ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 19 — Debugging

Patrick Lam and Jeff Zarnett

We've discussed testing, and now it's time to look at debugging. At one time, many programmers believed that debugging could not be done in a systematic way. It was expected that the only way to do it was the Richard Feynman method: *write down the problem; think very hard; write down the answer*. This approach might happen sometimes, but it requires deep insight to the code and a sudden burst of brilliance to get it done. That's not very realistic, especially if you are debugging code you didn't write in the first place [GHKW08].

All software has bugs. Programs can be divided into two categories: those that are so simple they obviously have no bugs, and those that have no obvious bugs. The first category of programs is smaller than we might think. Look at our labs - the amount of code we write is small, and yet, I can say with certainty that every implementation, including mine, has bugs.

Debugging is something you will do a lot of in co-op and even in other classes. So let's learn the systematic, scientific method.

Debugging Strategy

We can use the scientific method as a strategy for debugging. Here's an adaptation of the scientific method, from Zeller's book:

1. Observe a failure (i.e. as described in the problem description).
2. Invent a *hypothesis* as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make *predictions*.
4. Test the hypothesis by *experiments* and further *observations*:
   - If the experiment satisfies the predictions, refine the hypothesis.
   - If the experiment does not satisfy the predictions, create an alternate hypothesis.
5. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

Once you have a cause for the failure, then you can also fix the failure by modifying the program.

The more explicit you are about the various products (what's the failure? what's the hypothesis?), the easier you'll find it to fix the problem.

Bug Localization. A key part of the hypothesis underlying a bug is the location of the bug in the code. This is especially true for larger software, and should be the first step in your debugging strategy.

You should attempt to localize the bug to within a subsystem or a set of modules. You may have to do this iteratively, continuing to localize the bug to narrower and narrower sets of modules. Once you know *where* the bug is, you'll find it to be easier to refine your hypothesis about *what* the bug may be.

But how to localize the bug? The best way to do this is to split the problem space. Suppose you created a program that should do ten things in a sequence. It crashes, so you have a bug that you need to remove.

When you look at the output, you see that the first seven things worked fine, but the last three are not visible from the output. Now we have some idea about where the problem might be: it could have crashed on item eight, nine, or ten [Rea03]. And as a first guess, you might focus in on number eight.

When you start debugging, it might look like every possible line in the code is potentially the one going wrong. As you gain experience you will find some are more likely to cause problems than others: input/output, a complex block, interaction with third-party code. Finding roughly where the bug is will be a start, but we need to find which line the bug is on.

Debugging Tactics

You can combine the general strategy with the tactics I'll present here to debug programs. These tactics are applicable to many languages, but I'll tell you about Java implementations. There are four general kinds of tactics that you can use to debug your code.

- *Code review*: I find this to be the most effective technique. Stare at your code and think hard about what it's supposed to be doing, compared to what it's actually doing.
- *Code instrumentation*: Put `print` statements (or equivalent) into your code. Create hypotheses about what the program is doing, insert code to get more observable outputs, and verify your hypotheses by running the program.
- *Single-step execution*: You can also step through your code in a debugger line-by-line and manually inspect the program state. This tactic is only effective once the bug is localized down to a small area.
- *Take a Break:* Sometimes when you are stumped, taking a break will help. A 15-minute coffee break or a good night's sleep might be all your brain needs if you are stuck [Rea03]. If you take some time away from the code, your subconscious continues to work on the problem and may present you with a solution later on.

Tactics for bug localization. To formulate a hypothesis about the location or identity of a bug, you must collect sufficient diagnostic information using the above tactics: you might supply different inputs, run the instrumented program, set breakpoints, and examine internal state. Note that doctors do this—they send a patient for tests so that there is sufficient diagnostic info to formulate a hypothesis about the underlying cause.

Code instrumentation

This is a popular choice when you cannot attach the debugger to the program - either because it runs on a server or at some client site. Logs can also provide some statistics and performance related data. The amount of data to output in a log naturally involves a tradeoff between brevity and information. If not enough information is output to the log, you will not have what you need to identify and fix the bug. If too much data is output to the log, the data you need will be lost in the noise [Rea03].

