# Some Ideas on Debugging

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

May 10, 2022

# Outline

# Outline

# Outline

- Software defect

# Outline

# Software Defect

Defects (or faults) : deviations from requirement or design specifications or expectations which might lead to failures in operation.

Colloquially, called bugs.

# The Origin of "Bug"

"*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 (circa 1840's)

# The Origin of Computer "Bug" and "Debug"

One day in the 1940s, Harvard's famed Mark I—the precursor of today's computers—failed. When the Harvard scientists looked inside, they found a moth that had lodged in the Mark I's circuits. They removed the moth with a pair of tweezers, and from then on, whenever there was a problem with the Mark I, the scientists said they were looking for bugs. The term has stuck through the years. [*Dun's Business Month*, Feb. 1983: 125]

In some versions, the moth is said to have inspired the scientists to speak from then on of *debugging* the computer, with *bug* originating as a later back-formation from *debug*.

Figure: Source: See Shapiro[1]

[1] Fred R Shapiro. "Etymology of the computer bug: History and folklore". In: *American Speech* 62.4 (1987), pp. 376–378.

# The Origin of Computer "Bug" and "Debug"

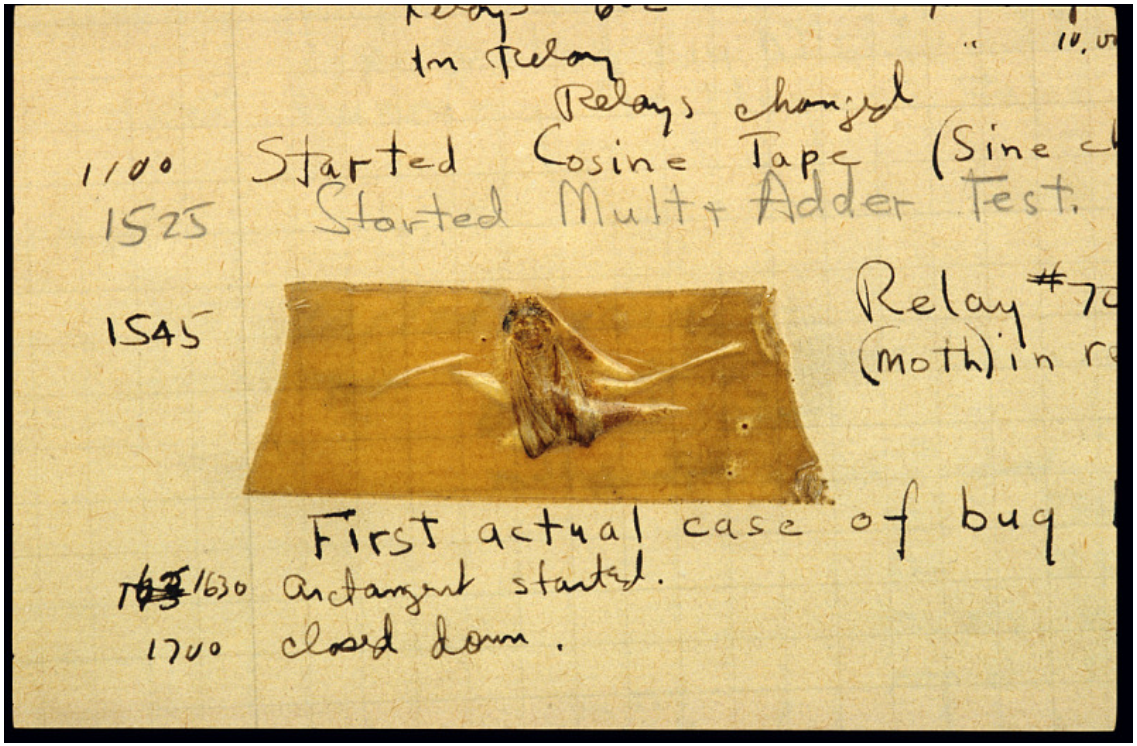



Figure: Archived at the Smithsonian Institution's National Museum of American History

