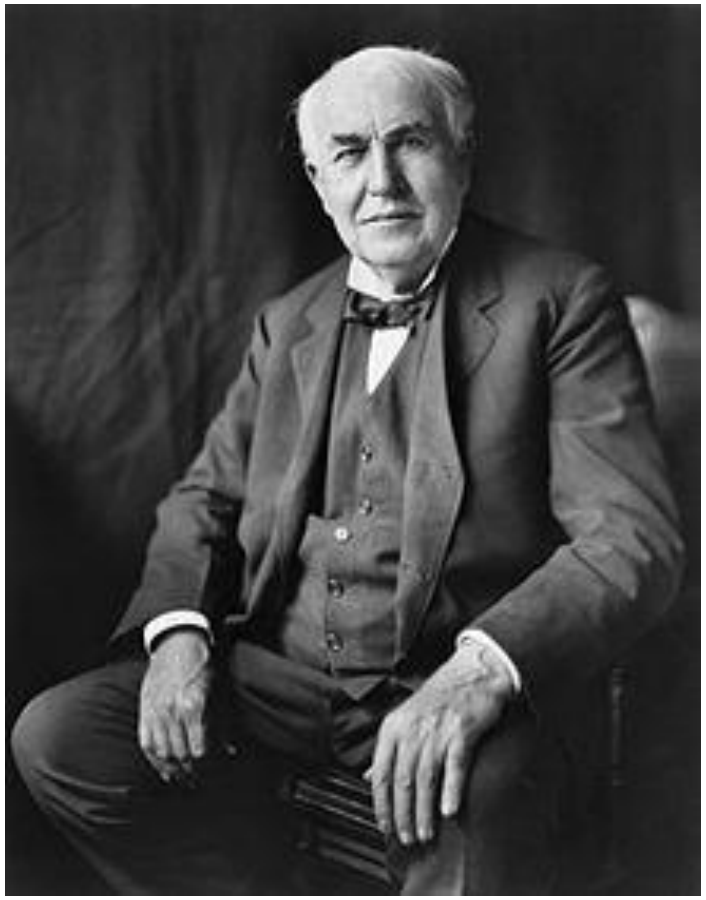
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Design and Debugging

18-213/613: Introduction to Computer Systems
15th Lecture, October 22, 2024

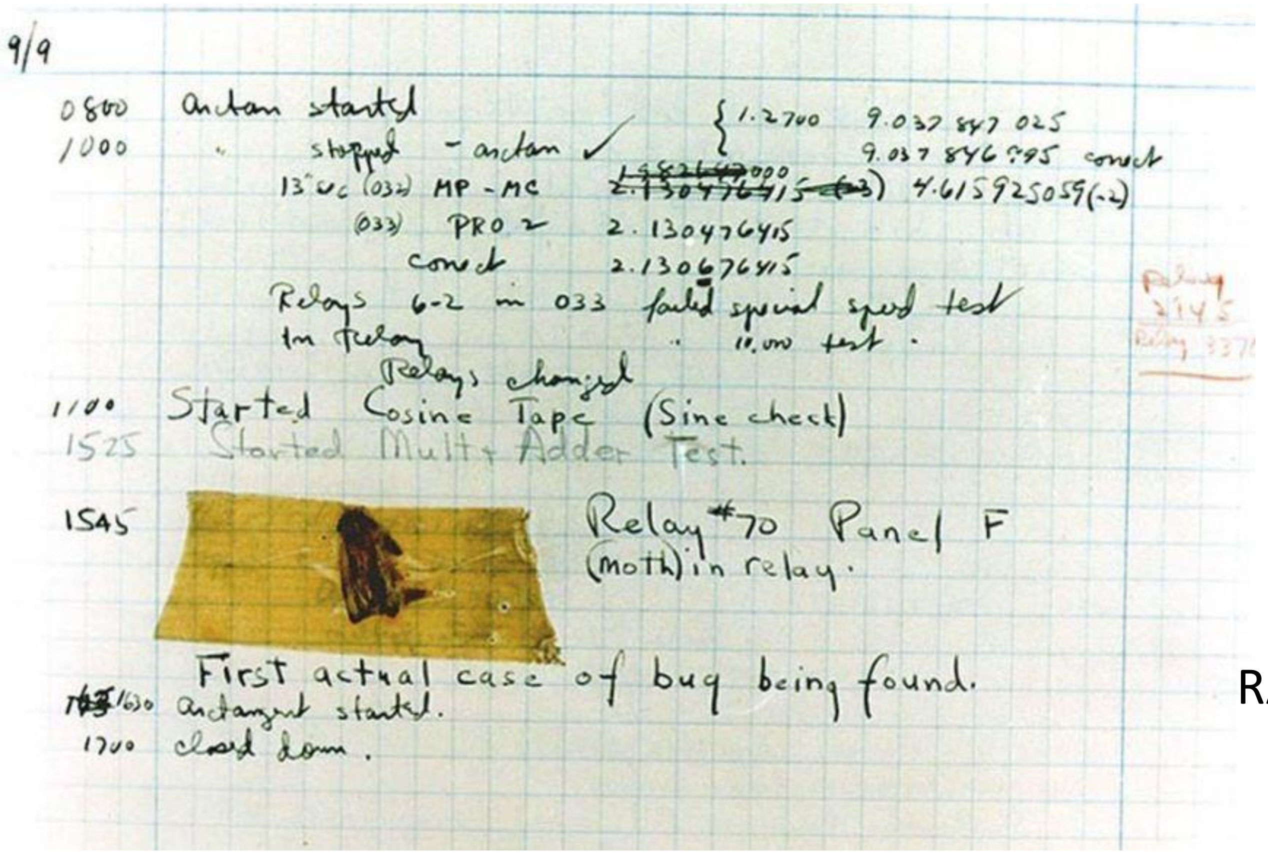
Bugs: You are in good company!



