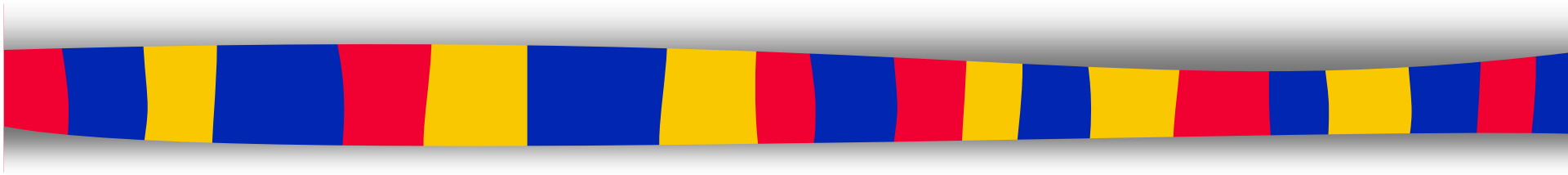
EECS 4313

Software Engineering Testing



Topic 02:

Reporting and analyzing bugs

- How to communicate efficiently to developers

Zhen Ming (Jack) Jiang

Failure, fault, error and incident

- Failure
  - Observable incorrect **behavior** of a program. Conceptually relate to the behavior of the program, rather than its code
    - A failure occurs when a fault executes
- Fault (bug)
  - Related to the **code**. Necessary (not sufficient) condition for the occurrence of a failure
    - A fault is the result of an error
    - A fault won't yield a failure without the **conditions** that trigger it.
    - Example: if the program yields 2+2=5 on the 10th time you use it, you won't see the error before or after the 10th use.
- Error
  - **Cause** of a fault. Usually a human error (conceptual, typo, etc.)
    - "People make errors"
- Incident
  - the symptom associated with a failure that alerts the user to the occurrence of a failure

Bug reporting

- Testers report bugs to programmers
- Problem Report forms are commonly used
- If the report is not clear and understandable, the bug will not get fixed
- To write a fully effective report you must:
  - Explain how to reproduce the problem
  - Analyze the problem so that it can be described with a minimum number of steps
  - Write a report that is complete, easy to understand, and non-antagonistic

What kind of error to report?

- Report all the following types of problems, but keep straight in your mind, and on the bug report, which type you're reporting.
  - Coding Error: The program does not do what the programmer would expect it to do.
  - Design Issue: It's doing what the programmer intended, but a reasonable customer would be confused or unhappy with it.
  - Requirements Issue: The program is well designed and well implemented, but it won't meet one of the customer's requirements.
  - Documentation / Code Mismatch: Report this to the programmer (via a bug report) and to the writer (usually via a memo or a comment on the manuscript).
  - Specification / Code Mismatch: Sometimes the spec is right; sometimes the code is right and the spec should be changed.

Bug Reports

- A **bug report** is a tool that you use to sell the programmer on the idea of spending his/her time and energy to fix a bug.
- Bug reports are your **primary work product** as a tester. This is what people outside of the testing group will most notice and most remember of your work.
- The best tester is not the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed.

Selling Bugs

- Time is in short supply. If you want to convince the programmer to spend his time fixing your bug, you may have to sell him on it.
- Selling revolves around two fundamental objectives:
  - Motivate the buyer (Make him WANT to fix the bug.)
  - Overcome objections (Get past his excuses and reasons for not fixing the bug.)

Motivating the Bug Fixer

- Some things that will often make programmers want to fix the bug:
  - It looks really bad.
  - It looks like an interesting puzzle and piques the programmer's curiosity.
  - It will affect lots of people.
  - Getting to it is trivially easy.
  - It has embarrassed the company, or a bug like it embarrassed a competitor.
  - Management (that is, someone with influence) has said that they really want it fixed.

