

Lecture Notes

December 1, 2020

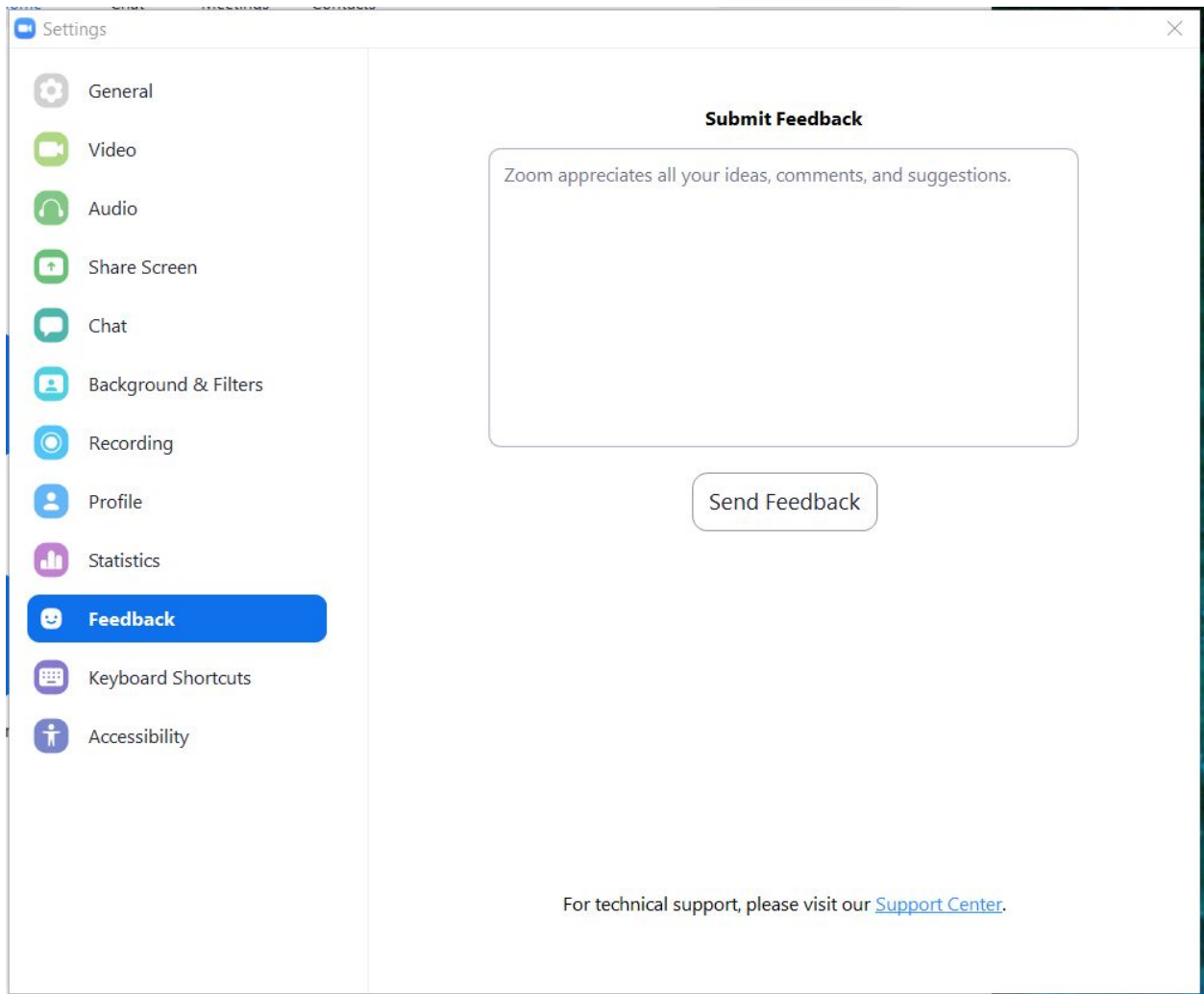
We started this segment last week but rushed through and skipped parts, so resuming from here.

Say the user detects a bug after deployment. This usually means the developer test suite is inadequate to reveal that bug (not always, may be a case of “[wontfix](#)”)

If there is no existing developer test that detects this bug, the developer needs to create a [new test case that reproduces and reliably detects the bug](#) as the starting point for debugging.

