



Debugging Strategies

(c) dan frost 2010

Information gathering

What was the system designed and built to do? Understanding how the system got into this state might explain its behaviour. Describe what the functions *should* be doing – is it different from what they *are* doing?

Endogenous –from within the system. Error messages, logs. **Exogenous** from outside. Monitor outputs of the system. Rig the inputs.

History Does the system's history explain the problem? Is the problem known? Is there any historical information or do we have to run new tests to gather information?

Logs Where are they? What's in them?

Error messages/codes. Often the first piece of information.

Code coverage Which pieces of code are actually being executed?

Small, specific, short tests. Design very simple tests. Gather specific pieces of information (the email was/wasn't sent). Keep length of time to run a single test very short – otherwise, debugging can take days.

Breakpoint Divides the problem space and reduces the time taken for each test.

Make it fail or - confirm the bounds of normal operation. **Confirm that you have control over the system** and that part of it behaves as expected.

One thing at a time otherwise information becomes blurred. If multiple changes: differentiate the output of these changes.

Keep an audit trail – Record what's changed.

Logic Tools

Differential diagnosis List likely causes and their symptoms. Remove any causes whose symptoms are not, or cannot be present. Gather information until the list of confirmed symptoms points, unequivocally to one cause.

Occam's razor – if a theory assumes more, it is less likely to be true and is harder to prove. Therefore assume the simplest theory is true.

Probably and possible causes

Probable: X% of cause producing the combination of symptoms

Possible: X can cause the symptoms

Divide the problem space. Don't use brute force, but reduce the possible causes using tests that divide the system in half.

Problem solving

Assumption reversal. (Check the plug.) Write down all assumptions and ask if the reverse of each could be true. What are the implication if the reverse is true?

Brute force

Methodically go through all possibilities, recording, but ignoring any information until the end. This provides complete information on the problem.

Trial and error

Helps reduce problem space. Each trial results in more information. Adjust the trial with each error.

Brain-storming

For getting ideas. Aim is to get as many ideas as possible. Rules are: Quantity of ideas is important; no criticism of ideas; unusual ideas welcome; combine and improve ideas.

Incubation Take all the available information and think about it. This isn't equal to putting it off!

Bug location strategies

Mental simulation (upwards reasoning): Mentally walks through what the should be executing. Compare to reality.

Causal reasoning (downwards reasoning): What cause does the output imply? Requires knowledge of the program. (e.g. Behaviour-to-bug)

Bug location errors

Not understanding / misunderstanding code

Inappropriate code to solve the problem

Misunderstanding the environment (e.g. the language, the program)

These causes *unnecessary, potentially dangerous changes.*

Analogy Find similar problem in a different situation or field. Apply the solution to your domain and take lessons from it.

Morphological box

For dealing with conflicting priorities.

Socratic method Dialectic method of enquiry – 1st speaker leads discussion, 2nd agrees to certain assumptions put forward. Flesh out the issue so that judgements can be made in relation to them.

Research

Has someone else solved it already? Research requires listing the symptoms using *standard* and *common* terms. I.e. describe the problem as someone else has done.

Consult / ask other people. Like research, you should describe the problem in standard/common terms

Prove it can't be solved using clear logic.