance, 8 percent of the problems result from a programmer's being pulled in too many directions at once and thus being unable to concentrate on one problem long enough to solve it.

## **2-Technical Problems**

Technical problems also affect maintenance productivity. Sometimes, they are a legacy of what developers and maintainers have done before. At other times, they result from particular paradigms or processes that have been adopted for the implementation.

### **a- Artifacts and Paradigms**

If the design's logic is not obvious, the team may not easily determine whether the design can handle proposed changes. A flawed or inflexible design can require extra time for understanding, changing and testing.

In general, inadequate design specifications and low-quality programs and documentation account for almost 10 percent of maintenance effort. A similar amount of effort is dedicated to hardware requirements: getting adequate storage and processing time. Problems also arise when hardware, software, or data are unreliable.

### **b- Testing difficulties**

Testing can be a problem when finding time to test is difficult. For example, an airline reservations system cannot be available around the clock. It may be difficult to convince users to give up the system for two hours of testing. When a system performs a critical function such as air traffic control or patient monitoring, it may be impossible to bring it offline to test. In these cases, tests are often run on duplicate systems: then, tested changes are transferred to the production system.

In addition to time availability problems, there may not be good or appropriate test data available for testing the changes made.

Most important, it is not always easy for testers to predicate the effects of design or code change and to prepare for them. This unpredictability exists especially when

different members of the maintenance team are working on different problems at the same time.

### **3- The Need to compromise**

The maintenance team is always involved in balancing one set of goals with another. Conflict arises between system availability for users and implementation of modification, corrections, and enhancements. Because failures occur at unpredictable times, the maintenance staff is constantly aware of this conflict.

For computing professionals, another conflict arises whenever change is necessary. Principles of software engineering compete with expediency and cost. Often, a problem may be fixed in one of two ways: a quick but inelegant solution that works but does not fit the system's design or coding strategy, or a more involved but elegant way that is consistent with the guiding principles used to generate the rest of the system. Programmers may be forced to compromise elegance and design principles because a change is needed immediately.

### **4- Maintenance cost**

All the problems of maintaining a system contributes to the high cost of software maintenance.

### **Factors Affecting Effort**

In addition to the problems already disused, there are many other factors that contribute to the effort needed to maintain a system. These factors can include the following:

- **Application type:** Real-time and highly synchronized systems are more difficult to change than those where timing is not essential to prepare functioning. We must take great care to ensure that a change to one component does not affect the timing of the others.
- **System novelty:** when a system implements a new application or a new way of performing common functions, the maintainers cannot easily rely on their experience and understanding to find and fix faults. it takes longer to understand the design, to locate the source of problems, and to test the

corrected code. In many cases, additional test data must be generated when old test data do not exist

- **Turnover and maintenance staff availability:** Substantial time is required to learn enough about a system to understand and change it. Maintenance effort suffers if team members are routinely rotated to other groups. If staff members leave the organization to work on another project, or if staff members are expected to maintain several different products at the same time.
- **System life span:** a system that is designed to last many years is likely to require more maintenance than one whose life is short. Quick correction and lack of care in updating documentation are probably acceptable for a system with short life: such habits can be deadly on a long term project, where they make it more difficult for other team members to make subsequent changes.
- **Depending on changing environment.** A system dependent on its hardware's characteristics is likely to require many changes if the hardware is modified or replaced.
- **Hardware characteristics:** Unreliable hardware components or unreliable vendor support may make it more difficult to track a problem to its source.
- **Design quality:** if the system is not composed of independent, cohesive components, finding and fixing the source of a problem may be compounded by changes creating unanticipated effects in other components.
- **Code quality:** if the code is unstructured or does not implement the guiding principles of its architecture, it may be difficult to locate faults. In addition, the language itself may make it difficult to find and fix faults: high-level languages often enhance maintainability.
- **Documentation quality:** undocumented design or code makes the search for a problem's solution almost impossible. Similarly, if the documentation is difficult to read or even incorrect, the maintainers can be thrown off track.
- **Testing quality:** if tests are run with incomplete data or do not anticipate the repercussions of a change, the modification and enhancements can generate other system problems.