Motivating the Bug Fix

- When you run a test and find a failure, you are looking at a symptom, not at the underlying fault. You may or may not have found the best example of a failure that can be caused by the underlying fault.
- Therefore you should do some follow-up work to try to prove that a defect:
  - **is more serious than it first appears.**
  - **is more general than it first appears.**

Look for follow-up errors

- When you find a coding error, you have the program in a state that the programmer did not intend and probably did not expect. There might also be data with supposedly impossible values.
- The program is now in a vulnerable state. Keep testing it and you might find that the real impact of the underlying fault is a much worse failure, such as a system crash or corrupted data.

Types of follow-up testing

- Vary the behaviour (change the conditions by changing what the test case does)
- Vary the options and settings of the program (change the conditions by changing something about the program under test).
- Vary the configuration (software and hardware environment).

1. Vary Your Behaviour

- Keep using the program after you see the problem.
- Bring it to the failure case again (and again). If the program fails when you do X, then do X many times. Is there a cumulative impact?
- Try things that are related to the task that failed. For example, if the program unexpectedly but slightly scrolls the display when you add two numbers, try tests that affect adding or that affect the numbers. Do X, see the scroll. Do Y then do X, see the scroll. Do Z, then do X, see the scroll, etc. (If the scrolling gets worse or better in one of these tests, follow that up, you're getting useful information for debugging.)
- Try things that are related to the failure. If the failure is unexpected scrolling after adding, try scrolling first, then adding. Try repainting the screen, then adding. Try resizing the display of the numbers, then adding.
- Try entering the numbers more quickly or changing the speed of your activity in some other way.
- Also try other exploratory testing techniques. For example, you might try some interference tests. Stop the program or pause it just as the program is failing. Or try it while the program is doing a background save. Does that cause data loss corruption along with this failure?

2. Vary Options and Settings

- In this case, the steps to achieve the failure are taken as given. Try to reproduce the bug when the program is in a different state:
  - Change the values of environment variables.
  - Change anything that looks like it might be relevant that allows you to change as an option.
- For example, suppose the program scrolls unexpectedly when you add two numbers. Maybe you can change the size of the program window, or the precision (or displayed number of digits) of the numbers

3. Vary the Configuration

- A bug might show a more serious failure if you run the program with less memory, a higher resolution printer, more device interrupts coming in etc.
  - If there is anything involving timing, use a really slow (or very fast) computer, network connection, printer etc.
  - If there is a video problem, try other resolutions on the video card. Try displaying MUCH more (less) complex images.
- We are interested in whether there is a particular configuration that will show the bug more spectacularly.
- Returning to the example (unexpected scrolling when you add two numbers), try things like:
  - Different video resolutions
  - Different mouse settings if your mouse has a scrolling wheel

Is this bug new to this version?

- In many projects, an old bug (from a previous release of the program) might not be taken very seriously if there were not lots of customer complaints.
  - If you know it's an old bug, check its history.
  - The bug will be taken more seriously if it is new.
  - You can argue that it should be treated as new if you can find a new variation or a new symptom that did not exist in the previous release. What you are showing is that the new version's code interacts with this error in new ways. That's a new problem.

Motivating the Bug Fix: Show it is More General

- Look for configuration dependence
- Bugs that do not fail on the programmer's machine are much less credible (to that programmer). If they are configuration dependent, the report will be much more credible if it identifies the configuration dependence directly (and so the programmer starts out with the expectation that it won't fail on all machines).

Configuration dependence

- In the ideal case (standard in many companies), test on 2 machines
  - Do your main testing on Machine 1. Maybe this is your powerhouse: latest processor, newest updates to the operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, fast connection etc.
  - When you find a defect, use Machine 1 as your bug reporting machine and replicate on Machine 2. Machine 2 is totally different. Different processor, different keyboard and keyboard driver, different video, barely enough RAM, slow, small hard drive, dial-up connection with a link that makes turtles look fast.
- Some people do their main testing on the slow machine and use the more powerful machine for replication.
- Write the steps, one by one, on the bug form at Machine 1. As you write them, try them on Machine 2. If you get the same failure, you have checked your bug report while you wrote it. (A valuable thing to do.)
- If you do not get the same failure, you have a configuration dependent bug. Time to do troubleshooting. But at least you know that you have to.