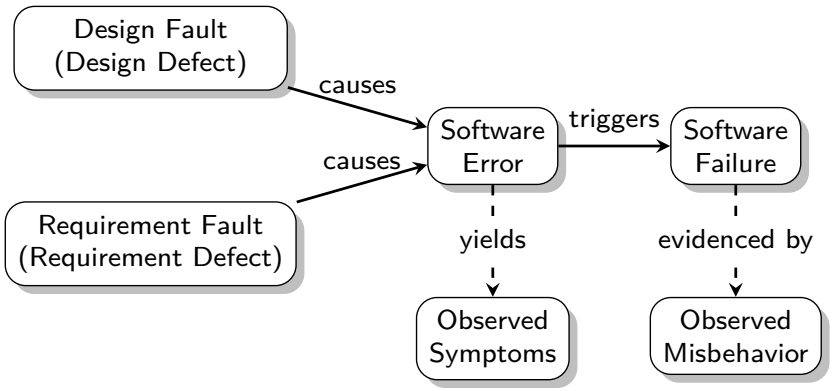
# Outline

# Need to Develop Debugging Skills

Debugging is a skill that we can gradually learn and develop

No matter how software development progresses, there will likely always be bugs to squash

# Defect, Errors, and Failure




(a) Relations of software defects (faults), errors, and failures.

```c
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };

int f(void) {
    char *text_buffer = (char *)malloc(
        BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }

    return 0;
}
```

(b) A code snippet showing a memory allocation defect

Figure: Relationship of defect, error, and failure and an example of defective code

"*an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of* ***error****. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders.*" – Ada Lovelace's notes on Babbage's Analytical Engine

# Defect, Errors, and Failure

A failure comes to be in three stages:

1. The programmer creates a defect (or fault)
2. The defect causes an infection (or error)
3. The infection (or error) causes a failure – an externally visible error.
   - Not every defect results in an infection, and not every infection results in a failure.

# Debugging is a Search Problem

Given a correct state and a failure state of the program, search across time and space to find the defect.

- State can be very large, consisting of millions of variables
- Time can be lengthy and consist of lots of individual (sometime concurrent) actions

Search is driven by two processes

- Separate relevant from irrelevant state
- Separate sane from infected

# Outline

# Debugging Process

To debug a program, generally, proceed in 7 steps:

1. Track the problem in the bug database
2. Reproduce the failure
3. Automate and simplify the test case
4. Find possible infection origins
5. Focus on the most likely origins
6. Isolate the infection chain
7. Correct the defect

# Tracking the Problem

Report and track the problem, generally, do the following,

- Look for duplicates
- Provide sufficient context
- Prioritize
- Relate the problem to
  - a test (or tests)
  - a feature
  - requirement specification
  - released version of the software

Bug (issues) trackers are now the norm

- PivotalTracker, BugZilla, GitHub's issue trackers, ...

# Reproducing the Failure

- Reproducing is one of the toughest problems in debugging.
- One must
  - recreate the environment in which the problem occurred
  - recreate the problem history – the steps that lead to the problem
- Iterative process of reproducing bugs
  1. Start with your environment
  2. While the problem is not reproduced, adapt more and more circumstances from the user's environment
  3. Iteration ends when problem is reproduced (or when environments are "identical")

# Reproducing the Environment

- Many configurations ...
- Testing on these configurations
- All needed to find & reproduce problems

# Reproducing Execution

After reproducing the environment, we need to recreate the problem history.

we must reproduce the execution

- Basic idea: Any execution is determined by the input (in a general sense)
- Reproducing input $\rightarrow$ reproducing execution ?
  - Easy to transfer and replicate
  - Caveat #1: Get all the data you need
  - Caveat #2: Get only the data you need
  - Caveat #3: Privacy issues

# Reproducing Communication

General idea: Record and replay, e.g., user interaction

- It can be slow
- Difficult for systems that still need to process user interactions
- Also, tracing can create lots of data, e.g.,
  - Web server with 10 requests/sec
  - A trace of 10 k/request means 8GB/day
  - All of this must be replayed to reproduce the failure (alternative: checkpoints)

# Reproducing Randomness and Concurrency

Program behaves different in every run. How to test?

Example.

- Based on random number generator
- Pseudo-random: save seed (and make it configurable)

Example.

- Trace-driven. record + reply sequence

Example.

