Software Engineering 1

Title: Software Testing **Module No: 7**

I. INTRODUCTION

Once the software application was being design and being coded, software testing is an important process of software development. It will help determine possible bugs or errors in the software application. It will help determine how to address this kind of bugs or defect that can be seen in a particular code of the software system.

We need to understand how bugs, errors or defect affect the entire software application in order to apply necessary intervention and fix this bugs, error or defect. It is also necessary to have a proper checklist in order to evaluate properly the entire software application.

II. LEARNING OBJECTIVES

After studying this module, you should be able to:

- Identify and understand different practices and guidelines in good programming
- Identify and understand software verification and validation
- Understand the difference between testing and development
- Understand unit testing
- Understand the difference between inspection and testing
- Understand and Identify debugging

III. TOPICS AND KEY CONCEPTS

A. Good Programming Practices and Guidelines

Maintainable Code

In most cases, maintainability is the most desirable quality of a software artifact. Code is no exception. Good software ought to have code that is easy to maintain. Fowler says, “Any fool can write code that computers can understand, good

programmers write code that humans can understand." That is, it is not important to write code that works, it is important to write code that works and is easy to understand so that it can be maintained. The three basic principles that guide maintainability are: simplicity, clarity, and generality or flexibility. The software will be easy to maintain if it is easy to understand and easy to enhance. Simplicity and clarity help in making the code easier to understand while flexibility facilitates easy enhancement to the software.

Self Documenting Code

From a maintenance perspective, what we need is what is called self documenting code. A self documenting code is a code that explains itself without the need of comments and extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc. That is, the meaning of the code should be evident just by reading the code without having to refer to information present outside this code.

The question is: how can we write code that is self-documenting?

There are a number of attributes that contributes towards making the program self documented. These include, the size of each function, choice of variable and other identifier names, style of writing expressions, structure of programming statements, comments, modularity, and issues relating to performance and portability.

Function Size

The size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Ideally speaking, a function should not be larger than 20 lines of code and in any case should not exceed one page in length. From where did I get this number 20 and one page? The number 20 is approximately the lines of code that fit on a computer screen and one page of course refers to one printed page. The idea behind these heuristics is that when one is reading a function, one should not need to go back and forth from one screen to the other or from one page to the other and the entire context should be present on one page or on one screen.

Coding Style Guide

Consistency plays a very important role in making it self-documenting. A consistently written code is easier to understand and follow. A coding style guide is aimed at improving the coding process and to implement the concept of standardized and relatively uniform code throughout the application or project. As a number of

programmers participate in developing a large piece of code, it is important that a consistent style is adopted and used by all. Therefore, each organization should develop a style guide to be adopted by its entire team.

This coding style guide emphasizes on C++ and Java but the concepts are applicable to other languages as well.

Naming Conventions

Hungarian Notation was first discussed by Charles Simonyi of Microsoft. It is a variable naming convention that includes information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc). Every company and programmer seems to have their own flavor of Hungarian Notation. The advantage of Hungarian notation is that by just looking at the variable name, one gets all the information needed about that variable.

Bicapitalization or camel case (frequently written CamelCase) is the practice of writing compound words or phrases where the terms are joined without spaces, and every term is capitalized. The name comes from a supposed resemblance between the bumpy outline of the compound word and the humps of a camel. CamelCase is now the official convention for file names and identifiers in the Java Programming Language.

In our style guide, we will be using a naming convention where Hungarian Notation is mixed with CamelCase.

B. Software Verification and Validation

Software Testing

To understand the concept of software testing correctly, we need to understand a few related concepts.

Software verification and validation

Verification and validation are the processes in which we check a product against its specifications and the expectations of the users who will be using it.

According to a known software engineering expert Berry Boehm, verification and validation are

Verification

- Does the product meet system specifications?
- Have you built the product right?

Validation

- Does the product meet user expectations?
- Have you built the right product?