Uncorner your corner cases

- We test at extreme values because these are the most likely places to show a defect. *But once we find the defect, we do not have to stick with extreme value tests.*
  - Try mainstream values. These are easy settings that should pose no problem to the program. Do you replicate the bug? If yes, write it up, referring primarily to these mainstream settings. This will be a very credible bug report.
- If the mainstream values don't yield failure, but the extremes do, then do some troubleshooting around the extremes.
  - Is the bug tied to a single setting (a true corner case)?
  - Or is there a small range of cases? What is it?
  - In your report, identify the narrow range that yields failures. The range might be so narrow that the bug gets deferred. That might be the right decision. Your reports help the company choose the right bugs to fix before a release, and size the risks associated with the remaining ones.

Overcoming Objections: Analysis of the Failure

- Things that will make programmers resist spending their time on the bug:
  - The programmer cannot replicate the defect.
  - Strange and complex set of steps required to induce the failure.
  - Not enough information to know what steps are required, and it will take a lot of work to figure them out.
  - The programmer does not understand the report.
  - Unrealistic (e.g., “corner case”)
  - It’s a feature.

Non-Reproducible Errors (1)

- Always report non-reproducible errors. If you report them well, programmers can often figure out the underlying problem.
- You must describe the failure as precisely as possible. If you can identify a display or a message well enough, the programmer can often identify a specific point in the code that the failure had to pass through.
- When you realize that you can't reproduce the bug, write down everything you can remember. Do it now, before you forget even more.
- As you write, ask yourself whether you're sure that you did this step (or saw this thing) exactly as you are describing it. If not, say so. Draw these distinctions right away. The longer you wait, the more you will forget.

Non-Reproducible Errors (2)

- Maybe the failure was a delayed reaction to something you did before starting this test or series of tests. Before you forget, note the tasks you did before running this test.
- Check the bug tracking system. Are there similar failures? Maybe you can find a pattern.
- Find ways to affect timing of the program or devices, slow down, speed up.
- Talk to the programmer and/or read the code.

Non-Reproducible bugs are reproducible

- Failures occur under certain conditions
- If you know the conditions, you can recreate a failure
- If you don’t know the critical conditions, you cannot recreate the failure
- What are some reasons you cannot reproduce a failure?

Reasons for non-reproducible bugs (1)

- Some problems have delayed effects:
  - a memory leak might not show up until after you cut and paste 20 times.
  - stack corruption might not turn into a stack overflow until you do the same task many times.
  - a wild pointer might not have an easily observable effect until hours after it was mis-set.
- If you suspect that you have time-delayed failures, use tools such as videotape/screen recording, capture programs, debuggers, debug-loggers, or memory meters to record a long series of events over time.

Reasons for non-reproducible bugs (2)

- The bug depends on the value of a hidden input variable. In any test, there are the variables that we think are relevant, and there is everything else. If the data you think are relevant don't help you reproduce the bug, ask what other variables were set, and what their values were.
- Some conditions are hidden and others are invisible. You cannot manipulate them and so it is harder to recognize that they're present. You might have to talk with the programmer about what state variables or flags get set in the course of using a particular feature.

Reasons for non-reproducible bugs (3)

- Some conditions are catalysts. They make failures more likely to be seen. Example: low memory for a leak; slow machine for a race. But sometimes catalysts are more subtle, such as use of one feature that has a subtle interaction with another.
- Some bugs are predicated on corrupted data. They don’t appear unless there are impossible configuration settings in the config files or impossible values in the database. What could you have done earlier today to corrupt this data?

