Lecture Notes
November 24, 2020

Volunteer demos: Iceberg, TeamShiba, Unleash Shirish

Working as a team means WORKING AS A TEAM!

I've heard reports that some students are not cooperating with their team members or completing their fair share of the project workload.

"Differential grading" will be applied to the team projects' overall scores. This means different members of the same team can get different scores.

Differential grading will be determined in part by team member responses on the second assessment and in part by the recommendations of the IAs. (Students with "problem" team members should discuss with their IA mentors asap!) A question similar to the following will be included in the second assessment. It will not be graded, but we will not grade the technical part of your assessment unless you provide a meaningful answer.

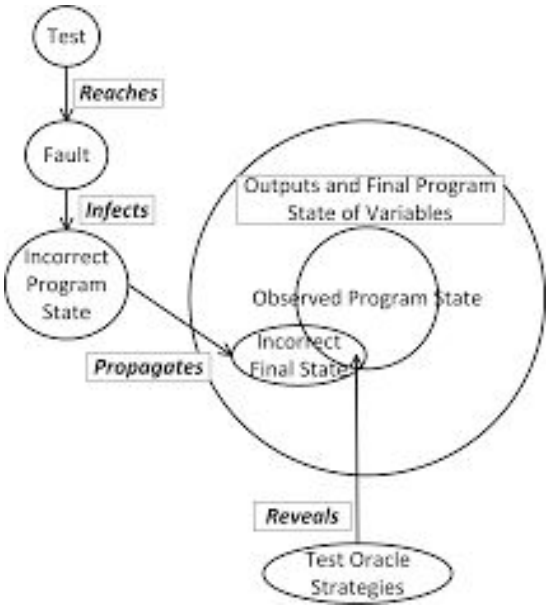
Allocate 100 points among the members of your team. Assign some number of points to each member of your team member, possibly 0, based on what you believe was that student's relative contribution to the team project and **explain your scores**. Remember to include yourself. Consider contributions to team meetings, software development and testing, writing the assignment submissions, setting up and presenting demos, anything else relevant to the team project.

For example, in a team where all team members did their fair share, the scores would be: Cora 20, Hannah 20, Kaedi 20, Kaylah 20, Nina 20 (this team has 5 members), and your explanation would describe who did which part(s) of the coding, testing, document writing, demo preparation, etc. Let's say the team score was A-: then all five team members would get A-.

But in a team with a substantial disparity in contributions, the scores might be: Iron Man 45, Captain America 40, Hulk 15, Thor 0 (this team has 4 members), and your explanation might describe Iron Man as the team leader, with Iron Man and Captain America doing most of the development. Hulk helped out with testing, but Thor did not do anything useful. Let's say the team score was A-: then Iron Man and Captain America would get at least A-, possibly A, Hulk might get B- or C+, and Thor would probably get D or F (depending on details).

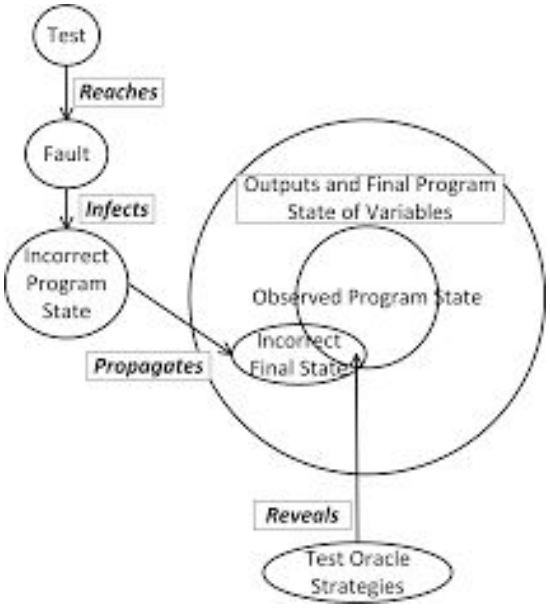
Working as a team: debugging

How does a system-level test suite actually detect bugs deep in a large codebase?