It is possible that a software application may fulfill its specifications but it may deviate from users expectations or their desired behavior. That means, software is verified but not validated. How is it possible? It is possible because during the requirements engineering phase, user needs might not have been captured precisely or the analyst might have missed a major stakeholder in the analysis. Therefore, it is important to verify as well as validate the software product.

Defect

The second major and a very important concept is Defect. A defect is a variance from a desired product attribute. These attributes may involve system specifications well as user expectation. Anything that may cause customer dissatisfaction, is a defect. Whether these defects are in system specifications or in the software products, it is essential to point these out and fix.

Therefore software defect is that phenomenon in which software deviates from its expected behavior. This is non-compliance from the expected behavior with respect to written specifications or the stakeholder needs.

Software and Defect

Software and defects go side by side in the software development life cycle. According to a famous saying by Haliburton, Death and taxes are inevitable. According to Kernighan: Death, taxes, and bugs are the only certainties in the life of a programmer. Software and defects cannot be separated, however, it is important to learn how discovering defects at an appropriate stage improves the software quality. Therefore, software application needs to be verified as well as validated for a successful deployment.

Software Testing

With these concepts, we are in a position to define software testing. Software testing is the process of examining the software product against its requirements. Thus it is a process that involves verification of product with respect to its written requirements and conformance of requirements with user needs. From another perspective, software testing is the process of executing software product on test data and examining its output vis-àvis the documented behavior.

Software testing objective

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- This is because exhaustive testing of software is not possible – it is simply too expansive and needs virtually infinite resources.

Successful Test

From the following sayings, a successful test can be defined "If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none" – Myers 1979. "A test is said to be successful if it discovers an error" – doctor's analogy.

The success of a test depends upon the ability to discover a bug not in the ability to prove that the software does not have one. As, it is impossible to check all the different scenarios of a software application, however, we can apply techniques that can discover potential bugs from the application. Thus a test that helps in discovering a bug is a successful test. In software testing phase, our emphasis is on discovering all the major bugs that can be identified by running certain test scenarios. However it is important to keep in mind that testing activity has certain limitations.

Test Cases and Test Data

In order to test a software application, it is necessary to generate test cases and test data which is used in the application. Test cases correspond to application functionality such that the tester writes down steps which should be followed to achieve certain functionality. Thus a test case involves

- Input and output specification plus a statement of the function under test.
- Steps to perform the function
- Expected results that the software application produces

However, test data includes inputs that have been devised to test the system.

C. Testing vs. development

Testing is an intellectually demanding activity and has a lifecycle parallel to software development. A common misperception about testing is that it is not a challenging activity. It should be noted here that the testing demands grip over the domain and application functionality as is required from an analyst or a designer who has to develop the application from requirements. As without having an in-depth knowledge about the system and the requirements from users, a tester cannot write test cases that can verify and validate software application with respect to documented specifications and user needs. Writing test cases and generating test data are processes that demand scenario building capabilities. These activities essentially require destructive instincts in a tester for the purpose of breaking system to discover loopholes into its functionality.

At the time when these two activities are being performed, merely initial design of the application is completed. Therefore, tester uses his/her imagination to come up with use patterns of the application that can help him/her in describing exact steps that should be executed in order to test a particular functionality. Moreover, tester needs to figure out loose points in the system from where he/she can discover defects. All these activities are highly imaginative and a tester is supposed to possess above average (if not excellent) analytical skills.

We shall explain the testing activities parallel to development activities with the help of the following diagram