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 before commercial success or failure is certainly reached.

-- Thomas Edison

Bugs: You are in good company!



9/9

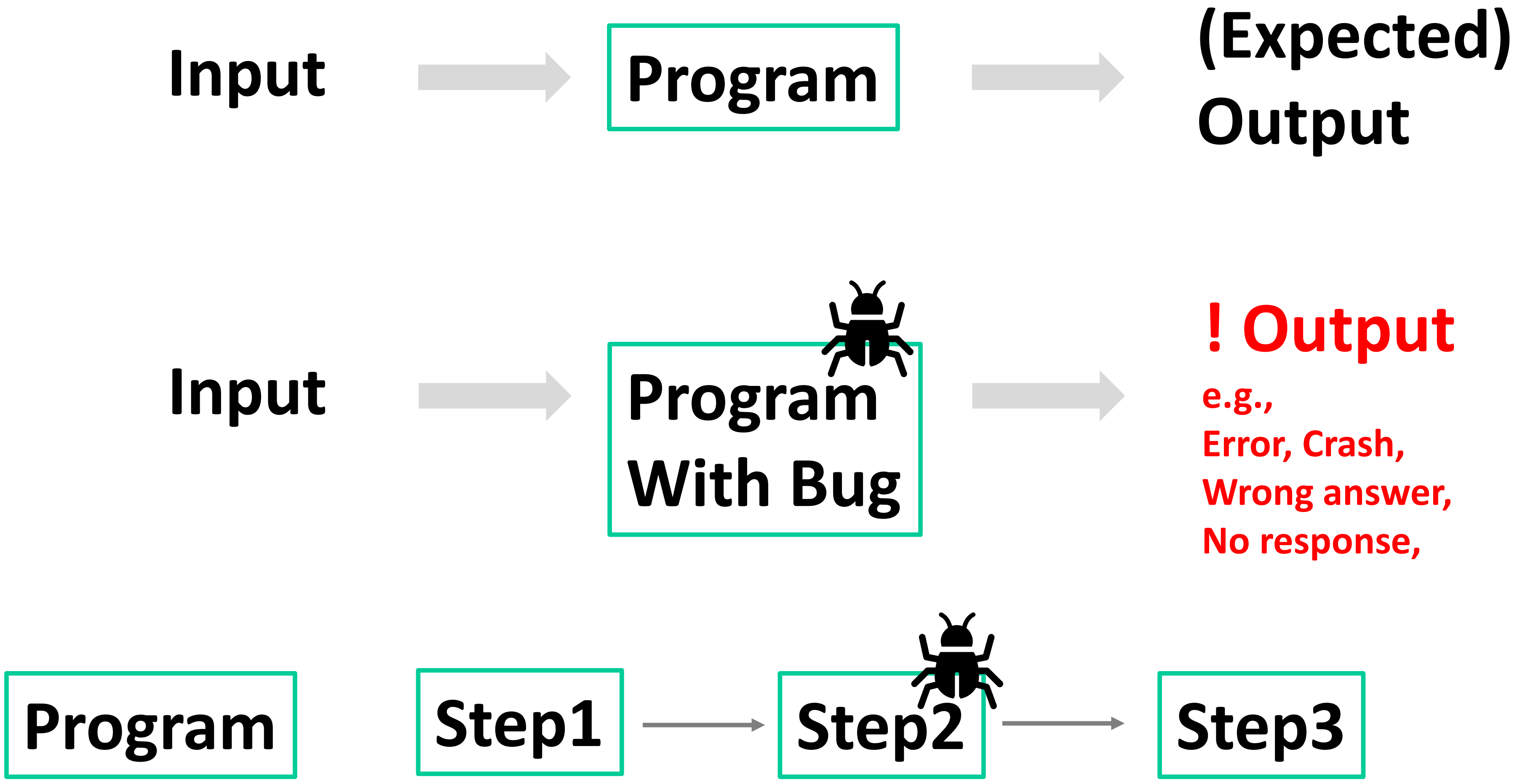
0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(23)~~) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Relay 2145
Relay 3370