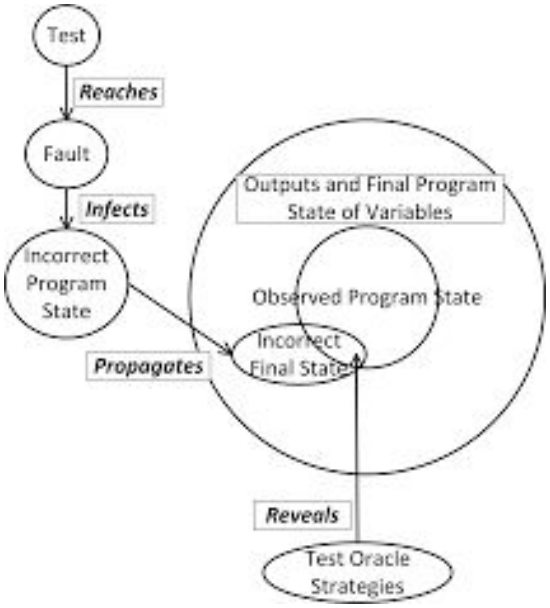


First, at least one test must *reach* the location (or locations) in the code containing the bug. Or reach the location(s) where the code ought to be, in cases where the bug is due to missing code - this could be anywhere in the program.

This is why structural (whitebox) test coverage is so important - if your test cases never reached part of the code, then your test cases never reached the bug in that part of the code (or could determine that none of your test cases reached the missing code, because it's missing). But your users will!



After that part of the code has been executed (or the entire program when code is missing), the state of the program must be incorrect with respect to achieving its responsibilities to the program design. The program state has been *infected*.



This infected program state must *propagate* through later code, if any, executed by the same test case, to cause some externally visible state or output to be incorrect.

Finally, the test case assertions (or human-checked logs, print statements, etc.) must *reveal* the incorrect aspects of the program state or output. Even if the buggy state is "visible", it won't actually be seen unless you look for it.

Testing *detects* bugs. Before you can fix a bug, you need to *localize* the bug. Not does it exist, but where is it?

Do not confuse with application "localization" or "internationalization", which concerns adaptation of UI and content to language, culture and locale

Localize bug = find coding mistake(s) in specific line(s) of code (or figure out where the missing code ought to be, which may be even harder and may require reconsideration of the design)

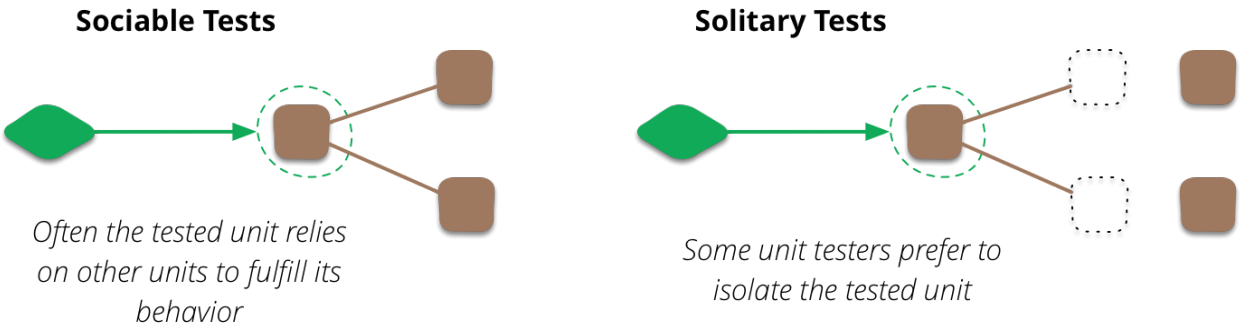
First step is get over idea that developer's code is right and the computer is wrong

Despite occasional news reports when a 20+ year old bug is finally noticed, bugs are exceedingly rare in hardware, compilers, interpreters, runtime environments, standard and widely-used frameworks and libraries - except perhaps in new releases, but large numbers of (other) users quickly shake them out

➔What developer told the computer to do was wrong

Six Stages of Debugging
1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work