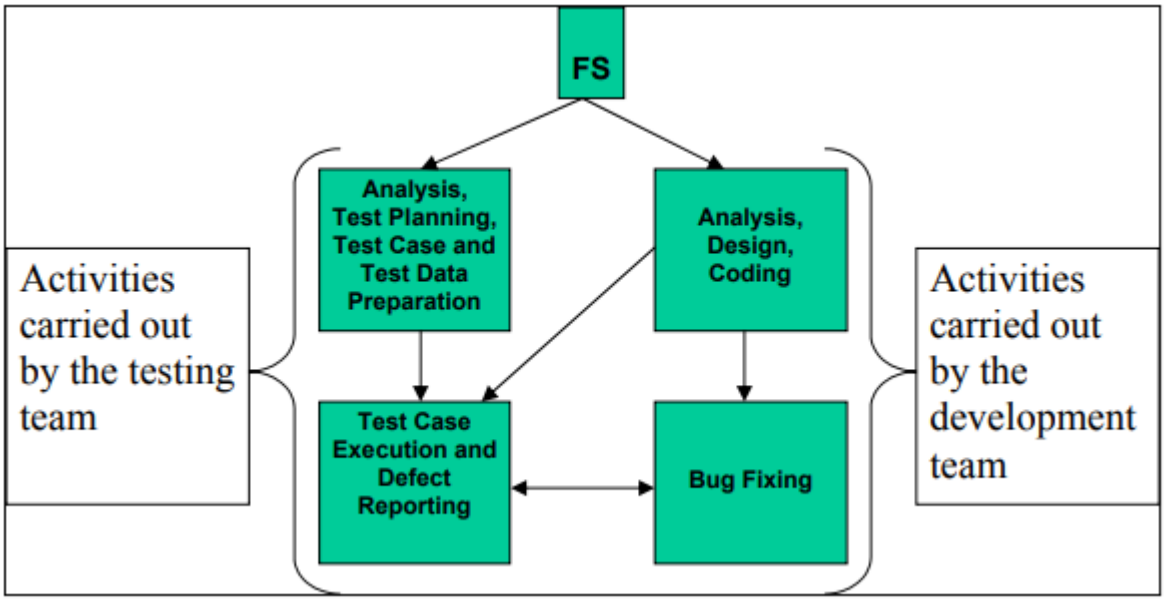

FIGURE 1.0 TESTING ACTIVITIES PARALLEL TO DEVELOPMENT ACTIVITIES

Description

- Functional specification document is the starting point, base document for both testing and the development
- Right side boxes describe the development, whereas, left side boxes explain the testing process
- Development team is involved into the analysis, design and coding activities.
- Whereas, testing team too is busy in analysis of requirements, for test planning, test cases and test data generation.
- System comes into testing after development is completed.
- Test cases are executed with test data and actual results (application behavior) are compared with the expected results,
- Upon discovering defects, tester generates the bug report and sends it to the development team for fixing.
- Development team runs the scenario as described in the bug report and try to reproduce the defect.
- If the defect is reproduced in the development environment, the development team identifies the root cause, fixes it and sends the patch to the testing team along with a bug resolution report.
- Testing team incorporates the fix (checking in), runs the same test case/scenario again and verifies the fix.
- If problem does not appear again testing team closes down the defect, otherwise, it is reported again.

The Developer and Tester

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

Scenarios missed or misunderstood during development analysis would never be tested correctly because the corresponding test cases would either be missing or would be incorrect.

The left side column is related to the development, and the right side describes the testing. Development is a creative process as developers have to build the system, whereas, testing is a destructive activity as the goal of a tester is to break the system to discover the defects. Objective of development is to show that the program works, objective of testing is to show that program does not work. However, FS is the base document for both of these activities.

Tester analyzes FS with respect to testing the system whereas; developer analyzes FS with respect to designing and coding the system. If developer does not understand the FS correctly then he cannot implement and test it right. Thus if the same person who has developed a system, tests it, chances of carrying the same misunderstanding in testing will be very high. Therefore, an independent testing can only prove his understanding wrong. Therefore, it is highly recommended that developer should not try to test his/her own work.

Usefulness of testing

Objective of testing is to discover and fix as many errors as possible before the software is put to use. That is before it is shipped to the client and the client runs it for acceptance. In software development organizations, a rift exists between the development and the testing teams. Often developers are found questioning about the significance or even need to have the testing resources in the project teams. Whoever doubts on the usefulness of the testing team should understand what could happen if the application is delivered to client without testing? At the best, the client may ask to fix all the defects (free of cost) he would discover during the acceptance testing. At the worst, probably he would sue the development firm for damages. However, in practice, clients are often seen complaining about the deliverables and a couple of defected deliverables are sufficient for breaking the relations next to the cancellation of contract.

