

Lecture Notes
November 1, 2018

Testing *detects* bugs. How to *localize* the bugs?

Do not confuse with application localization
(internationalization/i18n), which concerns adaptation
of UI and content to culture and locale

Localize bug = find coding mistake(s) in specific line(s)
of code, so can fix



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

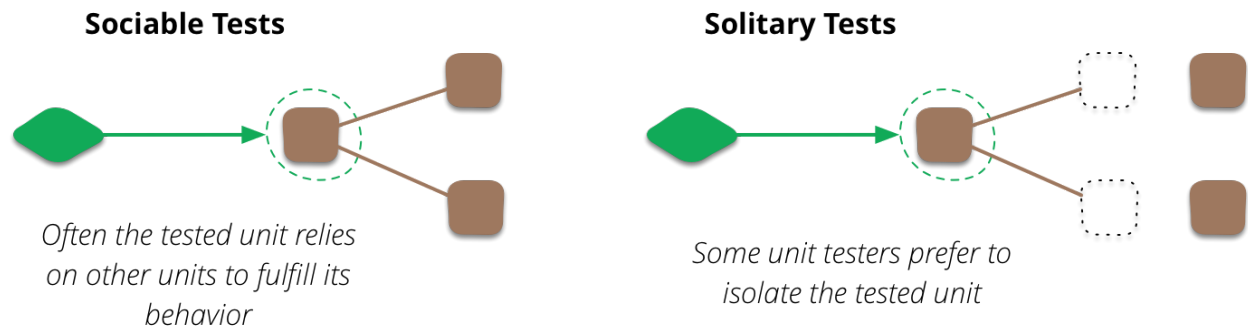
→ What developer told the computer to do was wrong



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: Unit test discovered the bug, during pre-commit CI, so developer knows fault *manifests* in that unit as a failure, and buggy unit has not propagated to other developers

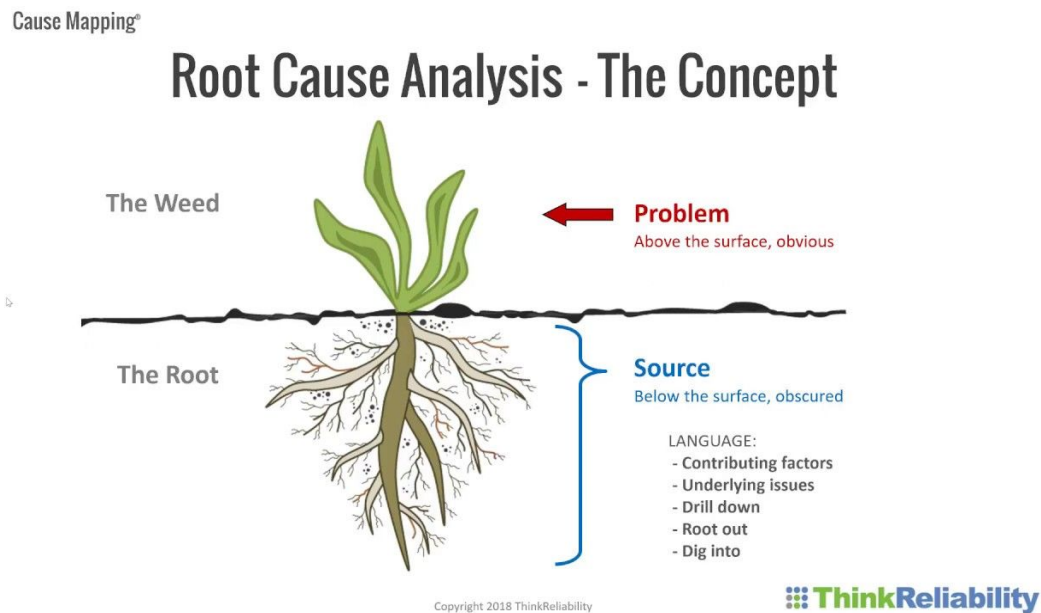


If unit tests are isolated and independent, i.e., "solitary", then the root cause (the underlying programming defect) *is also in that unit*

If using "sociable" unit testing, then the root cause is still limited to a reasonably small number of units or the interfaces among those units

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

But I just mean the buggy line(s) of code



Finding root cause is made easier with "[delta debugging](#)" - find smallest, simplest test case (*input*) that causes the same erroneous behavior