But the user [may not be able to report the step by step details to reproduce](#) (or isn't asked to)

This is not a good form for reporting bugs



The image shows a screenshot of the Zoom application's 'Settings' window, specifically the 'Feedback' section. The left sidebar contains a list of settings categories: General, Video, Audio, Share Screen, Chat, Background & Filters, Recording, Profile, Statistics, Feedback (highlighted in blue), Keyboard Shortcuts, and Accessibility. The main content area is titled 'Submit Feedback' and contains a text input field with the placeholder text 'Zoom appreciates all your ideas, comments, and suggestions.' Below the input field is a 'Send Feedback' button. At the bottom of the window, there is a link to the 'Support Center' for technical support.

Settings

- General
- Video
- Audio
- Share Screen
- Chat
- Background & Filters
- Recording
- Profile
- Statistics
- Feedback**
- Keyboard Shortcuts
- Accessibility

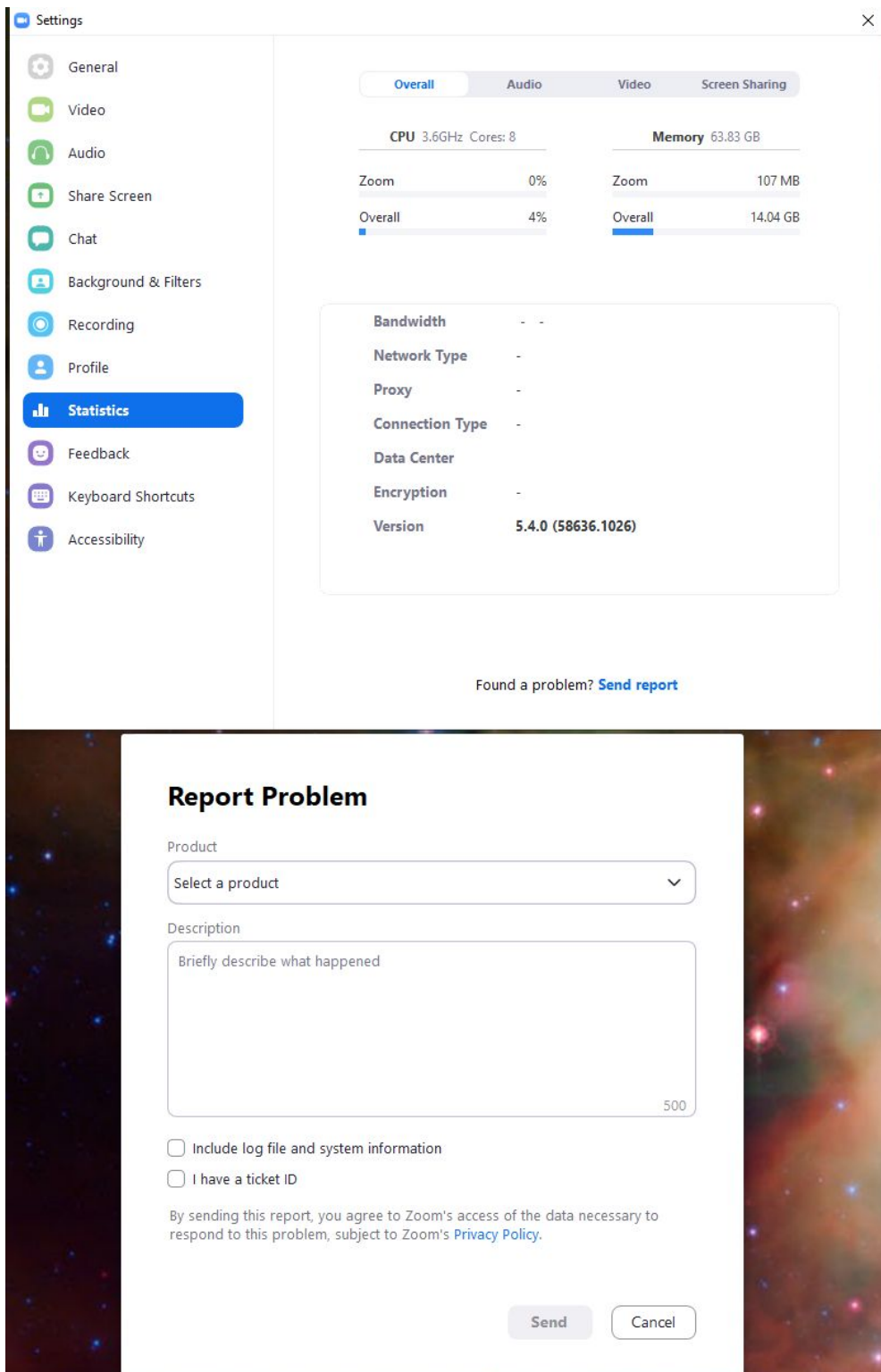
Submit Feedback

Zoom appreciates all your ideas, comments, and suggestions.

