Lecture 20 — Debugging II

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Part I

Creating Test Cases

Creating Test Cases

Hopefully while debugging we had a way to reproduce the bug.

This is the basis for our test case.

Add the test case to your regression tests.

Trimming the Test Case

Some reasons why we might need to trim the test case down:

- Privacy laws may require removal of proprietary datae.
- 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.

Trimming the Test Case

We want to remove as much as possible.

The remaining code does not have to make sense.

There are many things to try, but do them one at a time.

If the test case no longer fails, undo your last change.

Trimming the Test Case

Suggestions about what to try to make the test case simpler:

- 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.

Part II

Writing Debuggable Code

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

Writing Debuggable Code

Write code that is simple and easy to understand.

Syntactically clever might be shorter, but it's hard to follow.

It might make sense now, but will it in a few months?

Code Style

Use a consistent style for all team members.

Easier to read the source written by others.

Including something you wrote 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.

Example: `for (int i = 0; i < 5; i++)`

Variable Names

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`?

Spread Out

Don’t try to squeeze too much code into one line.

Chicago Daily Tribune

THE WORLD'S GREATEST NEWSPAPER

HOME

DEWEY DEFEATS TRUMAN

G. O. P. Sweep Indicated in State; Boyle Leads in City

REPUBLICAN TICKET AHEAD OF 1944 VOTE

Town Balloting Gives Trend

Tops Coghlan in Hot Race for Attorney

RECORD CITY VOTE SEEN IN LATE TALLIES

Suburban Ballot Near 375,000

BULLETINS ON ELECTIONS

Early Count Gives G.O.P. Senate Edge

PUTS G.O.P. BACK IN THE WHITE HOUSE

Sizable Electoral Margin Seen

Spread Out

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.

Use Temporary Variables

Use a temporary variable for the sub-expressions.

Instead of:

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

try:

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

Write Testable Code

Code that is testable will be easier to work with than code which is difficult to test.

If a test on a section passes, you can save time and look somewhere else first.

You might be able to adapt a unit test to help you debug:

- Invoice module is misbehaving for values above $1 000 000
- You have some tests which use a value like $10 000

Write Testable Code: Avoid `void`

Example: minimize the use of `void` methods.

Replace the void `invoice.calculateAndAddTax();`

With:

```
Money tax = invoice.calculateTax();
invoice.setTax(tax);.
```

This produces two discrete, testable methods.

- Previously there was only one harder-to-test method.

Write Testable Code: Performance Optimizations

Performance optimizations can be a problem.

Shortcuts and clever tricks might be faster, but harder to understand.

Use them only if they are truly necessary.

Comments

One view is that comments are necessary.

Another is they are dangerous: misleading when out of date.

Wrong comments are worse than no comments at all.

Write code so it's "self-document" (easy for an expert to understand what is going on).

Comment: Non-Obvious

Another view is “comment anything non-obvious”.

Problem: obvious to one programmer is not obvious to others.

Potential solution: code reviews.

Comments

My view: 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
```

Comments

Other things we might comment:

- What the function is supposed to do.
- Assumptions.
- Parameters to a function.
- Bug fixes (noting the ticket number).
- Unexpected side effects.
- Workarounds (see previous slide).

Documentation

If the documents are crap, nobody will read them.

A small amount of good documentation is best.

Don’t waste time writing too much.

Like comments, try to write self-explanatory code;
write documents where necessary.

Documentation

Same problem as comments: might be out of date.

Wrong docs are worse than no docs at all.

Log Files

Create log files. They help in diagnosing problems.

Airplanes have “black boxes” that record flight data.

If there is a crash/incident, data is available to investigators.

Choose wisely how detailed the logging is.
- Too much: important data lost in the noise.
- Too little: insufficient data to reconstruct the incident.

Part III

The Golden Rules of Debugging

The Golden Rules of Debugging

Rule 1: *Understand the Requirements*

Before you begin: check the specification / documentation.

Maybe the software is already doing what it is supposed to do.

The Golden Rules of Debugging

Rule 2: *Make it Fail*

Create a test case to reproduce the failure.

Bug reports are like eyewitness accounts: incomplete, conflicting, and full of interpretation mixed with fact.

The Golden Rules of Debugging

Rule 3: *Simplify the Test Case*

Make the test case simpler:

- Rule out irrelevant factors.
- Reduce runtime.
- Make the case easier to understand.

The Golden Rules of Debugging

Rule 4: *Look at the Right Error Message*

Which error message do we look at?

The first one!

Subsequent problems are sometimes caused by the first error.

The Golden Rules of Debugging

Rule 5: *Check the Environment*

Check what is otherwise really obvious.
"Is it plugged in?" "Is it turned on?"
"Have you tried turning it off and on again?"

Is the disk full? Is there enough memory?

The Golden Rules of Debugging

Rule 6: *Separate facts from interpretation*

Examine the bug report and find out what is a fact.

Suppose the process fails on a file with a name > 32 chars.
- Is it the file name length?
- Or the content of the file?

The Golden Rules of Debugging

Rule 7: *Divide and Conquer*

Consider potential causes of the problem.
Examples: Source code change, library update, OS upgrade.

Check version control history.

If you revert all changes and the problem still happens – check the environment.

The Golden Rules of Debugging

Rule 8: *Use the Right Tool*

The debugger is not the only tool in your arsenal.

- Memory debuggers
- Log files
- Static analysis tools
- Your own eyes

The Golden Rules of Debugging

Rule 9: *Make One Change at a Time*

Make a change and test to see if it fixed it.

If it didn't, revert the change before you go on.

If you discover something else, note it for later.

The Golden Rules of Debugging

Rule 10: *Keep an Audit Trail*

Take notes on what you have tried and the results.

Important when you work with others.

Also important to resume after an interruption.

The Golden Rules of Debugging

Rule 11: *Get a Fresh Perspective*

If stuck, try explaining the problem to someone else.

They might have valuable input.

Weird tip: explaining it to a rubber duck might help.

The Golden Rules of Debugging

Rule 12: *If you didn't fix it...*

You changed something, the bug went away, but you don't know why...

Maybe you didn't fix the bug. Maybe it's hidden or occurs for different input.

The Golden Rules of Debugging

Rule 13: *Create a Regression Test*

Bug fixed? Excellent.

Use the fix as the basis for a regression test.