Source: https://education.nationalgeographic.org/resource/worlds-first-computer-bug/

RADM Grace Hopper, PhD

High Level View



What is debugging?

Given as Input

Program With Bug

+

! Output

e.g.,
Error, Crash,
Wrong answer,
No response,

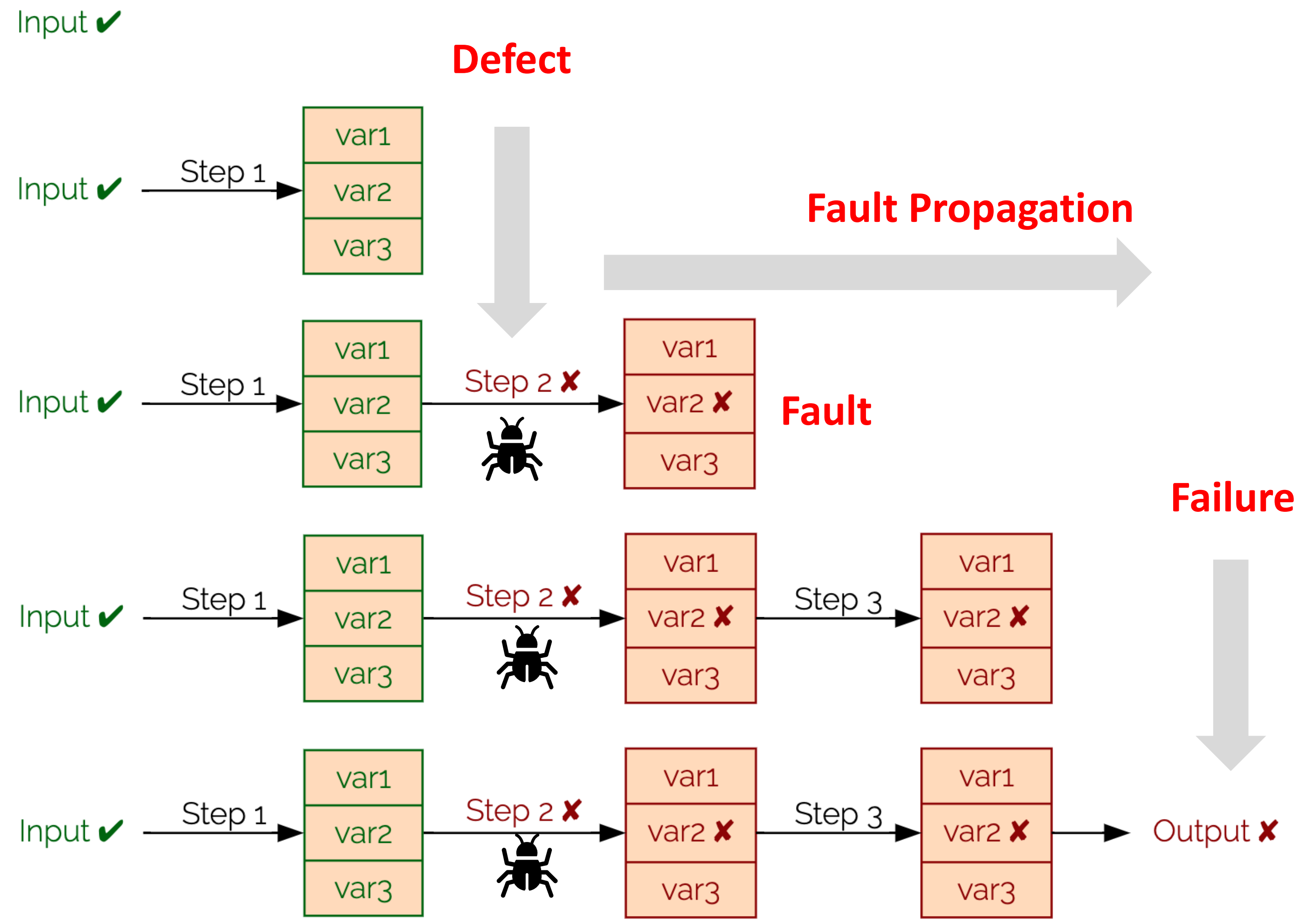
Step2

Produce the
Location, source of
Defect and
A potential fix!

Debugging

- Art and science of fixing bugs in ~~software~~ engineering systems