Reasons for non-reproducible bugs (4)

- The bug might appear only at a specific time of day or day of the month or year. Look for week-end, month-end, quarter-end and year-end bugs, for example.
- Programs have various degrees of data coupling. When two modules use the same variable, oddness can happen in the second module after the variable is changed by the first. In some programs, interrupts share data with main routines in ways that cause bugs that will only show up after a specific interrupt.

Reasons for non-reproducible bugs (5)

- The program may depend on one version of a DLL. A different program loads a different version of the same DLL into memory. Depending on which program is run first, the bug appears or does not.
- The bug depends on you doing related tasks in a specific order.
- The bug is caused by an error in error-handling. You have to generate a previous error message or bug to set up the program for this one.

Reasons for non-reproducible bugs (6)

- The program might be showing an initial state bug, such as:
  - The bug appears only the first time after you install the program (so it happens once on every machine.)
  - The bug appears once after you load the program but won't appear again until you exit and reload the program.

Reasons for non-reproducible bugs (7)

- You forgot some of the details of the test you ran, including the critical one(s) or you ran an automated test that lets you see that a crash occurred but does not tell you what happened.
- The bug depends on a crash or exit of an associated process.
- The problem might appear only under a peak load, and be hard to reproduce because you cannot bring the heavily loaded machine under debug control (perhaps it is a customer's system).

Reasons for non-reproducible bugs (8)

- On a multi-tasking or multi-user system, look for spikes in background activity.
- The bug occurred because a device that it was attempting to write to or read from was busy or unavailable.
- It might be caused by keyboard keybounce or by other hardware noise.

Reasons for non-reproducible bugs (9)

- The apparent bug is a side-effect of a hardware failure.
  - A flaky power supply creates irreproducible failures.
  - One prototype system had a high rate of irreproducible firmware failures. Eventually, these were traced to a problem in the building's air conditioning. The test lab was not being cooled, no fan was blowing on the unit under test, and prototype boards in the machine ran very hot. The machine was failing at high temperatures.

Incomprehensible bug reports

- Programmers will not spend time on a bug if the bug report:
  - Has a strange and complex set of steps required to induce the failure.
  - Does not have enough information to know what steps are required, and it will take a lot of work to figure them out.
  - Is hard to understand.

Reporting Errors

- As soon as you run into a problem in the software, fill out a Problem Report form. In a well-written report, you:
  - Explain how to reproduce the problem.
  - Analyze the error so you can describe it in a minimum number of steps.
  - Include all the steps.
  - Make the report easy to understand.
  - Keep your tone neutral and non-antagonistic.
  - Keep it simple: one bug per report.
  - If a sample test file is essential to reproducing a problem, reference it and attach the test file.

The Problem Report Form (1)

- A typical form includes many of the following fields
  - **Problem report number**: must be unique
  - **Reported by**: original reporter's name. Some forms add an editor's name.
  - **Date reported**: date of initial report
  - **Program (or component) name**: the visible item under test
  - **Release number**: like Release 2.0
  - **Version (build) identifier**: like version C or version 20000802a

The Problem Report Form (2)

- **Configuration(s)**: h/w and s/w configurations under which the bug was found and replicated
- **Report type**: e.g., coding error, design issue, documentation mismatch, suggestion, query
- **Can reproduce**: yes / no / sometimes / unknown. (Unknown can arise, for example, when the configuration is at a customer site and not available to the lab).
- **Severity**: assigned by tester. Some variation on small / medium / large

The Problem Report Form (3)

- **Priority**: assigned by programmer/project manager
- **Problem summary**: 1-line summary of the problem
- **Keywords**: use these for searching later, anyone can add to key words at any time
- **Problem description and how to reproduce it**: step by step reproduction description
- **Suggested fix**: leave it blank unless you have something useful to say
- **Status**: Tester fills this in. Open / closed / resolved

The Problem Report Form (4)

- **Resolution**: The project manager owns this field. Common resolutions include:
  - **Pending**: the bug is still being worked on.
  - **Fixed**: the programmer says it's fixed. Now you should check it.
  - **Cannot reproduce**: The programmer can't make the failure happen. You must add details, reset the resolution to Pending, and notify the programmer.
  - **Deferred**: It's a bug, but we'll fix it later.
  - **As Designed**: The program works as it's supposed to.
  - **Need Info**: The programmer needs more info from you. She has probably asked a question in the comments.
  - **Duplicate**: This is just a repeat of another bug report (XREF it on this report.) Duplicates should not close until the duplicated bug closes.
  - **Withdrawn**: The tester withdrew the report.

