ECE155: Engineering Design with Embedded Systems Winter 2015

Lecture 20 — Debugging II

Jeff Zarnett

Creating Test Cases

While debugging, we hopefully had a way to reproduce the bug consistently. This should be the basis for a test case. It should be possible to create a unit test (see our previous lectures). The inputs that led to the failure (bug) should be the input to the test case. The test should be added to your regression tests, so that the bug does not return.

In production software, however, sometimes the input given when debugging is huge (or complex), so we need to find a way to reduce the test case down to its minimal size. The GCC (GNU Compiler Collection) [FSF12] suggests the following reasons why we need to trim test cases:

- Privacy laws may require removal of proprietary data.
- More easily run if there are fewer dependencies.
- Smaller test cases make debugging easier.
- Easier for developers to work with.
- Easier to add to the regression tests.

The goal is to remove as much code as possible; the remaining code doesn't need to make logical sense. There are a number of different things to try, but always do them one at a time. After each change, run the test again and see if the failure still occurs. If it does not, undo the last changes and try something else. The GCC suggestions of what to try include:

- Remove function calls (replace with their return values or delete if they are `void`)
- Remove I/O (like logging) except where necessary to demonstrate the failure.
- Remove unused variables.
- Replace user-defined types (classes) with built-in types (`int`, `double`).
- Simplify class hierarchies: Remove base classes, or use base classes instead of derived ones.
- Remove: unused class variables/functions, unused data types, unneeded references to other classes.

Writing Debuggable Code

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> Brian Kernighan

The obvious lesson we should take away from this quote is: write code that is simple and easy to understand. It may be syntactically clever and might even be shorter, but if it's hard to read and follow, it's not a good idea. Code that makes sense to you may not make sense to a co-worker... or your future self a few weeks or months from now. There are some steps you can take to make your code easier to read and understand. Some may be familiar to you from ECE 150 or a co-op term.

Style. Whatever style you are using should be common to everyone in your team. If your coding style is consistent, it's easier to read the source written by other people (or yourself a long time ago).

Variable Names. Choosing good variable names takes taste, which you can develop. If a variable has a short lifetime, like a loop variable, then a short name is good. It's totally fine to write `for (int i = 0; i < 5; i++)`. If a variable is visible to a whole class or used in a calculation, you want a more descriptive name. Which of the following variable names is best for storing the total gross weight: `value`, `w`, `weight`, or `totalGrossWeight`? Typical programming style is to mash together all the words, with a capital letter at the start of each word except the first. So "well chosen variable name" becomes `wellChosenVariableName`.

Spread Out. Don't try to squeeze too much code into one line. Really long lines are herd to read. Humans, studies show, read columns much better than long lines of text. That's why newspapers (you know, the kind they make out of dead trees) have columns instead of long lines.

A trivial example, but compare:

```
if (x < 47 && y > 0) { z = 0;}
```

vs.

```
if (x < 47 && y > 0) {
    z = 0;
}
```

The second is easier to read, but also easier to debug. You can put a breakpoint now on the second line (`z = 0;` which will trigger only when the `x` and `y` conditions are met.

Use Temporary Variables for Complex Expressions. If faced with a very complex expression, it might be better to use a temporary variable for the sub-expressions, rather than try to put it all together at the end. Instead of:

```
return getBaseAmount() + getInsuranceCorrection() + getFreightCorrection();
```

try:

```
double sum = getBaseAmount() + getInsuranceCorrection() + getFreightCorrection();
return sum;
```

Once again, this makes debugging easier, because we can set a breakpoint at the creation of the sum and examine its value immediately before it is returned. If we're trying to track down an error, it's very tedious to have to enter each of the functions to see their values; instead we could make it easier to see by changing that to:

```
double baseAmount = getBaseAmount();
double insuranceCorrection = getInsuranceCorrection();
double freightCorrection = getFreightCorrection();
double sum = baseAmount + insuranceCorrection + freightCorrection;
return sum;
```

Yes, it's longer, but it's easier to debug.

Write Testable Code. Code that is testable will be easier to work with than code which is difficult to test. In particular, if you have a test for a piece of code and know it functions properly for a given set of input(s), then you can be reasonably sure that the problem lies somewhere else. If the invoice module is misbehaving for values above $1 000 000, and you have some tests which use a value like $10 000, you could see what happens if you replace the $10 000 with $1 000 000 in the input and expected value. But how actually to write testable code?

Example: one way is to minimize the use of `void` methods. Instead of `invoice.calculateAndAddTax();` you might instead write `Money tax = invoice.calculateTax(); invoice.setTax(tax);`. This produces two discrete, testable methods, where previously there was only one harder-to-test method. This also helps debugging, because you can put a breakpoint on the line `calculateTax()` and easily see the result of the tax calculation there.

Performance optimizations may also hinder debugging of the code, if it's the wrong kind. Putting in little shortcuts and clever tricks to get better performance might make the code that much harder to understand.

Comments. Some people think comments are necessary to understand code. Others are sure that comments are dangerous because they are misleading: someone wrote them, but maybe they are not maintained and no longer accurately describe what is going on. Wrong comments are worse than no comments at all. Write the code in such a way that it's "self-documenting"... that is, someone else can understand what it is doing easily.

Comments should be used to explain WHY something is done. Example:

```
//We multiply the numerator and denominator by 15/16 before the division takes place
```

vs.

```
//We multiply the numerator and denominator by 15/16 to work around the Pentium FDIV bug
```

Some people follow the rule "comment anything that is non-obvious". This is a tough standard, however, because what is obvious to one programmer is not necessarily obvious to another. In particular, a programmer who has spent a long time looking at a module or section of code has a good understanding of what it is and how it should work. If he or she comments only what is non obvious to him/her, and then quits the company, the next person to work on this module may be faced with code that is very difficult to understand. A potential solution? Code reviews. A co-worker at the same company will have the basic knowledge of the software but may not know the specifics of the code under review. The reviewer can point out areas that need comments to aid in their understanding.