- Where is the bug?
- When does it occur? What triggers it?
- How can we repair it?

Defects, Propagation, & Failures



Caveat: Curse of Debugging

Testing can only show the presence of errors – not their absence. (Dijkstra 1972)

- **Not every defect causes a visible/observable failure!**
- **Fortunately, most failures can be traced back to the defect that causes it!**

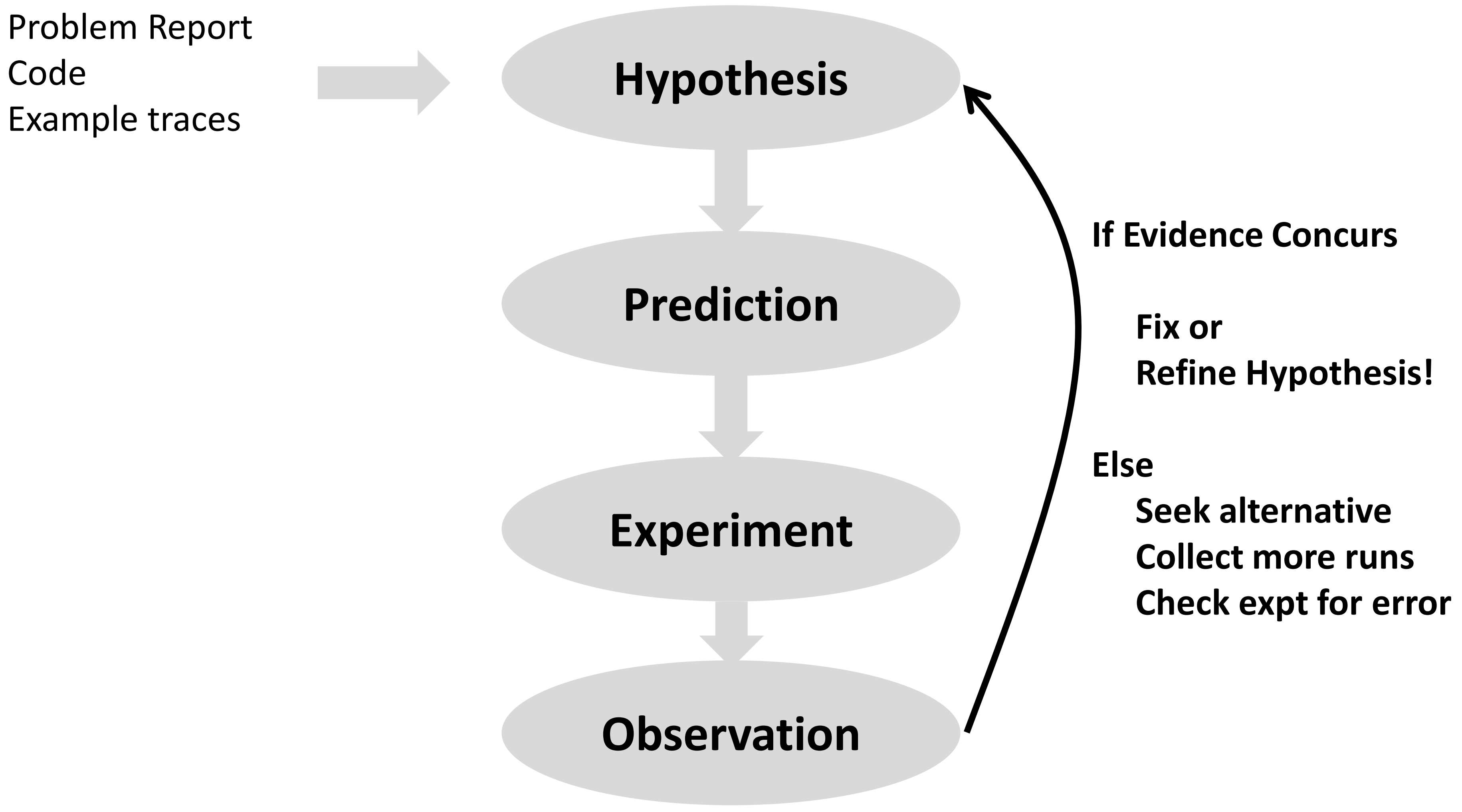
Debugging is hard, make no mistake!