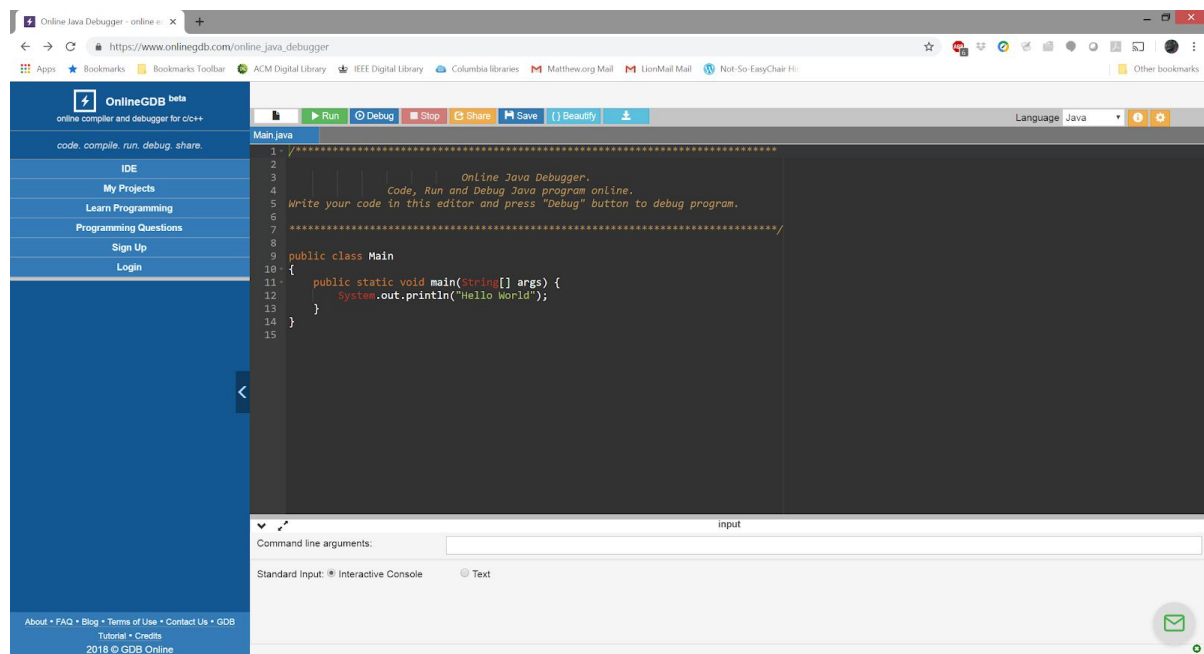
Most applicable when test inputs are "large", e.g., document in word processor, HTML in browser, cells in spreadsheet, avatars in action video game

Arguably, unit test inputs should have been small in the first place, but many faults are not triggered by small inputs, only large inputs. Also often useful to provide unit tests with "real" data (e.g., previously included in end-user bug report or supplied by beta tester)

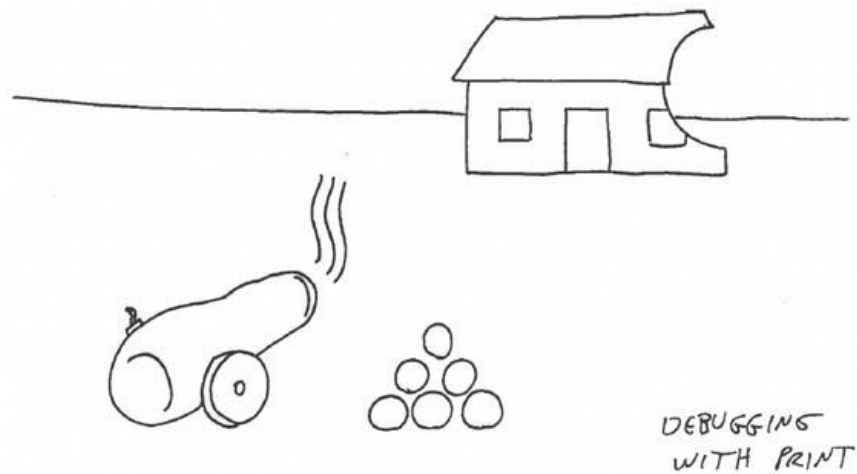
Search for "smallest" input by systematically removing parts and checking whether remainder still causes bug

Then start detailed debugging with that smallest input: does not need to be unique - smallest means if any more removed, the failure no longer occurs

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



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



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 developer is writing the method she usually is aware of her 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

Demo: eclipse debugger (Anthony)

Debugging with or without debugging tools is easiest in best case, that's why I called it the best case

Ok case: Integration testing fails during CI but all unit tests pass

What does this tell us?

