Bug Fixing in a Nutshell

Purpose

Bug fixing is a challenge for any developer. We have identified two steps in the process of fixing a bug and make suggestions for prevention of bugs. This document is aimed to educate university students who are specifically using Java but can easily be translated to other programming languages.

General Comments

Tip: Fixing a bug comes down to input and output. Was the input (specified in the expected way) and the output correct? This is bug fixing in a nutshell!

You need to make mistakes and learn from them to understand how things can go wrong, but hopefully you can learn some things from this document.

Step 1: Identify the bug.

Step one in the process involves identifying the type of bug, debugging and replication of the bug.

Identify the type (here are just some):

1. Functional Errors: These occur when the software does not perform its intended function.
    a. Example: A program that is supposed to add two numbers together but instead subtracts them.
2. Syntax Errors: These occur when the syntax of the code does not conform to the rules of the programming language.
    a. For example, if a semicolon is missing at the end of a statement, or if a variable is misspelled, this would be a syntax error.
3. Logical Errors: These occur when the code is syntactically correct but does not produce the expected results. This can be caused by incorrect algorithms or flawed logic in the code. More often this is due to incorrect data type assignments.
    a. Example: A program that is supposed to calculate the average of a set of numbers but instead produces the sum of the numbers. (Side note: it is important to consider what happens if one of these numbers is 0, or, in a different case, null).
4. Calculation Errors: These occur when the code performs incorrect calculations. This can be caused by rounding errors or incorrect math functions.
    a. Example: A program that is supposed to calculate the area of a circle but instead uses the wrong formula, resulting in an incorrect calculation.

5. Out of Bounds Errors: These occur when a program attempts to access an element of an array or a memory location that is outside the boundaries of the array or memory space. This can be caused by incorrect indexing or a lack of bounds checking.
    a. Example: A program that attempts to access the 6^{th} element of a 5-element array.

Debugging Code tips: how to do it in the quickest and most painless way:

- Make breakpoints at every major step (it is important to then identify these major steps), and check if the input and output is what you expect. Try eliminating where the bug could be in the code by homing in (hopefully eliminating ½, then ¼ etc).
- Remove sections of your code and check if the output is as expected. That is, the section you removed works as a stand alone and the rest of the code without that section works.
- If you have identified a problem that occurs at a specific point, you can skip the painful process of stepping through a million times by creating a simple breakpoint line. E.g. (breakpoint on line 2):
    i. If (i == 6 && j == 98) {
    ii. System.out.println("Hi");
    iii. }
- The application "WinMerge" is a helpful tool to compare files (if you want to find at what point there is a difference) -> It is important to note case, that is, make sure that data being compared (if strings) are all lower case for example.
- If the input is not as expected, go back one step, and check the output of the previous method.
- Log results at various points (print to command line) such that you can easily identify at what point did the input/output mismatch occur.
- Make sure the data that you have is not the problem; use expected data first and see if that works, then check the input created through the program.
- Check that 'catch' clauses output a message that is useful. E.g.
    i. System.out.println("Hi"); -> Not useful
    ii. System.out.println("An error occurred: " + e.getMessage()); -> Useful.
- Version control (aka git) can be useful to see if the program previously worked and what changed to cause it not to work.
- Null checks are key. Do not assume data is what you expect! Break points on null checks can also be useful. E.g. What happens on line 2 if x is null?
    i. for(String x : oldStringArray) {
    ii. if (x.length > 5) newStringArray.Add(x);
    iii. }

It is important to **replicate** the bug, that is, create a list of specific steps you must take in your application to get to the bug. In this way you can check at each step if the input is correct, and the output is correct using the above-mentioned tips.

Q&A: More applicable to bigger systems.

How do you handle large amounts of data with a bug?

- There is unfortunately no easy way to deal with this; you cannot directly debug it! Here are some tips:
    a. Null pointers and checking if the null points get hit (breakpoints).
    b. For bigger environments, if you can replicate it, then it is likely a system problem and if you cannot, then it is likely a data problem that needs to be addressed.
    c. Implement Useful catching processes (see below).
    d. SQL Profiler can be a useful tool for large amounts of data.

