Informatica Inglese

READING COMPREHENSION EXERCISE

DEBUGGING

(Adapted from: The Pragmatic Programmer: From Journeyman to Master Andrew Hunt)

Debugging

It is a painful thing To look at your own trouble and know That you yourself and no one else has made it

• Sophocles, Ajax

The word bug has been used to describe an "object of terror" ever since the fourteenth century. Rear Admiral Dr. Grace Hopper, the inventor of COBOL, is credited with observing the first computer bug - literally, a moth caught in a relay in an early computer system. When asked to explain why the machine wasn't behaving as intended, a technician reported that there was "a bug in the system," and dutifully taped it - wings and all - into the log book.

Regrettably, we still have "bugs" in the system, albeit not the flying kind. But the fourteenth century meaning—a bogeyman—is perhaps even more applicable now than it was then. Software defects manifest themselves in a variety of ways, from misunderstood requirements to coding errors. Unfortunately, modern computer systems are still limited to doing what you tell them to do, not necessarily what you want them to do.

No one writes perfect software, so it's a given that debugging will take up a major portion of your day. Let's look at some of the issues involved in debugging and some general strategies for finding elusive bugs.

Psychology of Debugging

Debugging itself is a sensitive, emotional subject for many developers. Instead of attacking it as a puzzle to be solved, you may encounter denial, finger pointing, lame excuses, or just plain apathy.

Embrace the fact that debugging is just problem solving, and attack it as such.

Having found someone else's bug, you can spend time and energy laying blame on the filthy culprit who created it. In some workplaces this is part of the culture, and may be cathartic. However, in the technical arena, you want to concentrate on fixing the problem, not the blame.

It doesn't really matter whether the bug is your fault or someone else's. It is still your problem.

A Debugging Mindset

The easiest person to deceive is one's self (Edward Bulwer-Lytton, The Disowned)

Before you start debugging, it's important to adopt the right mindset. You need to turn off many of the defences you use each day to protect your ego, tune out any project pressures you may be under, and get yourself comfortable. Above all, remember the first rule of debugging: Don't Panic.

It's easy to get into a panic, especially if you are facing a deadline, or have a nervous boss or client breathing down your neck while you are trying to find the cause of the bug. But it is very important to step back a pace, and actually think about what could be causing the symptoms that you believe indicate a bug.

If your first reaction on witnessing a bug or seeing a bug report is "that's impossible," you are plainly wrong. Don't waste a single neuron on the train of thought that begins "but that can't happen" because quite clearly it can, and has.

Beware of myopia when debugging. Resist the urge to fix just the symptoms you see: it is more likely that the actual fault may be several steps removed from what you are observing, and may involve a number of other related things. Always try to discover the root cause of a problem, not just this particular appearance of it.

Where to Start

Before you start to look at the bug, make sure that you are working on code that compiled cleanly—without warnings. We routinely set compiler warning levels as high as possible. It doesn't make sense to waste time trying to find a problem that the compiler could find for you! We need to concentrate on the harder problems at hand.

When trying to solve any problem, you need to gather all the relevant data. Unfortunately, bug reporting isn't an exact science. It's easy to be misled by coincidences, and you can't afford to waste time debugging coincidences. You first need to be accurate in your observations.

Accuracy in bug reports is further diminished when they come through a third party—you may actually need to watch the user who reported the bug in action to get a sufficient level of detail.

Andy once worked on a large graphics application. Nearing release, the testers reported that the application crashed every time they painted a stroke with a particular brush. The programmer responsible argued that there was nothing wrong with it; he had tried painting with it, and it worked just fine. This dialog went back and forth for several days, with tempers rapidly rising.

Finally, we got them together in the same room. The tester selected the brush tool and painted a stroke from the upper right corner to the lower left corner. The application exploded. "Oh," said the programmer, in a small voice, who then sheepishly admitted that he had made test strokes only from the lower left to the upper right, which did not expose the bug.