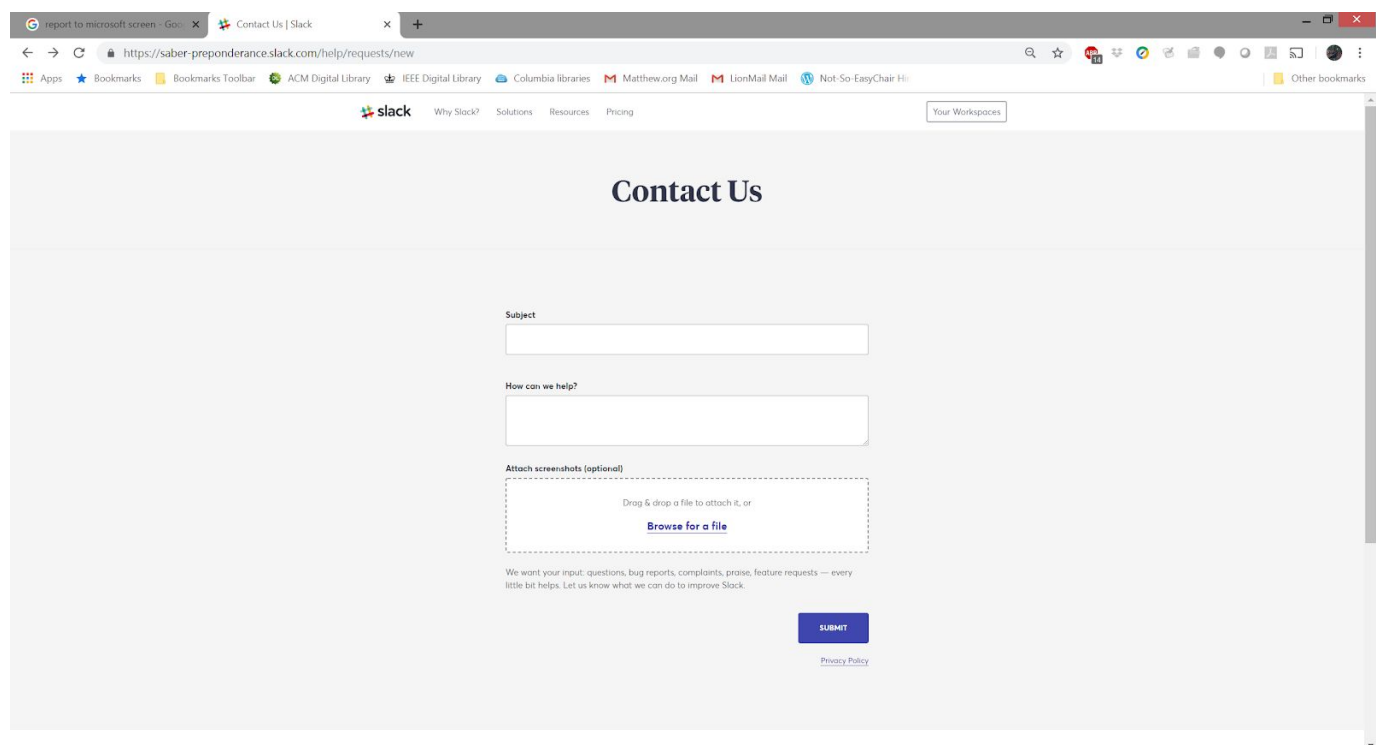
Bad case: System testing fails but all integration and unit tests pass

What does this tell us?

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

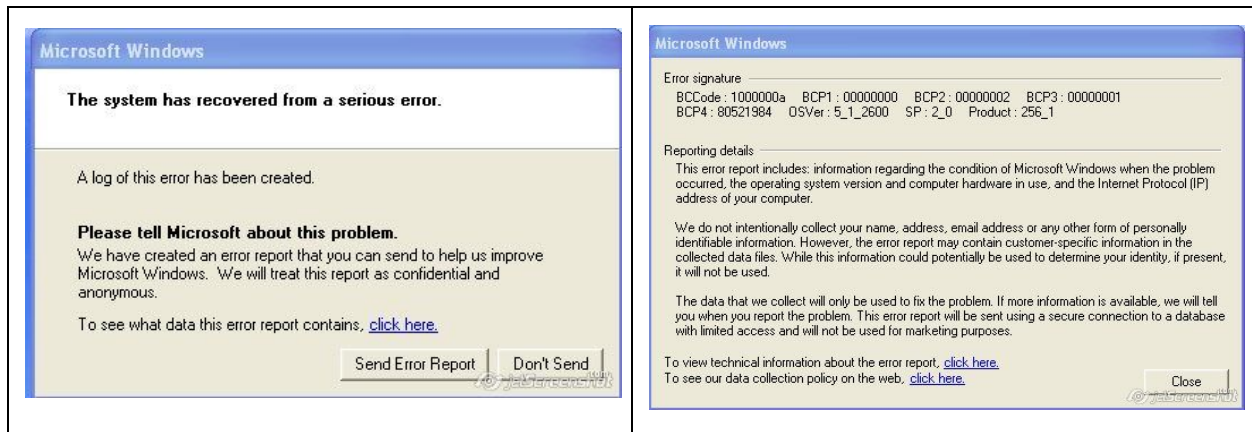
Before developer can debug, needs a [new test case that reliably detects the bug](#)

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

A screenshot of a web browser showing the Slack 'Contact Us' form. The browser's address bar displays the URL 'https://saber-preponderance.slack.com/help/requests/new'. The page has a light gray background with the Slack logo and navigation links at the top. The main heading is 'Contact Us'. Below it, there are three input fields: 'Subject', 'How can we help?', and 'Attach screenshots (optional)'. The 'Attach screenshots (optional)' field includes a dashed border and a 'Browse for a file' link. At the bottom, there is a 'SUBMIT' button and a 'Privacy Policy' link. The browser's bookmark bar is visible at the top, showing various links like 'Apps', 'Bookmarks', 'ACM Digital Library', and 'IEEE Digital Library'.