How do we go about useful catching processes?

a. Error codes can be important -> Know your HTML codes.
b. Catch at the beginning (making sure you get the correct information) and the end before saving -> Good practice standards, consistent.
c. Understanding the implications of things. Why is this not correct is important (and being able to explain that) and what subsequent events must happen.
    a. E.g.: User refreshes halfway through a backend call, what happens?
d. Catching may need to be able to reverse what was done to avoid massive problems (incomplete data for example).

Step 2: Fix

Tips for fixing each error type upon identification (listed above):

1. Functional Errors: Modify the code or design to ensure that the software performs its intended function.
2. Syntax Errors: Correct the syntax. This can involve adding or removing characters, correcting spelling errors, or making other adjustments to ensure that the code follows the rules of the programming language.
3. Logical Errors: Modify the code to correct the flawed logic (very hard to give suggestions)
4. Calculation Errors: Modify the code to use the correct formula or function.
5. Out of Bounds Errors: This can involve adding bounds checking code to prevent accessing elements outside of the array or memory space, adjusting the indexing of the array or memory, or modifying the code to handle out of bounds conditions in a more robust way.

When going about fixing, it is very important to consider the level of complexity of the bug, that is, what does it affect? If you change one line, what could possibly break? Are there

similar cases that could occur that are like the current problem (consider extreme cases)? Identify all cases of a specific method being used, and check if they still work after the change.

Consider code changes (as above), configuration changes (settings -> clear cache for example) and third-party dependencies (are the libraries working as expected?).

Future Proofing and Testing

No one likes to do it, but it is important to use modular code and to comment your code. Version control is a must!!

Furthermore, Documentation is key in the real world (even though it is not the fun part of a project). What happens when another developer takes over the project? Or extends the functionalities of the code? Or you work with clients?

Follow good coding principles, here are some acronyms:

- KISS (Keep It Simple Stupid)
- DRY (Don't repeat yourself)
- YAGNI (You Aren't Gonna Need it)
- SLAP (Single Level of Abstraction Principle)
- SOLID
    a. Single Responsibility Principle (SRP)
    b. Open-Closed Principle (OCP)
    c. Liskov Substitution Principle (LSP)
    d. Interface Segregation Principle (ISP)
    e. Dependency Inversion Principle (DIP)

The key to testing is to try to break your code!

Furthermore, there are different types of testing, namely:

- **Unit Testing**: Unit testing is the process of testing individual units or components of software in isolation from the rest of the system. To effectively perform unit testing, you should write test cases that cover all possible scenarios for each unit, including both valid and invalid inputs.
- **Integration Testing**: Integration testing is the process of testing how different units of software work together. To effectively perform integration testing, you should write test cases that simulate how different units interact with each other.
- **System Testing**: System testing is the process of testing the entire software system, to ensure that it meets all requirements and works as intended. To effectively perform

system testing, you should write test cases that cover all possible scenarios for the system, including edge cases and stress testing scenarios.

- **Performance Testing**: Performance testing is the process of testing the software's performance under different workloads, to ensure that it meets performance requirements. This type of testing focuses on identifying bottlenecks or other issues that cause the system to perform poorly under certain conditions. To effectively perform performance testing, you should write test cases that simulate real-world workloads, and use tools to measure and analyse system performance under different scenarios.
- **Acceptance Testing**: Acceptance testing is the process of testing software to ensure that it meets the requirements and expectations of the end-users. This type of testing focuses on validating that the software meets the needs of the intended users and works as intended in their environment. To effectively perform acceptance testing, you should involve end-users in the testing process, write test cases that reflect their needs and requirements, and use feedback to improve the software.

Identify critical parts of code and what must ALWAYS be tested after set of work done!

Written by Peter Couper

In collaboration with DreamTechLabs

In collaboration with Stellenbosch University Computer Science Department