There are two points to this story:

You may need to interview the user who reported the bug in order to gather more data than you were initially given.

Artificial tests (such as the programmer's single brush stroke from bottom to top) don't exercise enough of an application. You must brutally test both boundary conditions and realistic end-user usage patterns. You need to do this systematically.

Debugging Strategies

Once you think you know what is going on, it's time to find out what the program thinks is going on.

Visualize Your Data

Often, the easiest way to discern what a program is doing—or what it is going to do—is to get a good look at the data it is operating on. The simplest example of this is a straightforward "variable name = data value" approach, which may be implemented as printed text, or as fields in a GUI dialog box or list.

But you can gain a much deeper insight into your data by using a debugger that allows you to visualize your data and all of the interrelationships that exist. There are debuggers that can represent your data as a 3D fly-over through a virtual reality landscape, or as a 3D waveform plot, or just as simple structural diagrams. As you single-step through your program, pictures like these can be worth much more than a thousand words, as the bug you've been hunting suddenly jumps out at you.

Even if your debugger has limited support for visualizing data, you can still do it yourself—either by hand, with paper and pencil, or with external plotting programs.

Tracing

Debuggers generally focus on the state of the program now. Sometimes you need more—you need to watch the state of a program or a data structure over time. Seeing a stack trace can only tell you how you got here directly. It can't tell you what you were doing prior to this call chain, especially in event-based systems.

Tracing statements are those little diagnostic messages you print to the screen or to a file that say things such as "got here" and "value of x = 2." It's a primitive technique compared with IDE-style debuggers, but it is peculiarly effective at diagnosing several classes of errors that debuggers can't. Tracing is invaluable in any system where time itself is a factor: concurrent processes, real-time systems, and event-based applications.

You can use tracing statements to "drill down" into the code. That is, you can add tracing statements as you descend the call tree.

Trace messages should be in a regular, consistent format; you may want to parse them automatically. For instance, if you needed to track down a resource leak (such as unbalanced file opens/closes), you could trace each open and each close in a log file.

Rubber Ducking

A very simple but particularly useful technique for finding the cause of a problem is simply to explain it to someone else. The other person should look over your shoulder at the screen, and nod his or her head constantly (like a rubber duck bobbing up and down in a bathtub). They do not need to say a word; the simple act of explaining, step by step, what the code is supposed to do often causes the problem to leap off the screen and announce itself.

It sounds simple, but in explaining the problem to another person you must explicitly state things that you may take for granted when going through the code yourself. By having to verbalize some of these assumptions, you may suddenly gain new insight into the problem.

Process of Elimination

In most projects, the code you are debugging may be a mixture of application code written by you and others on your project team, third-party products (database, connectivity, graphical libraries, specialized communications or algorithms, and so on) and the platform environment (operating system, system libraries, and compilers).

It is possible that a bug exists in the OS, the compiler, or a third-party product—but this should not be your first thought. It is much more likely that the bug exists in the application code under development. It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken. Even if the problem does lie with a third party, you'll still have to eliminate your code before submitting the bug report.

Remember, if you see hoof prints, think horses—not zebras. The OS is probably not broken. And the database is probably just fine.

If you "changed only one thing" and the system stopped working, that one thing was likely to be responsible, directly or indirectly, no matter how farfetched it seems. Sometimes the thing that changed is outside of your control: new versions of the OS, compiler, database, or other third-party software can wreak havoc with previously correct code. New bugs might show up. Bugs for which you had a work-around get fixed, breaking the work-around. APIs change, functionality changes; in short, it's a whole new ball game, and you must retest the system under these new conditions. So keep a close eye on the schedule when considering an upgrade; you may want to wait until after the next release.

If, however, you have no obvious place to start looking, you can always rely on a good old-fashioned binary search. See if the symptoms are present at either of two far away spots in the code. Then look in the middle. If the problem is present, then the bug lies between the start and the middle point; otherwise, it is between the middle point and the end. You can continue in this fashion until you narrow down the spot sufficiently to identify the problem.

