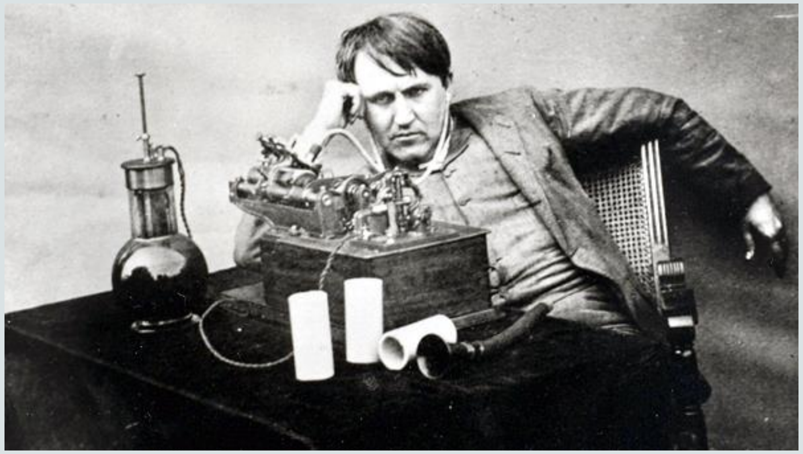
61757 - INTRODUCTION TO SOFTWARE TESTING

# Debugging (Part II)



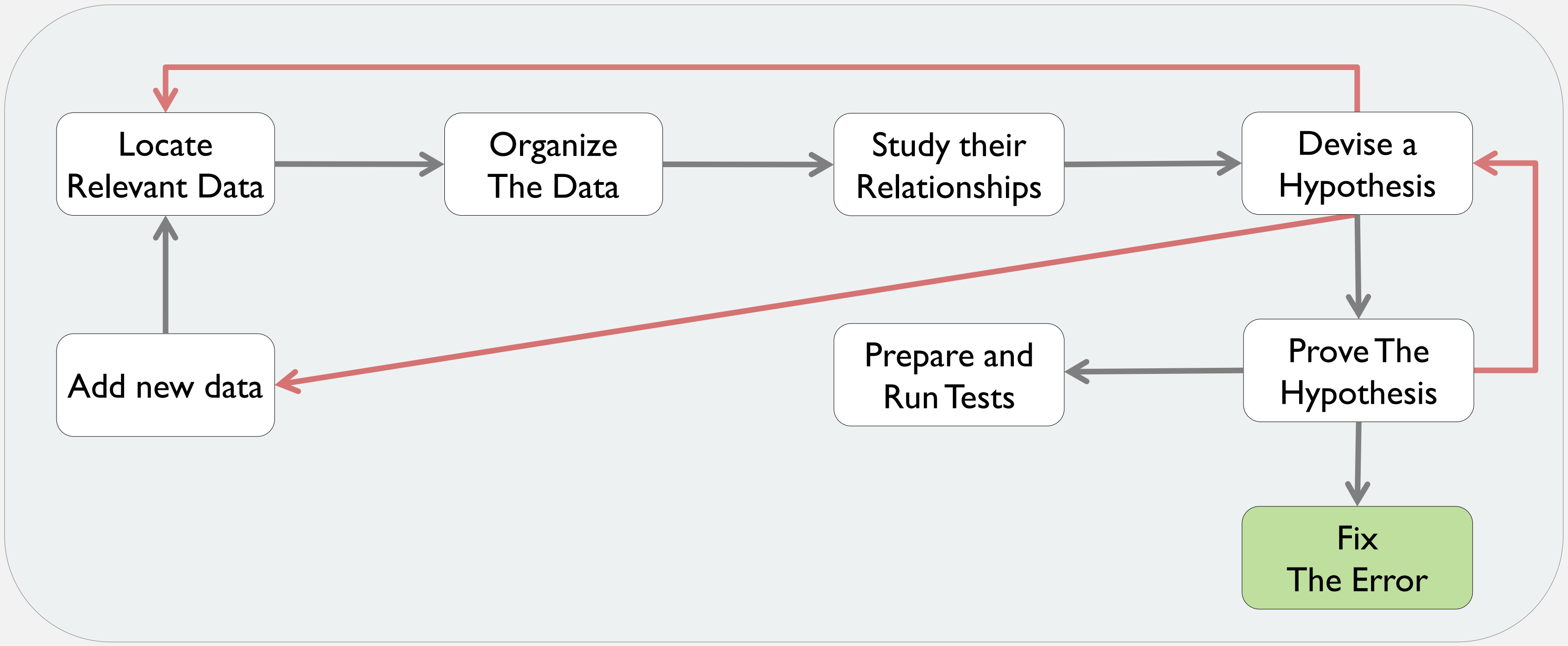
*"It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and [it is] then that 'Bugs'—as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite. . ."*