Therefore, it should be well preserved among the community of developers that testers are essential rather inevitable. A good tester has a knack of smelling errors – just like auditors and it is for the good of the organization not to harm it.

Testing and software phases

With the help of the following diagram we shall explain different phases of testing Software development process diagram.

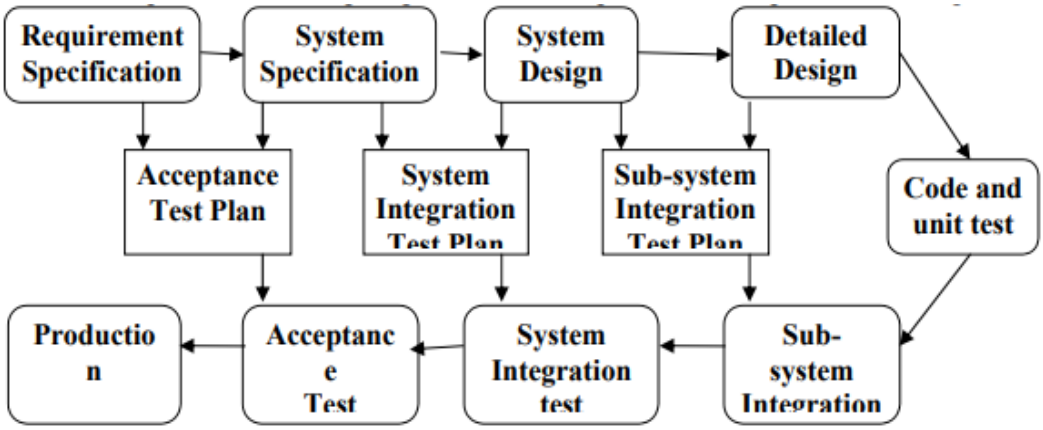

FIGURE 2.0 PHASES OF TESTING SOFTWARE DEVELOPMENT PROCESS

Description of testing phases

- Unit testing – testing individual components independent of other components.
- Module testing – testing a collection of dependent components – a module encapsulates related components so it can be tested independently.
- Subsystem testing – testing of collection of modules to discover interfacing problems among interacting modules.
- System testing – after integrating subsystems into a system – testing this system as a whole.
- Acceptance test – validation against user expectations. Usually it is done at the client premises.
- Alpha testing – acceptance testing for customized projects, in-house testing for products.
- Beta testing – field testing of product with potential customers who agree to use it and report problem before system is released for general use

In the following two types of testing activities are discussed.

Black box testing

In this type of testing, a component or system is treated as a black box and it is tested for the required behavior. This type of testing is not concerned with how the inputs are transformed into outputs. As the system's internal implementation details are not visible to the tester. He gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

Structural testing (white box)

As opposed to black box testing, in structural or white box testing we look inside the system and evaluate what it consists of and how is it implemented. The inner of a system consists of design, structure of code and its documentation etc. Therefore, in white box testing we analyze these internal structures of the program and devise test cases that can test these structures.

Effective testing

The objective of testing is to discover the maximum number of defects with a minimum number of resources before the system is delivered to the next stage. Now the question arises here how to increase the probability of finding a defect?

As, good testing involves much more than just running the program a few times to see whether it works or not. A good tester carries out a thorough analysis of the program to devise test cases that can be used to test the system systematically and effectively. Problem here is how to develop a representative set of test cases that could test a complete program. That is, selection of a few test cases from a huge set of possibilities. What should be the sets of inputs that should be used to test the system effectively and efficiently?