LANGUAGE & COMPREHENSION WORKSHOP

Part 1: Yes/No/Not Given Questions
Indicate whether each statement is true, false, or not given.

1. The first recorded “computer bug” was identified due to a calculation error in COBOL.
2. The term "bug" in the context of computing refers exclusively to software defects.
3. The debugging techniques discussed in the article include both modern and traditional methods.
4. The "Rubber Ducking" technique requires the presence of another person who is also knowledgeable in programming.
5. In the text, the author claims that most debugging errors are due to flaws in third-party libraries.

Part 2: Cloze Questions (Multiple Choice with Advanced Options)
Select the most accurate answer to complete each sentence.

1. The author emphasizes that debugging should be approached primarily as ___.
   - A) a collaborative task
   - B) a technical exercise focused on isolating errors
   - C) a psychological exercise involving self-awareness
   - D) an activity of blame allocation
   - E) an exercise in complex logic formulation
2. According to the text, the main purpose of “rubber ducking” is to ___.
   - A) receive feedback from a knowledgeable peer
   - B) challenge one’s understanding by verbalizing code logic
   - C) make the debugging process more collaborative
   - D) detect syntax errors in the code
   - E) gain emotional support during a frustrating process
3. Debugging begins with the assumption that ___.
   - A) third-party software is the most probable cause of issues
   - B) human error is a significant factor to rule out
   - C) the application’s environment is likely flawed
   - D) all errors can be traced back to user interactions
   - E) the code under development is the probable source of bugs

4. When investigating a new bug, the author advises programmers to ___.
    - A) rely on external libraries for advanced debugging tools
    - B) perform minimal tests to conserve time and resources
    - C) gather observational data meticulously before hypothesizing
    - D) update all software components as a first response
    - E) remove any custom code from third-party libraries
5. In the context of boundary testing, the author's anecdote about the graphics application emphasizes the need to ___.
    - A) test features only in controlled conditions
    - B) rely heavily on the developer's instincts
    - C) observe user behaviors to replicate real-world scenarios
    - D) limit testing to isolated cases to avoid interference
    - E) perform all tests in an IDE environment

Part 3: Gap-Fill Questions
Choose the most suitable word from the options provided to complete each sentence.

1. "It's crucial not to let ___ cloud your judgment when debugging; stay composed and logical."
    - (a) assumptions
    - (b) panic
    - (c) optimism
2. "By ___ the reported symptoms, developers can often avoid wasting time on unrelated coincidences."
    - (a) verifying
    - (b) ignoring
    - (c) intensifying
3. "In the 'rubber ducking' process, verbalizing each assumption can lead to unexpected ___ about the bug."
    - (a) insights
    - (b) obstructions
    - (c) oversights
4. "The anecdote with the graphic application underlines the importance of ___ test scenarios."
    - (a) automated
    - (b) realistic

    - (c) simplified

5. "Attempting to ___ the root cause, rather than just surface symptoms, leads to more effective debugging."
    - (a) diagnose
    - (b) ignore
    - (c) create

Part 4: Short Answer Questions

1. Explain the psychological approach to debugging as discussed in the article. How does this mindset benefit a developer?

    ____
    ____
    ____

2. Why does the author mention "Rubber Ducking" as an effective debugging strategy? Explain how it contributes to identifying bugs.

    ____
    ____
    ____

3. Describe the significance of boundary testing as illustrated by the story of the graphic application. Why does the author consider this type of testing important?

    ____
    ____
    ____

4. What role does data visualization play in debugging, according to the author? Provide an example of how this technique could assist in troubleshooting.

    ____
    ____
    ____

Part 5: Critical Analysis
Reflect on the statement: "If you see hoof prints, think horses—not zebras." What is the author's underlying message regarding debugging, and how can this philosophy aid in resolving complex issues?

____
____
____