*Thomas Edison*

# DEBUGGING BY INDUCTION

Induction or inductive logic is a kind of reasoning that draws generalized conclusions from a finite collection of specific observations. You move from the particulars of a situation to the whole picture.

## The scheme of inductive debugging process



Dr. Sergey Mazin "Introducon to Software Testing" 2

## DEBUGGING BY INDUCTION

**1. Locate the relevant data**

Don't take into account of all available data or symptoms about the problem. The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly - the symptoms that led you to believe there was an error. Additional valuable clues are provided different - test cases that do not cause the symptoms to appear.

**2. Organize the data**

Remember that induction implies that you're processing from the particulars to the general, so this step is to structure the relevant data to let you ***observe the patterns***. Particular importance is the search for contradictions, events such as that the error occurs. It is possible to use a form (**presented on the next slide**) to structure the available data.

# DEBUGGING BY INDUCTION

The "**what**" boxes list the general symptoms

The "**where**" boxes describe where the symptoms were observed

The "**when**" boxes list anything that Is known about the times that the symptoms occur

The "**to what extend**" boxes describe which additional details it's worth to add/not to add. Which test cases to create.

| Question | Is | Is Not |
|---|---|---|
| What? | | |
| Where? | | |
| When? | | |
| To what Extend? | | |

## DEBUGGING BY INDUCTION

3. **Devise a hypothesis**

   Next, study the relationships among the clues and devise, using the **patterns** that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If you can't devise a theory, more data are needed, perhaps from new test cases. If multiple theories seem possible, select the more probable one first.

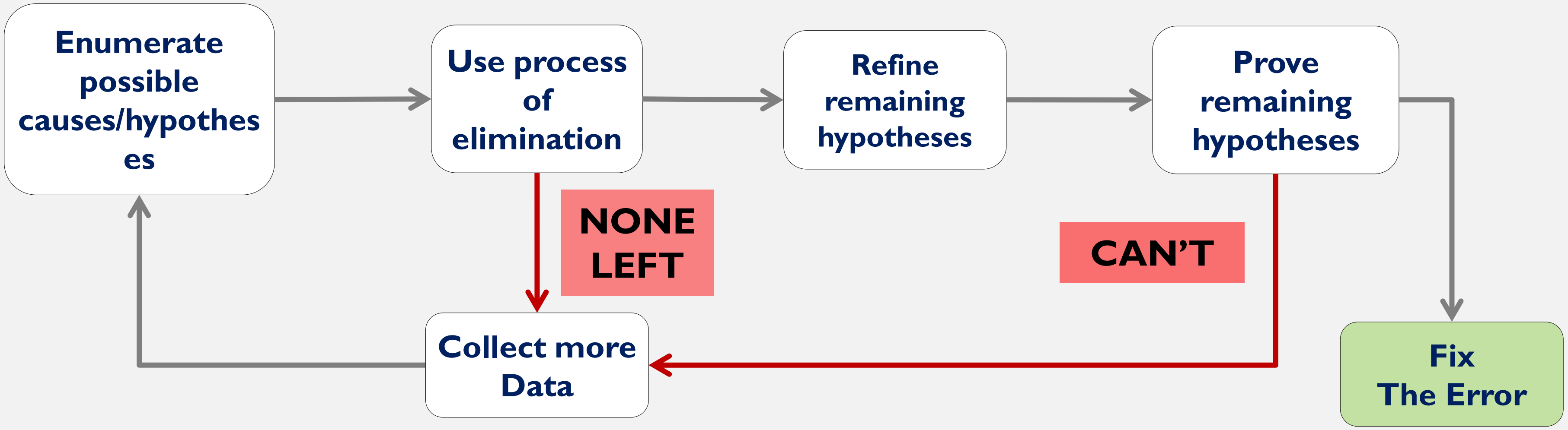
4. **Prove the hypothesis**

   A major mistake at this point, given the pressures under which debugging usually is performed, is skipping this step and jumping to conclusions to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before you proceed. If you skip this step, you'll probably succeed in correcting only the problem symptom, not the problem itself. Prove the hypothesis by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. ***If it does not, either the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.***