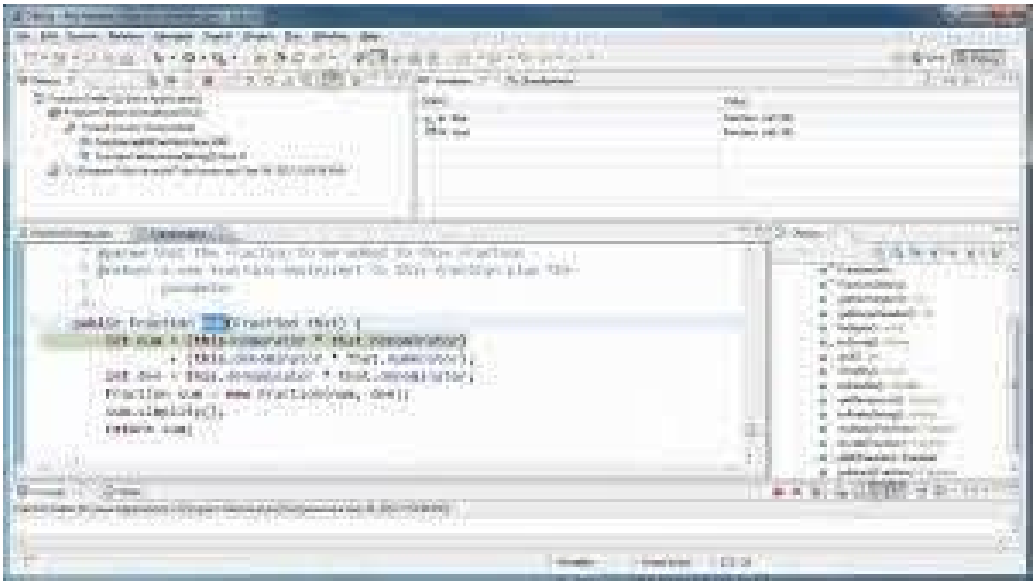
Best case (other than not having a bug at all): Unit test discovered the bug during testing on the developer's local machine prior to commit, so the developer knows the fault *manifests* in that unit as a failure, and the buggy unit has not propagated to other developers



If unit tests are isolated and independent, i.e., "solitary", then the buggy code (the underlying programming defect) *is also in that unit*

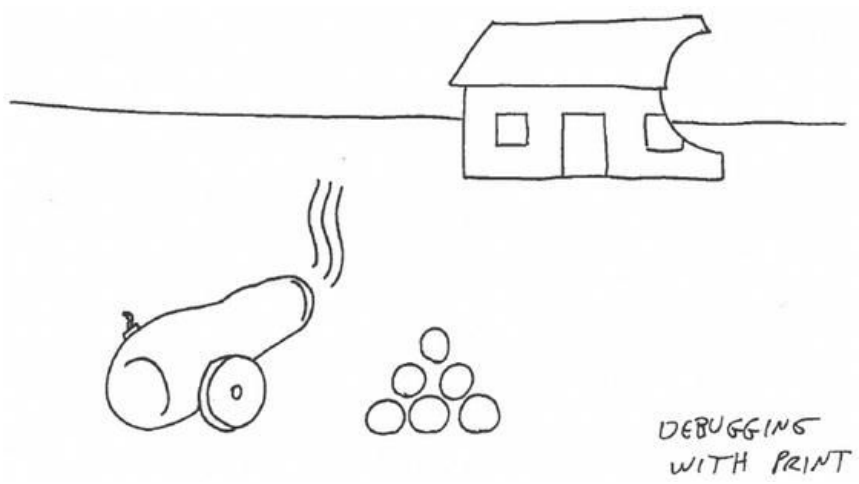
If using "sociable" unit testing, then the buggy code is still limited to a reasonably small number of units or the interfaces among those units. Maybe each of the units, in isolation, does its functionality correctly - but there is a mismatch in the assumptions made by the different units. Then there will be some lines of code, in one or both units, that embody those assumptions.

To locate buggy line(s) of code in a unit or small assembly, “best practice” is to use an interactive debugger, logging and log analyzer, and/or embedded assertions - but not print statements



What is wrong with print statement debugging? It’s what everyone learned first, it “worked”, why give it up?

Introducing print statements *changes* the code - modern languages are designed to prevent semantic changes due to built-in (or standard library) debugging statements, but cannot help if you introduce new code that cannot be distinguished from application code



This problem is most significant with C/C++ and other languages with unmanaged memory, but can occur in nearly every language.

Localizing a bug is a process of checking beliefs about the program. With a given input

- Developer might believe that a certain variable has a certain value (derived from input) at a certain place in the code
- Developer might believe that a function was called with particular parameters (derived by input)
- Developer might believe that a particular execution path (forced by input) is taken through conditionals

This is the main value of including assertions or logging statements within every method, because at the time the developer is writing the method they are usually aware of their beliefs regarding parameters, local variables, global state, expected return values of system/library calls

Guides interactive debugging to set breakpoints, single-step, examine variables, etc. in those units to double-check those beliefs

Debugging with or without debugging tools is easiest in best case (unit testing), that’s why I called it the best case

Ok case: All unit tests pass but integration testing fails

What does this tell us?

