Reporting and analyzing bugs

slides: João Neto

Bug Reporting

- Testers report bugs to programmers
- 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 error 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 to Report

- Report all following problem types but be clear which type is it:
  - Coding Error, the program doesn't 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 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 to the programmer (via a bug report) and to the writer (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 the idea of spending 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 isn’t 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 Bug Reports

- 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.
- Sales 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)

Selling Bug Reports

- 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.

Selling Bug Reports

- Use appropriate terms that will ease the understating of the report
- Describe the symptoms that allowed you to detect the bug (failure)
- Pinpoint the context/conditions that led to it
- Try to make the problem as reproducible as you can

Follow-up

- Do some follow-up work to check if:
  - is more serious than it first appears
  - Is more general than it first appears

- When finding a coding error, the program is in an error state, i.e., something 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 behavior (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 software and hardware environment.

Vary the behavior

- 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.)

Vary the behavior

- 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?

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.
- 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

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, link, modem or 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 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 you have a wheel mouse that does semi-automated scrolling

Bug curricula

- In many projects, an old bug (from a previous release of the program) might not be taken very seriously if there weren't 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 didn't 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

- Look for configuration dependence
  - Do your main testing on Machine 1. Maybe this is your powerhouse: latest processor, newest updates to the operating system, good video card, huge SSD, lots of RAM, 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…

Motivating the Bug Fix

- Some people do their main testing on the turtle and use the power 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've checked your bug report while you wrote it. (A valuable thing to do.)
- If you don't get the same failure, you have a configuration dependent bug. Time to do troubleshooting. At least you know that you must do.

Motivating the Bug Fix

- Bugs that don’t 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.)

Overcoming Objections: Analysis of the Failure

- Things that will make programmers resist spending their time on the bug:
  - The programmer can’t 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 doesn’t understand the report.
  - Unrealistic (e.g. “corner case”)
  - It’s a feature.

Overcoming Objections: 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

- 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 20200214a

The Problem Report Form

- A typical form includes many of the following fields:
    - Configuration(s): hardware and software 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: small / medium / large
    - Priority: assigned by programmer/project manager

The Problem Report Form

- A typical form includes many of the following fields:
    - Customer impact: often left blank. When used, typically filled in by tech support or someone else predicting actual customer reaction (such as support cost or sales impact)
    - 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

- A typical form includes many of the following fields:
  - 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 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 (reference it on this report.) Duplicates should not close until the duplicated bug closes.
    - Withdrawn: The tester withdrew the report.

Non-Reproducible Errors

- 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'll forget.

Non-Reproducible Errors

- 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.

Reasons for non-reproducible bugs

- Some problems have delayed effects:
  - a memory leak might not show up until after you cut and paste several 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 video capture programs, debuggers, or debug-loggers to record a long series of events over time.

Reasons for non-reproducible bugs

- 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 while using a feature.

Reasons for non-reproducible bugs

- 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

- The bug might appear only at a specific time of day or day of the month or year. Look for weekend, 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.
- Special cases appear in the code because of time or space optimizations or because the underlying algorithm for a function depends on the specific values fed to the function.
- The bug depends on you doing related tasks in a specific order.
- The bug is caused by an error in error-handling. You must generate a previous error message or bug to set up the program for this one.

Bug tracking systems

- There are many software for tracking bugs
  - Bugzilla, RedMine, Mantis, …
  - Usually, they require complex configurations
- We are going to use an online alternative
  - https://backlog.com/
  - Free plan (1 project, up to 10 members)
  - Some will create project `vvs2122_XX` and assign me as member
    - Extra component in the evaluation of project 2