Debugging

Dr. Satinder Bal Gupta
ALL YOUR BUG ARE BELONG TO ME!
Presentation Outline

- Debugging
- The Debugging Process
- Difference between Testing & Debugging
- Bug Life Cycle
- Bug Management
- Reporting
- Bug/Defect Types
- Defect Report
- Methods of Debugging
- General Guidelines of Debugging
- Error Analysis
Debugging

• Debugging is that activity which is performed after executing a successful test case.

• Remember that a successful test case is one that shows that a program does not do what it was designed to do.

• Debugging consists of determining the exact nature and location of the suspected error and fixing the error.

• Locating the error represents about 95% of the activity.

• Process of finding the location of an error, given a suspicion that an error exists, based on the results of a successful test case.
Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is an action that results in the removal of the error.

Debugging is a two-step process that begins when you find an error as a result of a successful test case.

**Step 1 is the determination of the exact nature and location of the suspected error within the program.**

**Step 2 consists of fixing the error.**
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 non corresponding data are a symptom of an underlying cause as yet hidden. Debugging attempts to match symptom with cause, thereby leading to error correction.
Debugging 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 one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.
## Difference between Testing and Debugging

<table>
<thead>
<tr>
<th>TESTING</th>
<th>DEBUGGING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing is the process of executing a program or system with the aim of findings errors or bugs.</td>
<td>Correcting these errors or bugs (found during testing) is debugging.</td>
</tr>
</tbody>
</table>
What is a BUG

A fault in a program, which causes the program to perform in an unintended or unanticipated manner.
A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be buggy.
Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, defect reports etc.

Why BUG Occurs

There are so many reasons that can cause a bug......

- Code errors
- Unfinished requirements
- Misunderstanding of user needs
- Requirements that are not detailed enough
- Logic errors in the design documents
- Lack of documentation
- Not enough sufficient testing
Bug Lifecycle

In software development process, the bug has a life cycle. The bug should go through the life cycle to be closed. A specific life cycle ensures that the process is standardized. The bug attains different states in the life cycle. The life cycle of the bug can be shown diagrammatically as follows:

States of a bug:
1) New
2) Open
3) Assign
4) Test
5) Verified
6) Deferred
7) Reopened
8. Rejected
9) Closed
Description of Stages

**New:** When the bug is posted for the first time, its state will be NEW. This means that the bug is not yet approved.

**Open:** After a tester has posted a bug, the lead of the tester approves that the bug is genuine and he changes the state as OPEN.

**Assign:** Once the lead changes the state as OPEN, he assigns the bug to corresponding developer or developer team. The state of the bug now is changed to “ASSIGN”.

**Resolved/Fixed/Test:** When developer makes necessary code changes and verifies the changes then he/she can make bug status as ‘Fixed’ and the bug is passed to testing team.

**Deferred:** The bug, changed to deferred state means the bug is expected to be fixed in next releases. The reasons for changing the bug to this state have many factors. Some of them are priority of the bug may be low, lack of time for the release or the bug may not have major effect on the software.
Description of Stages

Pending Reject: If the developers think that a particular behavior of the system, which the tester reports as a bug has to be same and the bug is invalid, in that case, the bug is rejected and marked as ‘Pending Reject’.

Rejected/Invalid: Some times developer or team lead can mark the bug as Rejected or invalid if the system is working according to specifications and bug is just due to some misinterpretation.

Pending Retest: After the bug is fixed, it is passed back to the testing team to get retested and the status of ‘Pending Retest’ is assigned to it.

Retest: The testing team leader changes the status of the bug, which is previously marked with ‘Pending Retest’ to ‘Retest’ and assigns it to a tester for retesting.

Duplicate: If the bug is repeated twice or the two bugs mention the same concept of the bug, then one bug status is changed to “DUPLICATE”.

Verified: Once the bug is fixed and the status is changed to TEST, the tester tests the bug. If the bug is not present in the software, he approves that the bug is fixed and changes the status to VERIFIED.
Description of Stages

**Could not reproduce:** If developer is not able to reproduce the bug by the steps given in bug report by QA then developer can mark the bug as ‘CNR’. QA needs action to check if bug is reproduced and can assign to developer with detailed reproducing steps.

**Reopened:** If the bug still exists even after the bug is fixed by the developer, the tester changes the status to REOPENED. The bug traverses the life cycle once again.

**Closed:** Once the bug is fixed, it is tested by the tester. If the tester feels that the bug no longer exists in the software, he changes the status of the bug to CLOSED. This state means that the bug is fixed, tested and approved.