Bad case: All integration and unit tests pass but system testing fails

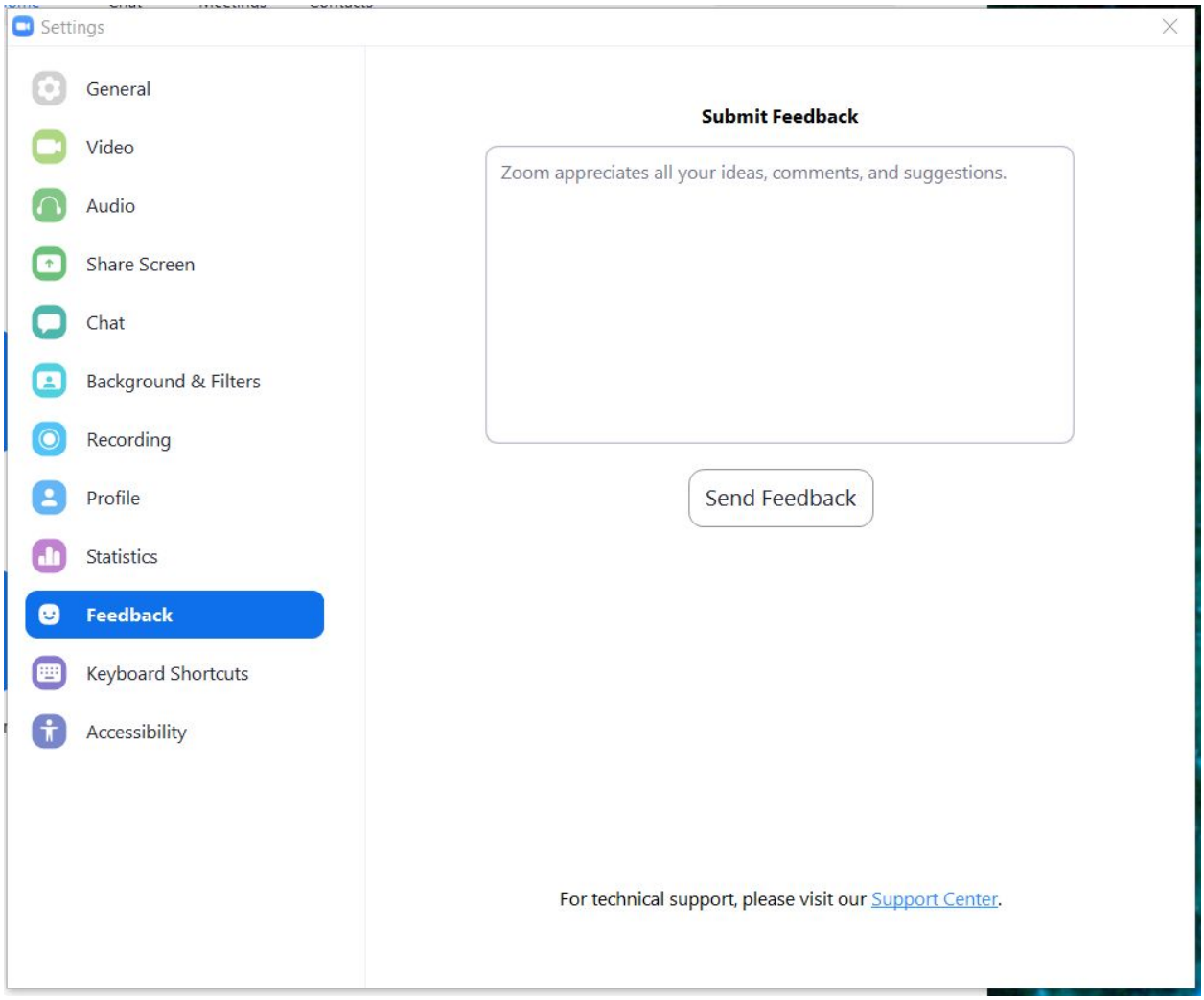
What does this tell us?