Send Feedback

For technical support, please visit our [Support Center](#).

This is better



The image shows a screenshot of the Zoom application interface. The top part displays the 'Settings' window, specifically the 'Statistics' tab. The left sidebar lists various settings categories: General, Video, Audio, Share Screen, Chat, Background & Filters, Recording, Profile, Statistics (highlighted), Feedback, Keyboard Shortcuts, and Accessibility. The main content area of the 'Statistics' tab shows performance metrics for CPU and Memory, with sub-tabs for Overall, Audio, Video, and Screen Sharing. Below these, a box lists system information: Bandwidth, Network Type, Proxy, Connection Type, Data Center, Encryption, and Version (5.4.0 (58636.1026)). At the bottom of the settings window is a link to 'Send report'.

Found a problem? [Send report](#)

The bottom part of the image shows a 'Report Problem' dialog box. It contains a 'Product' dropdown menu, a 'Description' text area with a 500-character limit, and two checkboxes: 'Include log file and system information' and 'I have a ticket ID'. A disclaimer states: 'By sending this report, you agree to Zoom's access of the data necessary to respond to this problem, subject to Zoom's [Privacy Policy](#).' At the bottom are 'Send' and 'Cancel' buttons.

Report Problem

Product
Select a product

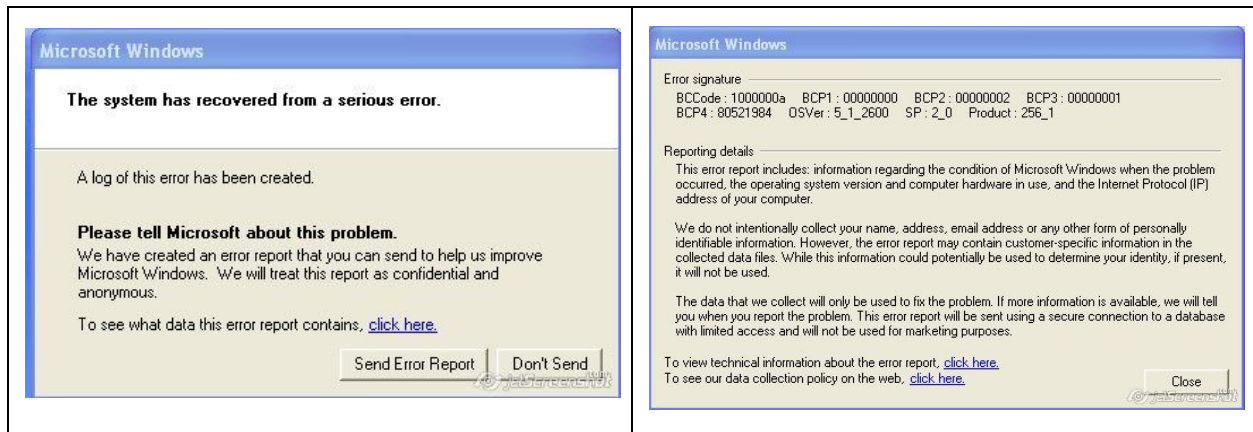
Description
Briefly describe what happened
500

☐ Include log file and system information
☐ I have a ticket ID

By sending this report, you agree to Zoom's access of the data necessary to respond to this problem, subject to Zoom's [Privacy Policy](#).

Send Cancel

Some bug reports may be accompanied by a stack trace, “core dump”, or other details [automatically embedded into the bug report by the program](#), rather than relying on the user’s memory of what happened