**Postponed:** Sometimes, testing of a particular bug has to be postponed for an indefinite period. This situation may occur because of many reasons, such as unavailability of Test data, unavailability of particular functionality etc. That time, the bug is marked with ‘Postponed’ status.
Bug Management

It is common practice for software to be released with known bugs that are considered non-critical. While software products may, by definition, contain any number of unknown bugs, measurements during testing can provide an estimate of the number of likely bugs remaining.

Most big software projects maintain two lists of "known bugs"—those known to the software team, and those to be told to users. This is not dissimulation, but users are not concerned with the internal workings of the product. The second list informs users about bugs that are not fixed in the current release, or not fixed at all, and a workaround may be offered.

There are various reasons for not fixing bugs:

- The developers often don't have time or it is not economical to fix all non-severe bugs.
- The bug could be fixed in a new version or patch that is not yet released.
- The changes to the code required to fix the bug would be large, and would bring with them the chance of introducing other bugs into the system.
- It's "not a bug". A misunderstanding has arisen between expected and provided behavior.
Common Types Of Computer Bugs

Maths bugs
• Division by zero
• Arithmetic overflow or underflow

Logic bugs
• Infinite loops and infinite recursion

Syntax bugs
• Use of the wrong operator, such as performing assignment instead of equality

Resource bugs
• Using an un-initialized variable
• Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
• Buffer overflow, in which a program tries to store data past the end of allocated storage.

Team working bugs
• Comments out of date or incorrect: many programmers assume the comments accurately describe the code
• Differences between documentation and the actual product
Reporting

**Daily Summary Data:**
In addition to entering new bugs and closing fixed bug in the bug tracking application,
- Number of test cases completed
- Number of new issues identified and reported
- Number of test cases, which failed
- Number of test cases blocked
- The executed test plans
- The current list of existing bugs by severity

**Weekly Summary Report:**
At the end of each week, creates a weekly report to provide some additional information to clients.

**End of Cycle Reports**
Q.A. engineers execute all test cases for a build, supply a complete set of test results for the cycle. provide the following information:
- Each completed test plan
- List of bugs found in this testing cycle
- List of bugs fixed in this testing cycle
- List of bugs deferred to a later release
- Full test tracking document for the completed cycle
Types of Bugs/Defects

Defects that are detected by the tester are classified into categories by the nature of the defect. The following are the classification:

**Showstopper (X):** The impact of the defect is severe and the system cannot go into the production environment without resolving the defect since an interim solution may not be available.

**Critical (C):** The impact of the defect is severe, however an interim solution is available. The defect should not hinder the test process in any way.

**Non critical (N):** All defects that are not in the X or C category are deemed to be in the N category. These are also the defects that could potentially be resolved via documentation and user training. These can be GUI defects.
Defect Report

Particulars that have to be filled by a tester are:

**Defect Id:** Number associated with a particular defect, and henceforth referred by its ID.

**Date of execution:** The date on which the test case which resulted in a defect was executed.

**Defect Category:** These are explained later on.

**Severity:** As explained, it can be Critical, Non-Critical and Showstopper.

**Module ID:** Module in which the defect occurred.

**Status:** New, Open, Assign, Test, Verified, Deferred, Reopened, Rejected, Closed.

**Defect description:** Description as to how the defect was found, the exact steps that should be taken to simulate the defect, other notes and attachments if any.
Defect Report

**Test Case Reference No:** The number of the test case and script in combination which resulted in the defect

**Owner:** The name of the tester who executed the test case

**Test case description:** The instructions in the test cases for the step in which the error occurred

**Expected Result:** The expected result after the execution of the instructions in the test case descriptions

**History of the defect:** Normally taken care of the automated tool used for defect tracking and reporting.

**Attachments:** The screen shot showing the defect should be captured and attached

**Responsibility:** Identified team member of the development team for fixing the defect.
Methods of Debugging

The methods of debugging are listed below.

1. **Debugging by Brute Force Attack.**
2. **Debugging by Induction**
3. **Debugging by Deduction**
4. **Debugging by Backtracking**
5. **Debugging by Testing**
Debugging by Brute Force

The most common scheme for debugging a program is the “brute force” method. It is popular because it requires little thought and is the least mentally taxing of the methods, but it is inefficient and generally unsuccessful.

Brute force methods can be partitioned into at least three categories:

1. Debugging with a storage dump.

2. Debugging according to the common suggestion to “scatter print statements throughout your program.”

3. Debugging with automated debugging tools.
Brute force Method - use of Storage Dump

The first, debugging with a storage dump (usually a crude display of all storage locations in hexadecimal or octal format).