Rushed through rest, skipped some material, continue from here next lecture

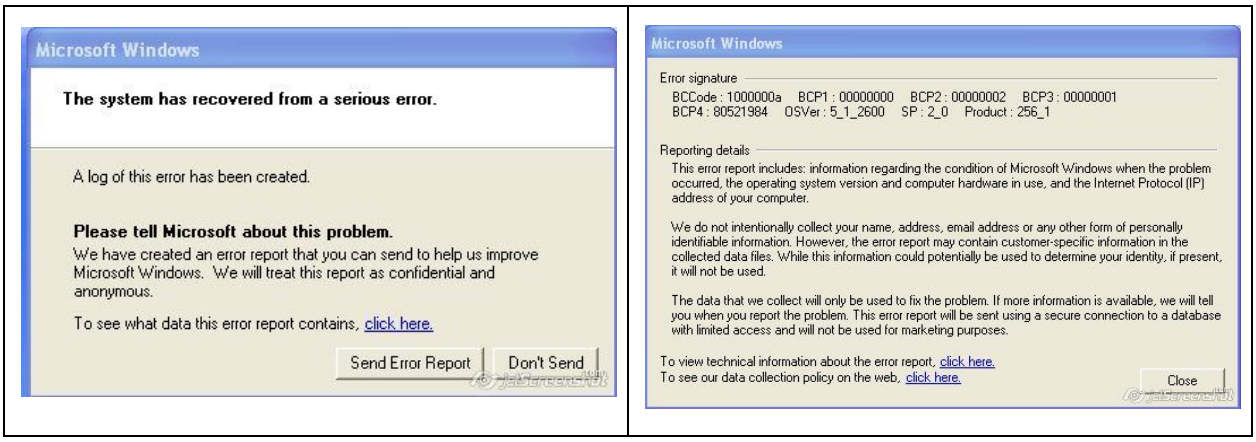
Worst case: Customer detects a bug after deployment, which usually means the developer test suite is inadequate to reveal that bug

Before developer can debug in such case, they need to create a new test case that reliably detects the bug

But customer may not be able to tell you step by step details to reproduce (or isn't asked to)



Some bug reports may be accompanied by a stack trace, “core dump”, or other details automatically embedded into the error report by the program, rather than relying on the user



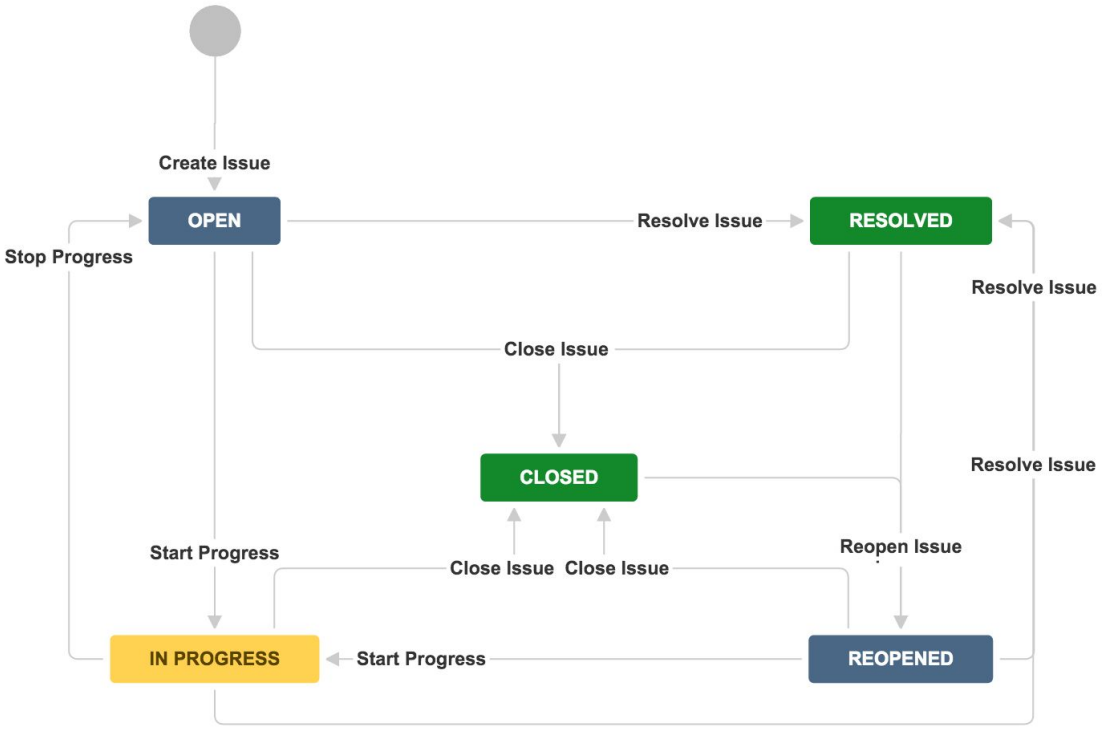
This material typically aims to support attaching a debugger “post-mortem”, so the developer can reason backwards from application state to try to construct a test case that reproduces the bug

If the bug report is accompanied by a record/replay log, then the replay faithfully reproduces the bug (similar but not quite the same as a test case). This enables the developer to attach a debugger to the replay, and single-step, breakpoint, etc. as if the program were running “live”.