The Problem Report Form (5)

- **Resolution version**: build identifier
- **Resolved by**: programmer, project manager, tester (if withdrawn by tester), etc.
- **Resolution tested by**: originating tester, or a tester if originator was a non-tester
- **Change history**: date-stamped list of all changes to the record, including name and fields changed.

The Problem Report Form (6)

- **Comments**: free-form, arbitrarily long field, typically accepts comments from anyone on the project. Testers, programmers, tech support (in some companies) and others have an ongoing discussion of reproduction conditions etc., until the bug is resolved. Closing comments (why a deferral is OK, or how it was fixed for example) go here.
  - This field is especially valuable for recording progress and difficulties with difficult or politically charged bugs.
  - Write carefully. It's easy to read a joke or a remark as a flame. Never flame.

Important Parts of the Report: Problem Summary

- This one-line description of the problem is the most important part of the report.
  - The project manager will use it when reviewing the list of bugs that haven't been fixed.
  - Executives will read it when reviewing the list of bugs that won't be fixed. They might only spend additional time on bugs with "interesting" summaries.

Problem Summary

- The ideal summary gives the reader enough information to help her decide whether to ask for more information. It should include:
  - A brief description that is specific enough that the reader can visualize the failure.
  - A brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug)?
  - Some other indication of the severity (not a rating but helping the reader envision the consequences of the bug.)

Can You Reproduce The Bug?

- You may not see this on your form, but you should always provide this information.
  - Never say it's reproducible unless you have recreated the bug. (Always try to recreate the bug before writing the report.)
  - If you have tried and tried but you can't recreate the bug, say "No". Then explain what steps you tried in your attempt to recreate it.
  - If the bug appears sporadically and you don't yet know why, say "Sometimes" and explain.
  - ***You may not be able to try to replicate some bugs.*** Example: customer-reported bugs where the setup is too hard to recreate.

How to Reproduce the Bug (1)

- First, describe the problem. Don't rely on the summary to do this - some reports will print this field without the summary.
- Next, go through the steps that you use to recreate this bug.
  - Start from a known place (e.g. boot the program)
  - Then describe each step until you hit the bug.
  - NUMBER THE STEPS. Take it one step at a time.
  - If anything interesting happens on the way, describe it. (You are giving people directions to a bug. Especially in long reports, people need landmarks.)

How to Reproduce the Bug (2)

- Describe the erroneous behaviour and, if necessary, explain what should have happened. (Why is this a bug? Be clear.)
- List the environmental variables (e.g., configuration) that are not covered elsewhere in the bug tracking form.
- If you expect the reader to have any trouble reproducing the bug (special circumstances are required), be clear about them.

How to Reproduce the Bug (3)

- It is essential to keep the description focused
- The first part of the description should be the shortest step-by-step statement of how to get to the problem.
- Add “Notes” after the description such as:
  - Comment that the bug won’t show up if you do step X between step Y and step Z.
  - Comment explaining your reasoning for running this test.
  - Comment explaining why you think this is an interesting bug.
  - Comment describing other variants of the bug.

Keeping the Report Simple

- If you see two failures, write two reports.
- Combining failures creates problems:
  - The summary description is typically vague. You say words like “fails” or “doesn’t work” instead of describing the failure more vividly. This weakens the impact of the summary.
  - The detailed report is typically lengthened and contains complex logic like: “Do this unless that happens in which case don’t do this unless the first thing, and then the testcase of the second part and sometimes you see this but if not then that”.
  - Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and therefore more intimidating.
  - You’ll often see one bug get fixed but not the other.
  - When you report related problems on separate reports, it is a courtesy to cross-reference them.