What code does can generally be understood by analyzing it. Why it does so needs to be recorded. Some suggestions for what you might comment are: what a function is supposed to do, assumptions on interface usage, unexpected side effects, and workarounds (as above).

Documentation. In Java, there are tools that automatically generate documentation (Javadocs) from special code comments. In a way, documentation is closely related to commenting, but documentation does usually go beyond code. Still, it merits its own category, because documentation can help or hinder in debugging.

If the documents are crap, nobody will read them. A small amount of good documentation is best; although some people think that more documentation is always better When writing documentation, consider who will be reading the code later. As with comments, the best approach is to write self-explanatory code and then document the things that are in some way non-obvious [Rea03].

Documentation tends to face the same hazard as comments: it can be dangerous because the documentation can be misleading: someone wrote them, but maybe they were not updated and no longer accurately describe what is going on. Wrong documentation is worse than no documentation at all.

Create Logs. And finally, create log files. They will help to find out what has gone wrong and prevent it from happening again. Airplanes have "black boxes" (unrelated to the concept we discussed in testing) that record data at all times. In the event of a crash or other incident, this data is available for authorities to investigate what happened. Choose wisely what information should go in the log file: too much information and anything important will be lost in the noise; too little and you may not have enough pieces of the puzzle to reconstruct the incident.

The Golden Rules of Debugging

We'll now sum up our section on debugging with 13 golden rules of debugging [GHKW08]. These are some generally applicable hints that you should follow. If you are choosing to violate these rules, you'd better have a good reason.

1. Understand the requirements. Make sure you understand the requirements before you start. Is there a specification or other documentation? Maybe the behaviour of the software is expected (and there is no bug).

2. Make it fail. Create a test case to reproduce the failure. This will be needed to ensure the problem is fixed when the code is changed, can be used in the regression test, and helps you understand all factors that contribute to the failure.

Bug reports are a bit like eyewitness accounts: they are usually incomplete, conflicting, and contain interpretation as well as fact. Even if the users are trying their best to give a truthful and complete account, it will be necessary to sort through the the data and find the facts.

3. Simplify the test case As we saw last lecture, we need to make the test case simpler. This is to rule out irrelevant factors, reduce the runtime of the test case, and make the test case easier to understand.

4. Look at the right error message When looking at a whole pile of error messages, which one do we focus on first? A hint from when you look at compiler errors: the first error. Sometimes subsequent problems are caused by the first error putting the program in a corrupt state. Therefore, fix the problems in the order that they appear.

5. Check the environment Check what is otherwise seemingly obvious ("is it plugged in?" "is it turned on?" "have you tried turning it off and on again?!"). Check for things that matter in the environment: does the user have permissions on the operating system? Is the disk full? Is there enough memory?

6. Separate facts from interpretation Examine the case and examine the bug report. Find out what facts are really facts and which are interpretation. If a bug report says the process fails on a file with a name more than 32 characters long, is it the file name length, or is it the content of the file? The bug report might suggest it's the length of the name, but check and be sure.

7. Divide and conquer In a situation where multiple factors can play a role (e.g, some feature worked last week and now it doesn't), assuming it's not obvious what caused it, then it's necessary to divide and conquer: split the problem up into two or more smaller instances.

Consider some potential causes of the problem and how you might identify it. Did the source code change (the most likely cause)? Did you change or upgrade the libraries? Is the host system different in some way?

Start by looking at the version control history of your software. Have there been any changes to the area where things are going wrong? If one sticks out to you, you can always backtrack to just before

that commit and see if the problem still occurs. If you are really stuck, go back one step at a time until you get to a version you know worked. If you find the one commit that is responsible for the bug, then you can examine the diffs of that commit to find the bug.

If you have backed out all changes since last week and the system is still going off the rails, then you may have a good case for it to be an environment change.

8. Use the right tool. The debugger is not the only tool in your arsenal. There are memory debuggers, log files, static analysis tools, and even your eyes. Choose the tool that best suits the problem.

9. Make one change at a time. If you change something, check to see if it solved the problem. If it did not, revert it back before trying the next thing. If you discover something else wrong while debugging, make a note for later, but don't start on it now.

10. Keep an audit trail. Often you will encounter a bug in a function that takes multiple parameters. If you keep an audit trail you will know what you tried and what the results were. The audit trail is especially important when you are working with others to debug a problem or even working on a problem in more than one sitting. In fact, if a developer is interrupted by a phone call or meeting it results in a context switch which may cause the loss of some data in his or her memory.

11. Get a fresh perspective. If you're really stuck, try explaining the problem to someone else. Their fresh perspective might help you to think of something you have not thought of before. If they are an expert, they may have an alternate theory, or just insight into how you might modify an existing one.

Neat tip: sometimes explaining the problem to an inanimate object (like a rubber duck) helps. The act of trying to explain the problem out loud (even to an object incapable of listening) might help you solve the problem.

12. If you didn't fix it... If you changed some things and the bug went away, but you don't understand why, then it's possible you didn't really fix the bug at all. Your changes might merely have made the bug less likely to occur, or occur for a different set of inputs than the ones you were using before.

13. Create a regression test. The problem is fixed – excellent. Use the bug fix as the basis for a test to be added to your regression testing. Earlier, we went over the details about how to do this.

References

[FSF12] Inc. Free Software Foundation. How to Minimize Test Cases for Bugs, 2012. Online; accessed 16-May-2013.

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

[Rea03] Robert L. Read. How to be a Programmer: A Short, Comprehensive, and Personal Summary, 2003. Online; accessed 1-December-2013.