"We want your input: questions, bug reports, complaints, praise, feature requests — every little bit helps. Let us know what we can do to improve Slack."

Some bug reports may be accompanied by a stack trace, "core dump", record/replay log, or other details automatically embedded into the error report by your product, rather than relying on the user

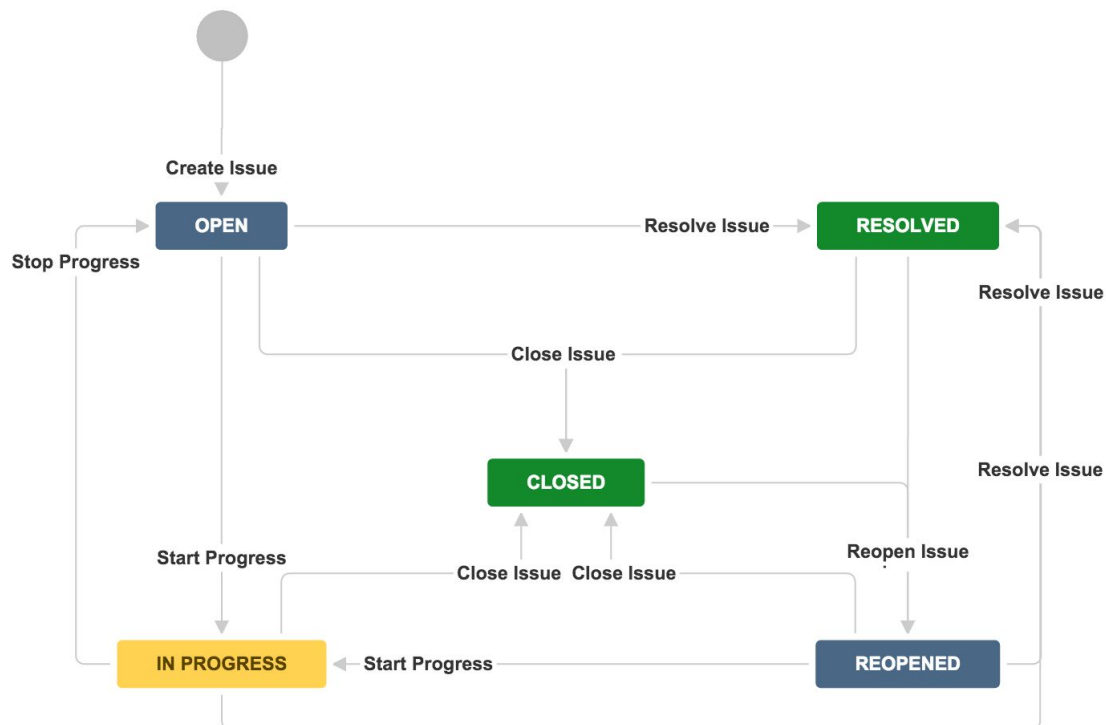


This material typically aims to support attaching a debugger "post-mortem", so can reason backwards from application state to try to construct a test case that reproduces the bug

If developers cannot reproduce the bug, the bug report might be "closed" as non-reproducible - probably 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 (all past issues have been closed):

<https://github.com/gmu-swe/phosphor/issues?q=is%3Aissue+is%3Aclosed>

Example (many issues remain open)

<https://github.com/EvoSuite/evosuite/issues>

If your team is using github, use their "issue tracker" to record bugs and other problems as they arise

Is anyone not using github?

If not using github or another repository with its own issue tracker, there are other no-cost issue trackers. For example, see [Capterra reviews](#), but most require you to self-host. The free version of [backlog](#) is cloud-hosted and sufficient for team projects in this course, but allows only one project. [Trello](#) or similar might also work, even though not intended as an issue tracker

No class Tuesday - university holiday

Reading for next Thursday: [7 things I learnt in my first job after graduation from university](#)

Guest lecture next Thursday: [Prof. Baishakhi Ray](#)

[First Iteration Demo](#) due next week on Thursday, November 8, 11:59pm. Please plan (human script) your demo in advance, and debug until it works smoothly

Next pair assignment [Bug Hunt 2](#) due Tuesday, November 20, 10:10am