- Thread changes are induced by a scheduler
- It suffices to record the schedule (i.e. the moments in time at which thread switches occur) and to replay it

# Be aware of Heisenbug

```
int f()
{
int i;
return i;
}
```

# Automating Tests

Automate and simplify the test case

- Can be system level (i.e. user behavior based)
- Can be lower layers of functionality

Good idea to reproduce the bug via a test case

- Easy way to ensure it is fixed
- Good to keep regression-testing it

# Finding and Fixing Defects

The rest of the process consists of

1. Find Possible Infection Origins
2. Focus Likely Origins
3. Isolate Infection Chain
4. Correct Defects

# Scientific Debugging

Some people are good at guessing causes! Unfortunately, intuition is hard to grasp:

- Requires a priori knowledge
- Does not work in a systematic and reproducible fashion
- In short: Intuition cannot be taught

The Scientific Method – can be taught

- The scientific method is a general pattern of how to find a theory that explains (and predicts) some aspect of the universe
- Called "scientific method" because it's supposed to summarize the way that (experimental) scientists work

# Scientific Method

1. *Observe* some aspect of the universe.
2. Invent a *hypothesis* that is consistent with the observation.
3. Use the hypothesis to make *predictions*.
4. *Tests* the predictions by experiments or observations and modify the hypothesis.
5. Repeat 3 and 4 to refine the hypothesis

# Scientific Debugging

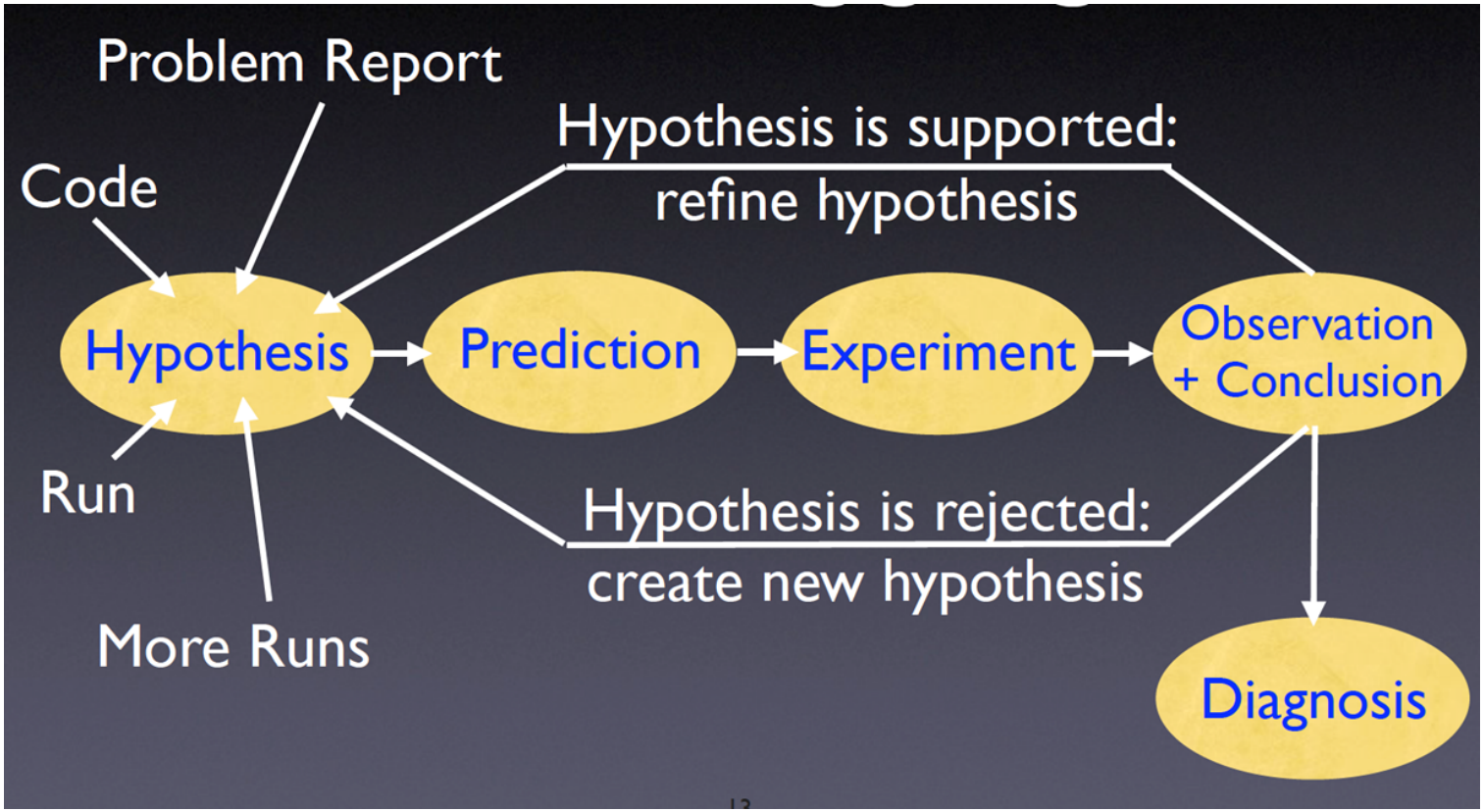



Figure: Source: Zeller[2]