String Equal Example

- For how many equal strings do I have to test to be in the comfortable zone?
- For how many unequal strings do I have to test to be in the comfortable zone?
- When should I say that further testing is unlikely to discover another error? Testing types

To answer these questions, we divide a problem domain in different classes. These are called Equivalence Classes.

Equivalence Classes or Equivalence Partitioning

Two tests are considered to be equivalent if it is believed that:

- if one discovers a defect, the other probably will too, and
- if one does not discover a defect, the other probably won't either.

Equivalence classes help you in designing test cases to test the system effectively and efficiently. One should have reasons to believe that the test cases are equivalent. As for this purpose, one would need to understand the system and see in how many partitions it can be divided. These partitions should be devised such that a clear distinction should be marked. Test cases written for one partition should not yield the same results when run for the second partition as otherwise these two partitions should become one. However, finding equivalence classes is a subjective process, as two people analyzing a program will probably come up with different sets of equivalence classes

Equivalence partitioning guidelines

- Organize your equivalence classes. Write them in some order, use some template, sequence, or group them based on their similarities or distinctions. These partitions can be hierarchical or organized in any other manner.
- Boundary conditions: determine boundary conditions. For example, adding in an empty linked list, adding after the last element, adding before the first element, etc.
- You should not forget invalid inputs that a user can give to a system. For example, widgets on a GUI, numeric instead of alphabets, etc.

D. Unit testing

A software program is made up of units that include procedures, functions, classes etc. The unit testing process involves the developer in testing of these units. Unit testing is roughly equivalent to chip-level testing for hardware in which each chip is tested thoroughly after manufacturing. Similarly, unit testing is done to each module, in isolation, to verify its behaviour. Typically the unit test will establish some sort of artificial environment and then invoke routines in the module being tested. It then checks the results returned against either some known value or against the results

from previous runs of the same test (regression testing). When the modules are assembled we can use the same tests to test the system as a whole.

Software should be tested more like hardware, with

- ✓ Built-in self testing: such that each unit can be tested independently
- ✓ Internal diagnostics: diagnostics for program units should be defined.
- ✓ Test harness

The emphasis is on built in testability of the program units from the very beginning where each piece should be tested thoroughly before trying to wire them together.

Unit Testing Principles

- In unit testing, developers test their own code units (modules, classes, etc.) during implementation.
- Normal and boundary inputs against expected results are tested.
- Thus unit testing is a great way to test an API.

Quantitative Benefits

- **Repeatable:** Unit test cases can be repeated to verify that no unintended side effects have occurred due to some modification in the code.
- **Bounded:** Narrow focus simplifies finding and fixing defects.
- **Cheaper:** Find and fix defects early

Qualitative Benefits

- **Assessment-oriented:** Writing the unit test forces us to deal with design issues - cohesion, coupling.
- **Confidence-building:** We know what works at an early stage. Also easier to change when it's easy to retest.

Unit Testing Tips

Unit test should be conveniently located

- For small projects you can imbed the unit test for a module in the module itself
- For larger projects you should keep the tests in the package directory or a /test subdirectory of the package

By making the code accessible to developers you provide them with:

- Examples of how to use all the functionality of your module
- A means to build regression tests to validate any future changes to the code

You can use the main routine with conditional compilation to run your unit tests.

Defect origination

In inspections the emphasis is on early detection and fixing of defects from the program. Following are the points in a development life cycle where defects enter into the program.

- Requirements
- Design
- Coding
- User documentation
- Testing itself can cause defects due to bad fixes
- Change requests at the maintenance or initial usage time

It is important to identify defects and fix them as near to their point of origination as possible.

E. Inspection versus Testing

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the verification and validation process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc. Inspection does not require execution of program and they maybe used before implementation. Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required. For inspections, checklists are prepared that contain information regarding defects. Reuse domain and programming knowledge of the viewers likely to help in preparing these checklists. Inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections may be applied to any representation of the system (requirements, design, test data, etc.) Thus inspections are a very effective technique for discovering errors in a software program.

