6.005 — Software Construction on MIT OpenCourseWare (https://ocw.mit.edu/ans7870/6/6.005/s16/) | OCW 6.005 Homepage (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/) Spring 2016

Reading 11: Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today’s class is systematic debugging.

Sometimes you have no choice but to debug – particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module. For those situations, we can suggest a systematic strategy for more effective debugging.

Reproduce the Bug

Start by finding a small, repeatable test case that produces the failure. If the bug was found by regression testing, then you’re in luck; you already have a failing test case in your test suite. If the bug was reported by a user, it may take some effort to reproduce the bug. For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution.

Nevertheless, any effort you put into making the test case small and repeatable will pay off, because you’ll have to run it over and over while you search for the bug and develop a fix for it. Furthermore, after you’ve successfully fixed the bug, you’ll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.

Here’s an example. Suppose you have written this function:

```
/**
 * Find the most common word in a string.
 * @param text string containing zero or more words, where a word
 *     is a string of alphanumeric characters bounded by nonalphanumerics.
 * @return a word that occurs maximally often in text, ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ...
}
```

A user passes the whole text of Shakespeare's plays into your method, something like `mostCommonWord(allShakespearesPlaysConcatenated)` , and discovers that instead of returning a predictably common English word like `"the"` or `"a"` , the method returns something unexpected, perhaps `"e"` .

Shakespeare's plays have 100,000 lines containing over 800,000 words, so this input would be very painful to debug by normal methods, like print-debugging and breakpoint-debugging. Debugging will be easier if you first work on reducing the size of the buggy input to something manageable that still exhibits the same (or very similar) bug:

- does the first half of Shakespeare show the same bug? (Binary search! Always a good technique. More about this below.)
- does a single play have the same bug?
- does a single speech have the same bug?

Once you've found a small test case, find and fix the bug using that smaller test case, and then go back to the original buggy input and confirm that you fixed the same bug.

Understand the Location and Cause of the Bug

To localize the bug and its cause, you can use the scientific method:

1. **Study the data.** Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.

2. **Hypothesize.** Propose a hypothesis, consistent with all the data, about where the bug might be, or where it *cannot* be. It's good to make this hypothesis general at first.

3. **Experiment.** Devise an experiment that tests your hypothesis. It's good to make the experiment an *observation* at first – a probe that collects information but disturbs the system as little as possible.

4. **Repeat.** Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis. Hopefully you have ruled out some possibilities and narrowed the set of possible locations and reasons for the bug.

Let's look at these steps in the context of the `mostCommonWord()` example, fleshed out a little more with three helper methods:

```
/**
 * Find the most common word in a string.
 * @param text string containing zero or more words,
 *     where a word is a string of alphanumeric
 *     characters bounded by nonalphanumerics.
 * @return a word that occurs maximally often in text,
 *         ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ... words = splitIntoWords(text); ...
    ... frequencies = countOccurrences(words); ...
    ... winner = findMostCommon(frequencies); ...
    ... return winner;
}

/** Split a string into words ... */
private static List<String> splitIntoWords(String text) {
    ...
}

/** Count how many times each word appears ... */
private static Map<String,Integer> countOccurrences(List<String> words) {
    ...
}

/** Find the word with the highest frequency count ... */
private static String findMostCommon(Map<String,Integer> frequencies) {
    ...
}
```

1. Study the Data

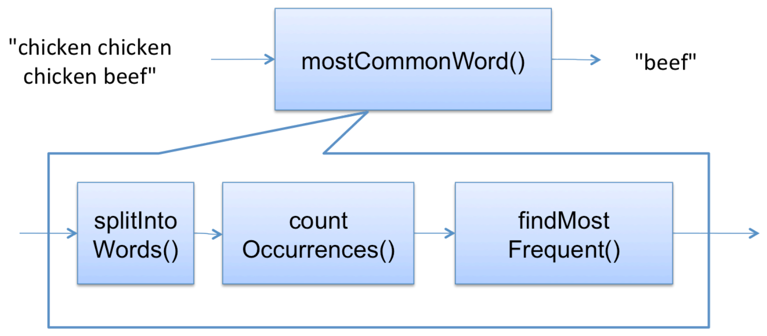
One important form of data is the stack trace from an exception. Practice reading the stack traces that you get, because they will give you enormous amounts of information about where and what the bug might be.

The process of isolating a small test case may also give you data that you didn't have before. You may even have two related test cases that *bracket* the bug in the sense that one succeeds and one fails. For example, maybe `mostCommonWords("c c, b")` is broken, but `mostCommonWords("c c b")` is fine.

2. Hypothesize

It helps to think about your program as modules, or steps in an algorithm, and try to rule out whole sections of the program at once.

The flow of data in `mostCommonWord()` is shown at right. If the symptom of the bug is an exception in `countOccurrences()`, then you can rule out everything downstream, specifically `findMostFrequent()`.



Then you would choose a hypothesis that tries to localize the bug even further. You might hypothesize that the bug is in `splitIntoWords()` , corrupting its results, which then cause the exception in `countOccurrences()` . You would then use an experiment to test that hypothesis. If the hypothesis is true, then you would have ruled out `countOccurrences()` as the source of the problem. If it's false, then you would rule out `splitIntoWords()` .

3. Experiment

A good experiment is a gentle observation of the system without disturbing it much. It might be:

- Run a **different test case.** The test case reduction process discussed above used test cases as experiments.
- Insert a **print statement** or **assertion** in the running program, to check something about its internal state.
- Set a **breakpoint** using a debugger, then single-step through the code and look at variable and object values.

It's tempting to try to insert *fixes* to the hypothesized bug, instead of mere probes. This is almost always the wrong thing to do. First, it leads to a kind of ad-hoc guess-and-test programming, which produces awful, complex, hard-to-understand code. Second, your fixes may just mask the true bug without actually removing it.

For example, if you're getting an `ArrayOutOfBoundsException` , try to understand what's going on first. Don't just add code that avoids or catches the exception, without fixing the real problem.

Other tips

Bug localization by binary search . Debugging is a search process, and you can sometimes use binary search to speed up the process. For example, in `mostCommonWords` , the data flows through three helper methods. To do a binary search, you would divide this workflow in half, perhaps guessing that the bug is found somewhere between the first helper method call and the second, and insert probes (like breakpoints, print statements, or assertions) there to check the results. From the answer to that experiment, you would further divide in half.

Prioritize your hypotheses . When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than recently-added code. Java library code is probably more trustworthy than yours. The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

Swap components . If you have another implementation of a module that satisfies the same interface, and you suspect the module, then one experiment you can do is to try swapping in the alternative. For example, if you suspect your binarySearch() implementation, then substitute a simpler linearSearch() instead. If you suspect java.util.ArrayList, you could swap in java.util.LinkedList instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailing components, however, so don't do this unless you have good reason to suspect a component.

Make sure your source code and object code are up to date. Pull the latest version from the repository, and delete all your binary files and recompile everything (in Eclipse, this is done by Project → Clean).

Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.005 students usually do know what you're talking about, so they're even better.

Sleep on it. If you're too tired, you won't be an effective debugger. Trade latency for efficiency.

Fix the Bug

Once you've found the bug and understand its cause, the third step is to devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the code? Try to make the code safe from future bugs like this. Also consider what effects your fix will have. Will it break any other code?

Finally, after you have applied your fix, add the bug's test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.

Summary

In this reading, we looked at how to debug systematically:

- reproduce the bug as a test case, and put it in your regression suite
- find the bug using the scientific method
- fix the bug thoughtfully, not slapdash

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.