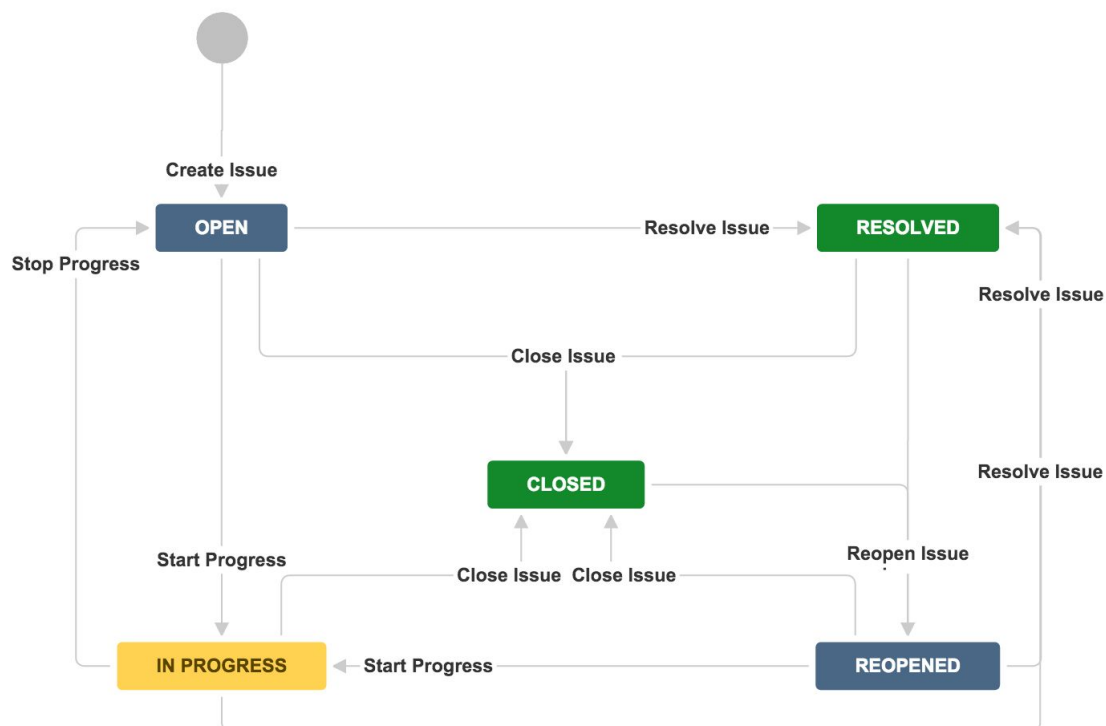
This bug report material typically aims to support attaching a debugger “post-mortem”, so the developer can reason backwards from application state to try to construct a test case that reproduces the bug

If the bug report is accompanied by a [record/replay log](#), then the replay faithfully reproduces the bug. This enables the developer to attach a debugger to the replay, and single-step, breakpoint, etc. as if the program were running “live”. If you can faithfully replay the program state and externally visible manifestations leading up to a bug, do you still need a test case?

If developers cannot reproduce the bug, the bug report might be closed as non-reproducible - possibly reopened if other users report the same or similar 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:



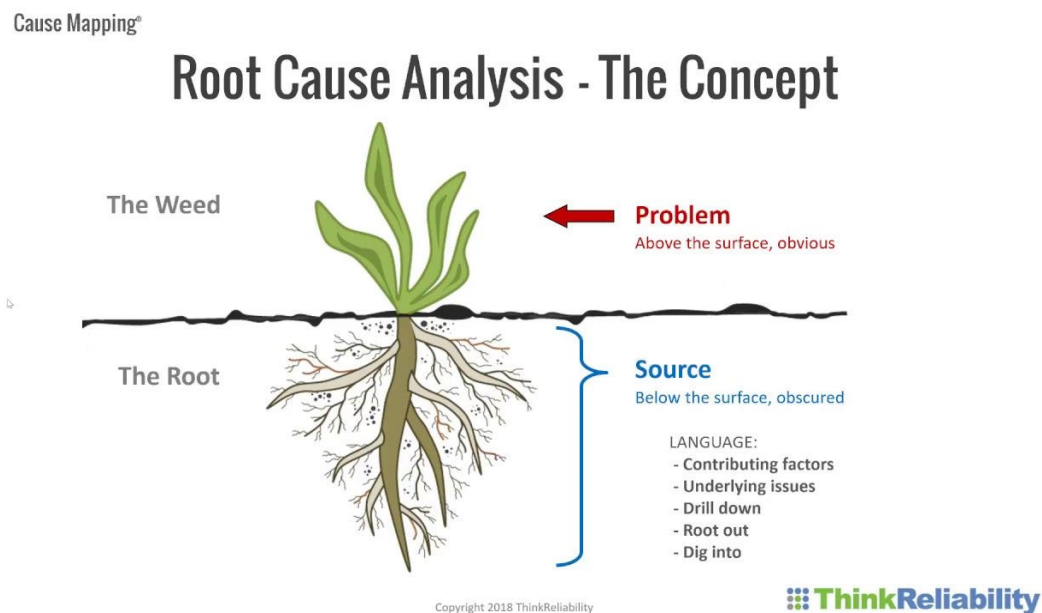
What does a good bug report look like? See templates at <https://marker.io/blog/bug-report-template/>