Logging for Android Development `System.out.println()` is great for debugging console applications, but doesn't work on Android. Instead, use:

```
Log.d("tag", "i = "+i);
```

This writes out a debug (d) logging message, which appears e.g. in your Eclipse `LogCat` window. Instead of `d`, you can write `Log.d`, `.i`, `.v`, `.w` or `.wtf`. You can then filter out logging messages by level or tag, so that you only see the ones you're interested in.

Beyond `Log.d()` for getting information out of your program, you can rely on assertions. Recall what we learned about Assertions:

Assertions. An assertion is a statement about the world; for instance, I assert that PowerPoint is often harmful to education. In the context of code, assertions are logical expressions that should always be true. When the program executes an assertion, it verifies that the logical expression indeed evaluates to true. If not, then it throws an `AssertionError`, which will usually stop the program. You can also supply a second parameter to the assertion, containing a message to be reported with the error in the event of assertion failure.

Two examples of assertions:

- `if (i % 2 == 0) { ... } else { /* i is odd */ }`: we know that `i` is odd, so use an assertion to document this, e.g

  ```
  if (i % 2 == 0) { ... } else { assert i % 2 == 1; }
  ```

- When implementing a doubly-linked list, you know that following `next` and then `prev` should get back to the original node. Include:

  ```
  assert this.next.prev == this : "List fails doubly-linked node invariant";
  ```

Assertions should never have side effects.

Single-step execution

Single-step execution can help you both with localization and hypothesis testing. Since programs can be huge, you don't want to run through the whole program every time. Hence *breakpoints*. You can run the program until it hits the breakpoint and then single-step.

Sprinkle breakpoints throughout the program, before lines that you suspect to be faulty. Then single-step across the questionable lines, inspecting (and perhaps modifying) relevant program state both before and after the line. If the state is infected, then you've isolated the bug.

Kinds of Breakpoints. Eclipse supports a variety of different breakpoint types[R.13], although most of the time a Line Breakpoint will suffice.

- *Line breakpoints* halt program execution at a particular line. (You can also make them conditional: only halt program execution when a particular condition is true; or when a particular value changes.)
- *Exception breakpoints* halt program execution and load the debugger when the program throws a particular exception.
- *Watchpoints* halt program execution when a particular memory location changes (or is accessed).
- *Method breakpoints* halt program execution upon entry or exit from a particular method. (Entry is easy to simulate with line breakpoints, but exit is potentially hard for large methods.)

It's easy to waste a lot of time fiddling with the debugger. Debuggers work best if you know what you're looking for ahead of time, and have a specific experiment in mind for a particular debugger invocation. Otherwise you're likely to just waste time. Single-stepping may be useful as you're gaining experience with software development, but tends to be generally less useful as you get better as debugging.

Types of Bugs

Bugs are our enemies, so we need to get to know them and get to understand them, and we will be a long way towards defeating them. Our guides in this matter [GHKW08] gives us the members of the bug family. Fixing the bug is going to be much easier if the failure can be reproduced consistently (we say the bug is reproducible). How to reproduce the bug will depend on the kind of bug it is.

The Common Bug

This one is the basic type. It is in the source code, and behaves predictably. It is sometimes the result of an ambiguous specification or something not tested, or simply programmer error.

The Sporadic Bug

The common bug strikes predictably when a test case is executed. The sporadic bug is not so consistent, but it can be lured out if you are careful. Some tricks that can lure out the bug: leaving a trap in place (a watchdog to identify when something has gone wrong) or finding the right bait (test case). Once the right test case is found, however, this bug is reproducible.

Heisenbugs

Recall the Heisenberg uncertainty principle from physics: *The more precisely the position of a particle is known, the less precisely the momentum is, and vice versa.* In software, the situation is analogous: the harder you try to debug the problem, the better the bug is at hiding. When you step through the code the bug doesn't happen, or printing debug statements prevents the problem from happening at all.

Heisenbugs can usually be broken down into one of the following problems:

Race Condition A *race condition* is the cause of most Heisenbugs. This is a situation where the program behaviour depends on the order of completion of certain tasks (that's why it's called a race). Parallel and multi-threaded programs suffer from these commonly, but it can happen in single threaded programs. Execution order is important that's why adding additional debugging statements will change how long it takes for a task to complete and possibly suppress the bug.