[2] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

# Scientific Debugging: Example

```
$ sample 9 8 7
output: 7 8 9

$ $ sample 11 14
output: 0 11
```

# Scientific Debugging: Iteration 1

```
 1 int main(int argc, char *argv[])
 2 {
 3     int *a;
 4     int i;
 5
 6     a = (int *)malloc((argc - 1) * sizeof(int));
 7     for (i = 0; i < argc - 1; i++)
 8         a[i] = atoi(argv[i + 1]);
 9
10     shell_sort(a, argc);
11
12     printf("Output: ");
13     for (i = 0; i < argc - 1; i++)
14         printf("%d ", a[i]);
15     printf("\n");
16
17     free(a);
18
19     return 0;
20 }
```

- Observe the input and "Output:" ...
- Hypothesis: "the infection exists after the call to shell sort ..."
- Experiment and confirmed: `a[] = [0, 11, 14]`

# Scientific Debugging: Iteration 2

```
 1 int main(int argc, char *argv[])
 2 {
 3     int *a;
 4     int i;
 5
 6     a = (int *)malloc((argc - 1) * sizeof(int));
 7     for (i = 0; i < argc - 1; i++)
 8         a[i] = atoi(argv[i + 1]);
 9
10     shell_sort(a, argc);
11
12     printf("Output: ");
13     for (i = 0; i < argc - 1; i++)
14         printf("%d ", a[i]);
15     printf("\n");
16
17     free(a);
18
19     return 0;
20 }
```

- Observe the input and "Output:", and the result of the previous hypothesis ...
- Hypothesis: "the infection does not occur until the call the shell sort "
- Experiment and confirmed: `a[] = [11, 14]; size = 2`

# Scientific Debugging: Iteration 3

```
1  int main(int argc, char *argv[])
2  {
3      int *a;
4      int i;
5
6      a = (int *)malloc((argc - 1) * sizeof(int));
7      for (i = 0; i < argc - 1; i++)
8          a[i] = atoi(argv[i + 1]);
9
10     shell_sort(a, argc);
11
12     printf("Output: ");
13     for (i = 0; i < argc - 1; i++)
14         printf("%d ", a[i]);
15     printf("\n");
16
17     free(a);
18
19     return 0;
20 }
```

- Observe the input and "Output:", and the result of the previous hypotheses ...
- Hypothesis: "the call to shell sort is wrong"
- Experiment and confirmed: the error goes away when making it `argc-1`

# Scientific Bugging: More Examples

- Example 1
- Example 2

# Scientific Debugging: Making the Process Explicit

"Everything gets written down, formally, so that you know at all times where you are, where you've been, where you are going, and where you want to get. In scientific work and electronics technology this is necessary because otherwise problems get so complex you get lost in them and confused and forget about what you know and what you don't know and have to give up" – "Zen and the Art of Motorcycle Maintenance" by Robert M. Pirsig

People can remember about seven chunks in short-term memory (STM) tasks (See Miller[3])

[3] George A Miller. "The magical number seven, plus or minus two: Some limits on our capacity for processing information.". In: *Psychological review* 63.2 (1956), p. 81.

# Scientific vs. Quick and Dirty

- Not every problem needs the strength of the scientific method or explicit debugging
- a quick-and-dirty process sometimes suffices.
- Suggestion: Go quick and dirty for several minutes, and then apply the scientific method.

# Outline

# Summary

To debug a program, generally, proceed in 7 steps:

1. Track the problem in the bug database
2. Reproduce the failure
3. Automate and simplify the test case
4. Find possible infection origins
5. Focus on the most likely origins
6. Isolate the infection chain
7. Correct the defect

Use scientific debugging method in conjunction with the quick and dirty method.

# Outline

Zeller, Andreas. Why programs fail: a guide to systematic debugging. Elsevier, 2009.
"Engineering Software as a Service" by Armando Fox and David Patterson (2nd Edition)
"Introduction to Software Design with Java" by Martin P. Robillard
"Essentials of Software Engineering" by Frank Tsui, Orlando Karam, and Barbara Bernal(4th Edition)