Memory/Storage dump is the display or printout of the contents of memory. When a program abends, a memory dump can be taken in order to examine the status of the program at the time of the crash. The programmer looks into the buffers to see which data items were being worked on when it failed. Counters, variables, switches and flags are also inspected. It is the most inefficient of the brute force methods. Here's why:

- There's a massive amount of data, most of which is irrelevant.
- A memory dump is a static picture of the program, showing the state of the program at only one instant in time; to find errors, you have to study the dynamics of a program (state changes over time).
- A memory dump is rarely produced at the exact point of the error, so it doesn't show the program's state at the point of the error.
- There aren't adequate methodologies for finding errors by analyzing a memory dump.
**Brute force Method- Scattering Print Statements Randomly**

Scattering statements throughout a failing program to display variable values is better than a dump as it is not static and shows the dynamics of a program, but this method, too, has many shortcomings:

- Again you are not thinking.
- It produces a massive amount of data to be analyzed.
- It requires you to change the program; such changes can mask the error, or introduce new errors.
- It may work on small programs, but the cost of using it in large programs like operating systems is quite large.
- It is often too costly or even infeasible for real-time software.
Automated debugging tools work similarly to inserting print statements within the program, but rather than making changes to the program, you analyze the dynamics of the program with the debugging features of the programming language.

A common function of debugging tools is the ability to set breakpoints that cause the program to be suspended when a particular statement is executed or when a particular variable is altered, and then the programmer can examine the current state of the program.

Again, this method is largely hit or miss and often results in an excessive amount of irrelevant data.
Debugging by Brute Force

The general problem with these brute force methods is that they ignore the process of thinking.

There also is some evidence to indicate that whether the debugging teams are experienced programmers or students, people who use their brains rather than a set of aids work faster and more accurately in finding program errors.

Therefore, we could recommend brute force methods only

(1) when all other methods fail or

(2) as a supplement to, not a substitute for, the thought processes
Debugging by Induction

It should be obvious that careful thought will find most errors without the debugger even going near the computer. One particular thought process is induction, where you move from the particulars of a situation to the whole. That is, start with the clues (the symptoms of the error, possibly the results of one or more test cases) and look for relationships among the clues. The induction process is illustrated in Figure
Debugging by Induction

The steps are as follows:

1. *Locate the pertinent data.* A major mistake debuggers make is failing to take account of all available data or symptoms about the problem. The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly.

2. *Organize the data.* Remember that induction implies that you’re processing from the particulars to the general, so the second step is to structure the pertinent data to let you observe the patterns.

3. *Devise a hypothesis.* Next, study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error.

4. *Prove the hypothesis.* A major mistake at this point, given the pressures under which debugging usually is performed, is skipping this step and jumping to conclusions to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before you proceed. If you skip this step, you’ll probably succeed in correcting only the problem symptom, not the problem itself.

5. *Fix the Error.* If hypothesis is proved successfully then go on to fix the error else repeat steps 3 and 4.
Debugging by Deduction

The process of deduction proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error). See the Figure.
Debugging by Deduction

The steps are as follows:

1. **Enumerate the possible causes or hypotheses.** The first step is to develop a list of all conceivable causes of the error. They don’t have to be complete explanations; they are merely theories to help you structure and analyze the available data.

2. **Use the data to eliminate possible causes.** Carefully examine all of the data, particularly by looking for contradictions, and try to eliminate all but one of the possible causes. If all are eliminated, you need more data through additional test cases to devise new theories.

3. **Refine the remaining hypothesis.** The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory.

4. **Prove the remaining hypothesis.** This vital step is identical to step 4 in the induction method.
Debugging by Backtracking

An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray.

In other words, start at the point where the program gives the incorrect result—such as where incorrect data were printed.

At this point you deduce from the observed output what the values of the program’s variables must have been. By performing a mental reverse execution of the program from this point and repeatedly using the process of “if this was the state of the program at this point, then this must have been the state of the program up here,” you can quickly pinpoint the error.
The last “thinking type” debugging method is the use of test cases. Consider two types of test cases:

Test cases for testing, where the purpose of the test cases is to expose a previously undetected error, and

Test cases for debugging, where the purpose is to provide information useful in locating a suspected error.

The difference between the two is that test cases for testing tend to be “fat” because you are trying to cover many conditions in a small number of test cases. Test cases for debugging, on the other hand, are “slim” since you want to cover only a single condition or a few conditions in each test case.

In other words, after a symptom of a suspected error is discovered, you write variants of the original test case to attempt to pinpoint the error. Actually, this method is not an entirely separate method; it often is used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis. It also is used with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.
General Guidelines on Fixing Bugs

- Where there is one bug, there is likely to be another.
- Fix the error, not just the symptom.
- The probability of the fix being correct is not 100%.
- The probability of the fix being correct drops as the size of the program increases.
- Beware of the possibility that a bug fix creates a new bug.

Guaranteed Bug-Free