If developers cannot reproduce the bug, the bug report might be “closed” as non-reproducible - possibly reopened if other users report same failure, or marked “wontfix” or similar designation

When end-users submit bug reports, or “tickets”, these are normally triaged and cleaned up by technical support staff or developers before formal posting to the development team’s issue tracking system (such as Jira or Bugzilla). Bug reports from internal testers and developers would probably be directly posted with the issue tracker

Typical workflow:



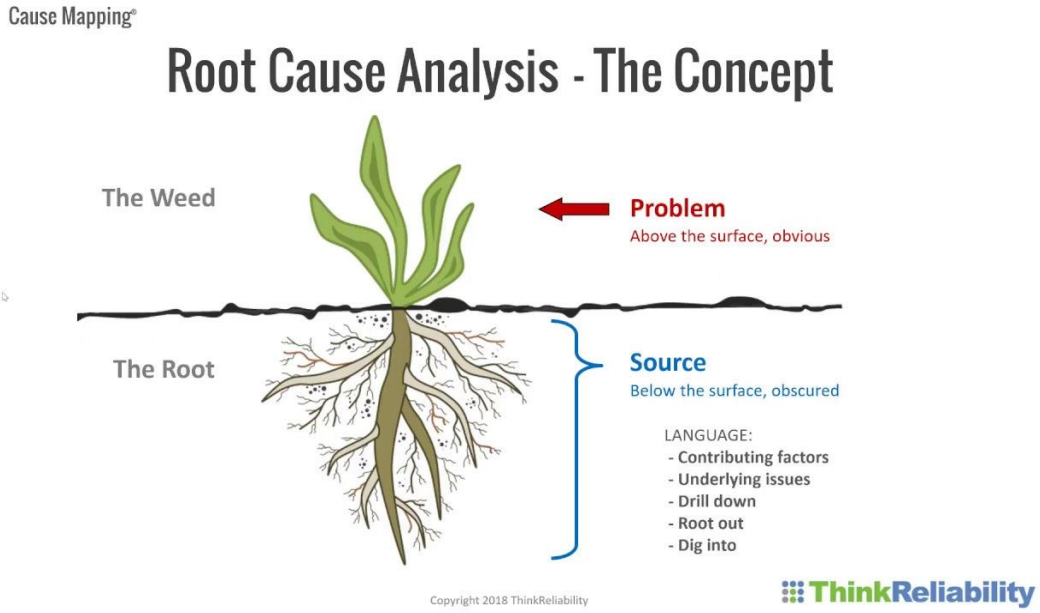
What does a good bug report look like? See templates at https://marker.io/blog/bug-report-template/

Example in github (most past issues have been closed): https://github.com/gmu-swe/phosphor/issues

Another example in github (many issues remain open) https://github.com/clowdr-app/clowdr-web-app/issues

Once you have some input that triggers the bug, you want to find the root cause.

Root Cause Analysis sometimes refers to treating the programming defect as a symptom, and seeking a more fundamental underlying cause in requirements, design, developer training, software process, etc.



But I just mean the buggy lines of code.

One approach that makes it easier to find the root cause is “spectrum-based fault localization”. This is a fancy way of saying 1. keep track of which code units had bugs in the past; 2. check which of those units, if any, are exercised during the failing test case(s); 3. look at those units in order of which had the most previous bugs.

Another technique that helps find the root cause is "*delta debugging*" = winnow down to smallest, simplest test case (*input)* that causes the same erroneous behavior as the failing test case(s) that detected the error

This is most useful when test inputs are 'large', e.g., document or other media, data table, stream of network packets, video game save

A 'large input' might be provided in an end-user bug report, since the user uses real-world inputs that matter to them, not tests crafted for triggering corner cases

Search for 'smallest' input by systematically removing parts of the known bug-triggering 'large' input and checking whether the remainder still causes the bug

Then start detailed debugging with that 'smallest' input (it may not be *unique*)

Question to think about: After you have discovered which line(s) of code are buggy, how do you figure out how to fix the bug?

Assignment T5: Second Iteration: https://courseworks2.columbia.edu/courses/104335/assignments/490853 due December 4

Assignment T6: Final Demo: https://courseworks2.columbia.edu/courses/104335/assignments/491047 due December 10

Extra Credit: Optional Demo Video https://courseworks2.columbia.edu/courses/104335/assignments/543277 due December 20