Inspection pre-conditions

A precise specification must be available before inspections. Team members must be familiar with the organization standards. In addition to it, syntactically correct code must be available to the inspectors. Inspectors should prepare a checklist that can help them during the inspection process.

Inspection checklists.

Checklist of common errors in a program should be developed and used to drive the inspection process. These error checklists are programming language dependent such that the inspector has to analyze major constructs of the programming language and develop checklists to verify code that is written using these checklists. For example, in a language of weak type checking, one can expect a number of peculiarities in code that should be verified. So the corresponding checklist can be larger. Other example of programming language dependant defects are defects in variable initialization, constant naming, loop termination, array bounds, etc.

Inspection Checklist

Following is an example of an inspection checklist.

Exception management faults	• Have all possible error conditions been taken into account?
Fault Class	• Inspection Check

Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the lower bound of arrays be 0, 1, or something else? • Should the upper bound of arrays be size or size -1? • If character strings are used, is a delimiter explicitly assigned?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for?
Input/Output faults	• Are all input variables used? • Are all output variables assigned a value before they are output?
Interface faults	• Do all function and procedure calls have correct number of parameters? • Do formal and actual parameters types match? • Are the parameters in right order? • If components access shared memory, do they have the same model of shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly assigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly de-allocated after it is no longer required?

The checklist mentioned above, a number of fault classes have been specified and their corresponding inspection checks are described in the column at the right side. This type of checklist helps an inspector to look for specific defects in the program. These inspection checks are the outcomes of experience that the inspector has gained out of developing or testing similar programs.

Static analyzers

Static analyzers are software tools for source text processing. They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the verification and validation team. These tools are very effective as an aid to inspections. But these are supplement to but not a replacement for inspections.

Checklist for static analysis

Data faults	• variable used before initialization • variable declared but never used • variables assigned twice but never used between assignments • possible array bound violations • undeclared variables
Control faults	• unreachable code • unconditional branches into loops
Input/Output faults	• variable output twice with no intervening assignment
Storage Management fault	• unassigned pointers • pointer arithmetic

F. Debugging

Debugging

As Benjamin Franklin said, "in this world, nothing is certain but death and taxes." If you are in the software development business, however, you can amend that statement. Nothing in life is certain except death, taxes, and software bugs. If you cryogenically freeze yourself, you can delay death indefinitely. If you move to a country with no income tax, you can avoid paying taxes by not buying anything. If you develop software, however, no remedy known to mankind can save you from the horror of software bugs.

What is a Bug?

We call them by many names: software defects, software bugs, software problems, and even software "features." Whatever you want to call them, they are things the software does that it is not supposed to do (or, alternatively, something the software doesn't do that it is supposed to). Software bugs range from program crashes to returning incorrect information to having garbled displays.

A Brief History of Debugging

It would appear that as long as there have been computers, there have been computer bugs. However, this is not exactly true. Even though the earliest known computer programs contained errors, they were not, at that time, referred to as "bugs." It took a lady named Admiral Grace Hopper to actually coin the term "bug."

After graduating from Vassar in 1928, she went to Yale to receive her master's degree in mathematics. After graduating from Yale, she worked at the university as a mathematics professor. Leaving Yale in 1943, with the onset of World War II, Mrs. Hopper decided to work for the Navy. Mrs. Hopper's first assignment was under Commander Howard Aiken at Howard University, working at the Bureau of Ordinance Computation. She was a programmer on the Mark II, the world's first automatically sequenced digital computer. The Mark II was used to determine shooting angles for the big guns in varying weather conditions during wartime.

It was during her term with the Mark II that Hopper was credited with coining the term "bug" for a computer problem. The first "bug" was actually a moth, which flew through an open window and into one of the Mark II's relays. At that time, physical relays were used in computers, unlike digital components we use today. The moth shorted out across two contacts, temporarily shutting down the system. The moth would later be removed (de-bugged?) and pasted into the logbook of the project. From that point on, if her team was not producing numbers or working on the code, they claimed to be "debugging" the system.