Memory Errors Another possible source of Heisenbugs are memory access problems. Reading an uninitialized value, reading off the end of an array, reading an area of memory after it's been freed, and so on, can be sources of bugs that are hard to reproduce. If the variable is uninitialized and we read it, who knows what value we will get. It might be zero sometimes (and in that circumstance the program works okay) and other times it might have another value and the error occurs.

This isn't a problem in Java, because accessing an uninitialized variable is a compile-time warning and will always result in a null value at runtime. Similarly, because of how memory works in Java, we won't have the situation of using a variable after the memory for it has been released. Although we use it in the labs, Java is not the only programming language in the world, so we had better be prepared for this situation.

Optimization

> … *premature optimization is the root of all evil.*
>
> Donald Knuth

Optimizations will sometimes be necessary and are occasionally desirable. However, sometimes an optimization is a shortcut. This might pay off, but in other cases the shortcut results in an error. The optimization will need to be changed or turned off. If you suspect that an optimization is part of the problem, consider what happens if the algorithm is executed the "slow" way.

Bugs Hiding Behind Bugs

Sometimes we have multiple bugs that are interacting in some way. As a rule, you need to fix the first bug that occurs before the others. Although we typically want to change only one thing at a time, in this situation, it's necessary to keep very good notes and try to solve the bugs in sequence.

Secret Bugs

One problem faced in the real world is the secret bug – it strikes when the customer is using the software and encounters a bug, but they don't want to tell you about it because it involves confidential information, or they are not experts in software and cannot provide you the relevant information.

You can try to reproduce the problem in house. Whether this is by creating test data or asking the customer for an anonymized set of test data, it's necessary to reproduce the bug. It might also be helpful to see what's going on at the customer site. Maybe you can visit in person, or via remote tools (screen sharing, console login, etc). In other cases you will have to settle for logging.

Configuration or Environment Bugs

Another problem you may face is that there's nothing wrong with the software, but something wrong with the environment. The user might not have the permissions necessary for the program to run as expected. The software can be changed such that it handles the situation more gracefully, or change the environment or configuration procedure to avoid the error in the first place.

Hardware Bugs

In embedded systems there are lots of possibilities for hardware bugs to deal with. Proving that the error comes from hardware can be a challenge, but once it's done we may be able to work around this kind of bug in our software.

However, hardware bugs are not limited to embedded systems. Consider a very famous bug in the original Pentium processors (this is known as the FDIV bug), where the processor returned an incorrect value when doing a floating point division operation. The bug was identified by Thomas Nicely in 1994 [Nic11].

$4195835.0/3145727.0 = 1.3338204491362410025$ (Correct value)
$4195835.0/3145727.0 = 1.3337390689020375894$ (Flawed Pentium)

Intel ended up fixing the problem in its future processors, and recalling the flawed chips. As a workaround, Intel recommended multiplying the numerator and denominator each by $15/16$ before the division is performed (this factor will of course cancel when the division takes place) because it shifts the bits in such a way that the error is not encountered.

The Not-A-Bug

Sometimes when you are examining a bug, you find that it's not really a bug at all. The system is performing the way the specification says it should. In this case you may not need to change anything, or it may be a bug in the specification. Explaining it to the customer, however, might not be that easy.

References

[GHKW08] Thorsten Gröther, Ulli Holtmann, Holger Keding, and Markus Wloka. *The Developer's Guide to Debugging*. Springer, 2008.

[Nic11] Thomas R. Nicely. Pentium FDIV flaw, 2011. Online; accessed 15-May-2013. URL: `http://www.trnicely.net/pentbug/pentbug.html`.

[R.13] Prakash G. R. Types of Breakpoints in Eclipse, 2013. Online; accessed 21-December-2013. URL: `http://www.eclipse-tips.com/tips/29-types-of-breakpoints-in-eclipse`.

[Rea03] Robert L. Read. How to be a Programmer: A Short, Comprehensive, and Personal Summary, 2003. Online; accessed 1-December-2013. URL: `http://samizdat.mines.edu/howto/HowToBeAProgrammer.html?x`.