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 the developer can reproduce the bug, the next step is to locate the root cause = bug localization

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 sometimes makes it easier to find the root cause is “spectrum-based fault localization”. (Also known as “[Round up the usual suspects](#)”.) 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 both the current and *previous* (historical) failing test case(s); 3. rank those units in order of which had the most previous bugs; 4. examine the units in order starting from the highest ranked.

This approach is appropriate for software that already has a substantial history

“Statistical debugging” is another approach to searching for the root cause. This just means: 1. keep track of which parts of the code are exercised by *multiple* (current, not past) failing tests; 2. consider the parts in a hierarchy: packages, classes, methods, basic blocks, individual statements; 3. rank those parts in order of which were exercised by the most failing tests; 4. examine the parts in order starting from the highest ranked at smallest granularity.

This approach may be helpful when there are large numbers of currently failing tests, not just one

Another technique that may help find the root cause is “[delta debugging](#)”. Delta refers here to difference.

Systematically remove portions of a known bug-triggering input and check whether the remainder still causes the bug. The goal is to find the simplest such input (or one of the simplest - not necessarily unique)

This is most useful when initial test inputs are ‘large’, e.g., document or other media, data table, stream of network packets, video game save. These often come from user bug-reports or other real-world data. Large refers to the size of structured data, not an individual value

Teams need to apply several testing techniques to complete “Assignment T5: Second Iteration”

<https://courseworks2.columbia.edu/courses/104335/assignments/490853> (originally due this Friday Dec 4, extended to Sunday Dec 6)

Define the [equivalence partitions](#), both valid and invalid, for each of your major methods and devise corresponding unit tests for those methods.

There is often more than one invalid equivalence class. For example, consider the equivalence classes for a string. How would you figure out what its equivalence classes ought to be?

Remember to group tests that should be run together, in a specific order, with the corresponding [test fixture](#)

For those equivalence partitions that have [boundaries](#) (not all do), determine what specifically are those boundaries and devise unit tests for exactly at the boundary, one less and one more, for each boundary.

Boundaries (and equivalence classes) can refer to the *size* of data, not just its *value*

Note: The term “[boundary validation](#)” refers to checking untrusted external data (potentially from a hacker) entering a system. “Boundary conditions” also means something different in the context of differential equations.

Say you have a string `myString` that is received from some external entity (not necessarily a human user) during the execution of a method `foo()`.

```
void foo () {  
    some code  
  
    myString = call-API(params);  
  
    some more code  
}
```

Breakout room exercise (~7 minutes):

How would you figure out the valid and invalid equivalence classes for myString, for the purpose of unit-testing `foo()` ?

What about it's boundary conditions? Strings almost always have boundaries.

How would you implement those unit test cases?

Say the API call returned a serialized data structure, such as a tree, rather than a string. After reconstructing (deserializing) it, how would you test its equivalence classes?

Set up your branch coverage tool before running your test suite, since many branches will be covered while running regular blackbox tests - including integration and system tests, not just unit tests. Remember that [statement coverage does not imply branch coverage](#).

If you do not already achieve 100% branch coverage, try to devise new test cases that will cover missed branches.

Say you have a branch like this one

```
void foo () {  
    lots of code  
    x = outside-call(y);  
  
    if ( x > y ) { do this } else { do that }  
  
    lots more code  
}
```

and the coverage tool reports that the false branch has been exercised (i.e., “do that”) but not the true branch (“do this”). Note `foo()` has no parameters.

How would you write a test case that forces the branch to be true?

Continuous integration - include configuration file(s) in your github repo and submit something as part of your iteration report to demonstrate that the CI works

Assignment T6: Final Demo:

<https://courseworks2.columbia.edu/courses/104335/assignments/491047> still due December 10

Seeking volunteer teams to do demos in class on either Tuesday December 8th or Thursday December 10 (last two class sessions). You will still need to do a separate demo for your IA mentor to give your IA a chance to “drive”.

More Extra Credit: Optional Demo Video

<https://courseworks2.columbia.edu/courses/104335/assignments/543277> still due December 20

- Any questions about spring E6156 Topics in Software Engineering?