From that auspicious beginning, computer debugging developed into something of a hitor-miss procedure for quite a few years. Early debugging efforts mostly centered around either data dumps of the system or used output devices, such as printers and display lights, to indicate when an error occurred. Programmers would then step through the code line by line until they could determine the location of the problem. The next step in the evolution of debugging came with the advent of command-line debuggers. These simple programs were an amazing step forward for the programmers. Although difficult to use, even at the time, these programs

represented the first real attempt to turn debugging from a hit-or-miss proposition into a reproducible process.

Once the debugger became a part of the programmer's arsenal, the software world began to consider other ways that programs could be more easily debugged. Software projects starting getting bigger, and the same techniques that worked well for small projects no longer worked when the program reached a certain size.

Importance of Debugging

As we mentioned earlier in this course, one of the prime objectives of software engineering is to develop cost effective software applications. According to a survey, when a software application is in the maintenance phase, 20% of its lifecycle cost is attributed towards the defects which are found in the software application after installation. Please bear in mind that the maintenance is the phase in which 2/3rd of the overall software cost incurs. Therefore, 20% of the 2/3rd cost is again a huge cost and we need to understand why this much cost and effort is incurred. In fact, when a software application is installed and being used, any peculiarity in it can cost a lot of direct and indirect damages to the organization. A system downtime is the period in which tremendous pressure is on developers end to fix the problem and make the system running again. In these moments, every second costs hugs losses to the organization and it becomes vital to find out the bug in the software application and fix it. Debugging techniques are the only mechanism to reach at the code that is malfunctioning. In the following subsection, we shall discuss an incident that took place in 1990 and see how much loss the company had to suffer due to a mere bug in the software application.

Problem at AT&T

In the telecommunications industry, loss of service is known as an outage. For most of us, when as outage occurs, we lose our telephone service; we cannot make calls and we cannot receive calls. Outages are accepted and expected hazards in the industry.

On January 15. 1990, AT&T had a US wide telephone system outage that lasted for nine hours. The cause was due to a program error in the software that was meant to make the system more efficient. Eight years later, on April 13, 1998, AT&T suffered another massive failure in its frame-relay network, which affected ATM

machines, credit card transactions and other business data services. This failure lasted 26 hours. Again, the bug was introduced during a software upgrade.

AT&T statement about the Problem

"We believe that the software design, development, and testing processes we use are based on solid, quality foundations. All future releases of software will continue to be religiously tested. We will use the experience we've gained through this problem to further improve our procedures."

We do not believe we can fault AT&T's software development process for the 1990 outage, and we have no reason to believe that AT&T did not rigorously test its software update. In hindsight, it is easy to say that if the developers had only tested the software, they would have seen the bug. Or that if they had performed a code inspection, they would have found the defect. Code inspection might have helped in this case. However, the only way the code inspection could have uncovered this bug is if another engineer saw this particular line of code and asked the original programmer if that was his or her intension. The only reason that the code reviewers might have asked this question is if they were familiar with the specifications for this particular code block.

Art and Science of Debugging

Debugging is taken as an art but in fact it is a scientific process. As people learn about different defect types and come across situations in which they have to debug the code, they develop certain heuristics. Next time they come across a similar situation, they apply those heuristics and solve the problem in lesser time and with a lesser effort. While discussing the debugging process we discuss the phenomenon of "you miss the obvious". When a person writes a code, he develops certain impression about that code. One can term this impression as a personal bias that the developer builds towards his creation the "code" and when he has to check this code, he can potentially miss out obvious mistakes due to this impression or bias. Therefore, it is strongly recommended that in order to reach to a defect in the code, one needs "another pair of eyes". That is, start discovering the defect by applying your own heuristics and if you could reach to the problem, fine, otherwise ask a companion to help you in this process.