## DEBUGGING BY DEDUCTION

The process of *deduction* proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error)

As opposed to the process of induction in a murder case, for example, where you induce a suspect from the clues, you start with a set of suspects and, by the process of elimination (the gardener has a valid alibi) and refinement (it must be someone with red hair), decide that the butler must have done it. The steps are as follows:


Enumerate possible causes/hypothes es
Use process of elimination
Refine remaining hypotheses
Prove remaining hypotheses
NONE LEFT
CAN'T
Collect more Data
Fix The Error

## DEBUGGING BY DEDUCTION

1. **Enumerate the possible causes or hypotheses**
   The first step is to develop a list of all conceivable causes of the error. They don't have to be complete explanations; they are merely theories to help you structure and analyze the available data.

2. **Use the data to eliminate possible causes**
   Carefully examine all of the data, particularly by looking for contradictions, and try to eliminate all but one of the possible causes. If all are eliminated, you need more data through additional test cases to devise new theories. If more than one possible cause remains, select the most probable cause first.

3. **Refine the remaining hypothesis**.
   The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory. For example, you might start with the idea that "there is an error in handling the last transaction in the file" and refine it to "the last transaction in the buffer is overlaid with the end-of-file indicator."

4. **Prove the remaining hypothesis**
   This vital step is identical to step 4 in the ***induction*** method.

## DEBUGGING BY TESTING

- After a symptom of a suspected error is discovered, it's written variants of the original test case to attempt to pinpoint the error. The difference between test cases is:
  - test cases for testing tend to be "fat" because you are trying to cover many conditions in a small number of test cases;
  - test cases for debugging are "slim" since you want to cover only a single condition.

- Usually it is used in conjunction with the ***induction*** method to obtain information needed to generate a hypothesis and/or to prove a hypothesis. It also is used with the ***deduction*** method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.

## “POSTMORTEM” ANALYSIS

***Where was the error made?***

This question is the most difficult one to answer, because it requires a backward search through the documentation and history of the project, but it also is the most valuable question. It requires that you pinpoint the original source and time of the error.

***For example:*** the ***original*** source of the error might be an ambiguous statement in a specification, a correction to a prior error, or a misunderstanding of an enduser requirement.

***Who made the error?***

If it will be discovered that 60 percent of the design errors were created by one of the 10 analysts, or that programmer X makes three times as many mistakes as the other programmers, it will be useful not for the purposes of punishment but for the purposes of education only.

## “POSTMORTEM” ANALYSIS

***What was done incorrectly?***

It is not sufficient to determine when and by whom each error was made; for instance, the missing link determines exactly why the error occurred. Was it caused by someone’s inability to write clearly? Someone’s lack of education in the programming language? A typing mistake? An invalid assumption? A failure to consider valid input?

***How could the error have been prevented?***

What can be done differently in the next project to prevent this type of error? The answer to this question constitutes much of the valuable feedback or learning for which we are searching.

***Why wasn’t the error detected earlier?***

If the error is detected during a test phase, you should study why the error was not detected during earlier testing phases, code inspections, and design reviews.

## "POSTMORTEM" ANALYSIS

***How could the error have been detected earlier?***

The answer to this is another piece of valuable feedback. How can the review and testing processes be improved to find this type of error earlier in future projects? Providing that we are not analyzing an error found by an end user (that is, the error was found by a test case), we should realize that something valuable has happened:

- ✓ We have written a successful test case.
- ✓ Why was this test case successful?
- ✓ Can we learn something from it that will result in additional successful test cases, either for this program or for future programs?

Lucky Debugging!!

## *Backup*