Keeping it Simple: Eliminate Unnecessary Steps (1)

- Sometimes it's not immediately obvious what steps can be dropped from a long sequence of steps in a bug.
  - *Look for critical steps -- Sometimes the first symptoms of a failure are subtle.*
- You have a list of the steps you took to show the error. You're now trying to shorten the list. Look carefully for any hint of a failure as you take each step -- A few things to look for:
  - Error messages (you got a message 10 minutes ago. The program did not fully recover from the error, and the problem you see now is caused by that poor recovery.)
  - Delays or unexpectedly fast responses.
  - Display oddities, such as a flash, a repainted screen, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, etc.

Keeping it Simple: Eliminate Unnecessary Steps (2)

- Sometimes the first indicator that the system is working differently is that it sounds a little different than normal.
- An in-use light or other indicator that a device is in use when nothing is being sent to it (or a light that is off when it shouldn't be).
- Debug messages—turn on the debug monitor on your system (if you have one) and see if/when a message is sent to it.

- If you have found what looks like a critical step, try to eliminate almost everything else from the bug report. Go directly from that step to the last one (or few) that shows the bug. If this does not work, try taking out individual steps or small groups of steps.

Put Variations After the Main Report

- Suppose that the failure looks different under slightly different circumstances. For example, suppose that:
  - The timing changes if you do two additional sub-tasks before hitting the final reproduction step
  - The failure won't show up or is much less serious if you put something else at a specific place on the screen
  - The printer prints different garbage (instead of the garbage you describe) if you make the file a few bytes longer
- This is all useful information for the programmer and you should include it. But to make the report clear:
  - Start the report with a simple, step-by-step description of the shortest series of steps that you need to produce the failure.
  - Identify the failure. (Say whatever you have to say about it, such as what it looks like or what impact it will have.)
  - Then add a section that says "ADDITIONAL CONDITIONS" and describe, one by one, in this section the additional variations and the effect on the observed failure.

Unrealistic cases

- Some reports are inevitably dismissed as unrealistic (having no importance in real use).
  - If you're dealing with an extreme value, do follow-up testing with less extreme values.
  - Check with people who might know the customer impact of the bug:
    - -- Technical marketing
    - -- Human factors
    - -- Network administrators
    - -- In-house power users
    - -- Technical support
    - -- Documentation
    - -- Training
    - -- Maybe sales

It’s not a bug, it’s a feature

- An argument over whether something is or is not a bug is really an argument about the oracle you should use to evaluate your test results.
- An **oracle** is the principle or mechanism by which you recognize a problem.
- "Meets the specification" or "Meets the requirements" is a heuristic oracle.
- If you know it’s "wrong" but you don't have a mismatch to a spec, what can you use?

Some useful oracle heuristics

- **Consistent with History**: Present function behaviour is consistent with past behaviour.
- **Consistent with our Image**: Function behaviour is consistent with an image that the organization wants to project.
- **Consistent with Comparable Products**: Function behaviour is consistent with that of similar functions in comparable products.
- **Consistent with Claims**: Function behaviour is consistent with documented or advertised behaviour.
- **Consistent with User's Expectations**: Function behaviour is consistent with what we think users want.
- **Consistent within Product**: Function behaviour is consistent with behaviour of comparable functions or functional patterns within the product.
- **Consistent with Purpose**: Function behaviour is consistent with apparent purpose.

Editing Bug Reports

- Some groups have a second tester (usually a senior tester) review reported defects before they go to the programmer. The second tester:
  - checks that critical information is present and intelligible
  - checks whether she can reproduce the bug
  - asks whether the report might be simplified, generalized or strengthened.
- If there are problems, she takes the bug back to the original reporter.