IV. TEACHING AND LEARNING MATERIALS RESOURCES

- PC Computer || Laptop || Smartphone
- Internet Connection
- Browsers
- Any available Programming Software
- GC-LAMP
- Google Classroom
- Google Meet
- Facebook Group
- Facebook Messenger
- For online activity sites:
    - ✓ https://www.blogger.com/
    - ✓ https://www.wix.com/

V. LEARNING TASKS

A. ENGAGE

Activity 1: Blogging

A blog is a discussion or informational website published on the World Wide Web consisting of discrete, often informal diary-style text entries. Posts are typically displayed in reverse chronological order, so that the most recent post appears first, at the top of the web page.

Materials Needed: PC/Laptop/Smart phone, Internet Connection and Browser

Instruction: Based on your own understanding, kindly define the following terminologies:

A. SOFTWARE TESTING
B. UNIT TESTING
C. INSPECTION CHECKLIST
D. MAINTAINABLE CODE
E. DEBUGGING

Your answer will be in a form of a blog.
Kindly create your own title for each post.
Each post must contain at least 2 to 3 images.
Continue doing on your previous Blog using the following online sites:
https://www.blogger.com/
or
https://www.wix.com/

Rubric:

Completed the activities and understood the topic based on the given answer	Outstanding 50 points	Very Good 40 points	Good 30 points	Fair 20 points	No Work Output

Activity 2: Group Work (PSEUDOCODE)

Note: This is in preparation for your Final Project Output.

a. Collate the pseudo code you used in your project system.

b. Save it in a document file with proper labeling or simply put a comment section for what exactly a particular code will do.

Rubric:

Completed the activities and understood the topic based on the given answer	Outstanding 50 points	Very Good 40 points	Good 30 points	Fair 20 points	No Work Output

B. EXPLORE & EXPLAIN

Answer the following questions:

1. What is the difference between testing and development?

_____
_____
_____
_____
_____
_____
_____

2. How can you explain the art and science of debugging?

3. What is the difference between inspection and testing?

4. What is a defect?

5. What is the difference between a developer and a tester?

Rubric:

Each correct answer will be given 10 points. Total score = 50 points	Question 1	Question 2	Question 3	Question 4	Question 5	Total Score

C. ELABORATE & EVALUATION

Answer the following questions: **Identification**

_____**1.** Checklist of common errors in a program should be developed and used to drive the inspection process.

_____**2.** Whatever you want to call them, they are things the software does that it is not supposed to do.

_____**3.** Software tools for source text processing.

_____**4.** A code that explains itself without the need of comments and extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc.

_____**5.** Go side by side in the software development life cycle.

_____**6.** This type of testing is not concerned with how the inputs are transformed into outputs.

_____**7.** Correspond to application functionality such that the tester writes down steps which should be followed to achieve certain functionality.

_____**8.** Is roughly equivalent to chip-level testing for hardware in which each chip is tested thoroughly after manufacturing.

_____**9.** In inspections the emphasis is on early detection and fixing of defects from the program.

_____**10.** An intellectually demanding activity and has a lifecycle parallel to software development.

Rubrics:

Each correct answer will be given 5 points. Total score = 50 points	Question 1	Question 2	Question 3	Question 4	Question 5	Total Score
	Question 6	Question 7	Question 8	Question 9	Question 10	Total Score

VI. REFERENCES

- https://www.genrica.com/vustuff/CS504/CS504_handouts_1_45.pdf
- https://vulms.va.com.pk/discussion/60/cs504-handout-and-other-help-data-download
- https://vulms.va.com.pk/categories/cs504-software-engineering-i
- https://vulms.vu.edu.pk/Courses/CS504/Downloads/CS504%20Updated%20Handouts.pdf
- https://en.wikipedia.org/wiki/Blog
- https://www.blogger.com/about/?r=1-null_user
- https://www.wix.com/html5bing/hiker-blog?utm_source=bing&utm_medium=cpc&utm_campaign=ms_en_e_1_NEW^bl_blogging_rest&experiment_id=blogging^be^79714673617818^blogging&msclkid=983ab99d6f3e1cb92c8de6b674948445