- Program “states” are large
- Executions can have multiple steps
- Defects/errors could be subtle
- Defects/errors only triggered for some edge inputs!
- Other factors: concurrency, randomization, distributed state, …

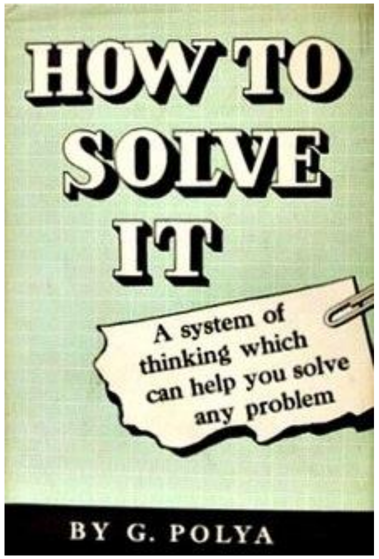
How do you make progress if its hard?

- **Scientific hypothesis testing**
  - Meta: Problem solving strategies to generate hypothesis/leads
  - Tools to prune search space and collect data
- **“Rubberducking”**
- **..**

Scientific Process for Hypothesis Testing



Meta : Art of Problem Solving



Understand

Can you restate the problem in your own words?
•Can you think of a picture or a diagram that might help you understand the problem?
•Is there enough information to enable you to find a solution?

↓

Make a Plan

Guess and check
Consider special cases
Solve a simpler problem
Divide and conquer
…

↓

Execute the Plan

Grit/Persevere ☺

↓

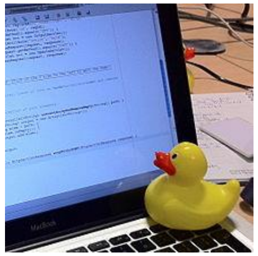
Look back on your work

Did you solve all edge cases/
Was there a better fix?
Was there a quicker bug finding?

Meta: Good practices

- **"Visualize" the problem/Whiteboard/Ipad**

- **Keep a log!**



- **Explain the problem to someone else.**

  Even more important is that *you are explaining the problem to yourself*

"Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind. I see what's wrong. Sorry to bother you." [Kernighan 1999]

Occam’s Razor

“The simplest explanation is often the best one.”

Before diving in too deeply, take a step back, take a breadth:

- What are the symptoms?
- Under what circumstance(s) do they occur?
  - Is the situation completely within the design envelop?
- What code is responsible for this/these circumstance(s)?
  - Look there first.

- Look for more distant explanations only after verifying that the invariants are met and that that the bug isn’t local to the code exhibiting it or the most likely places for things to have become tainted.

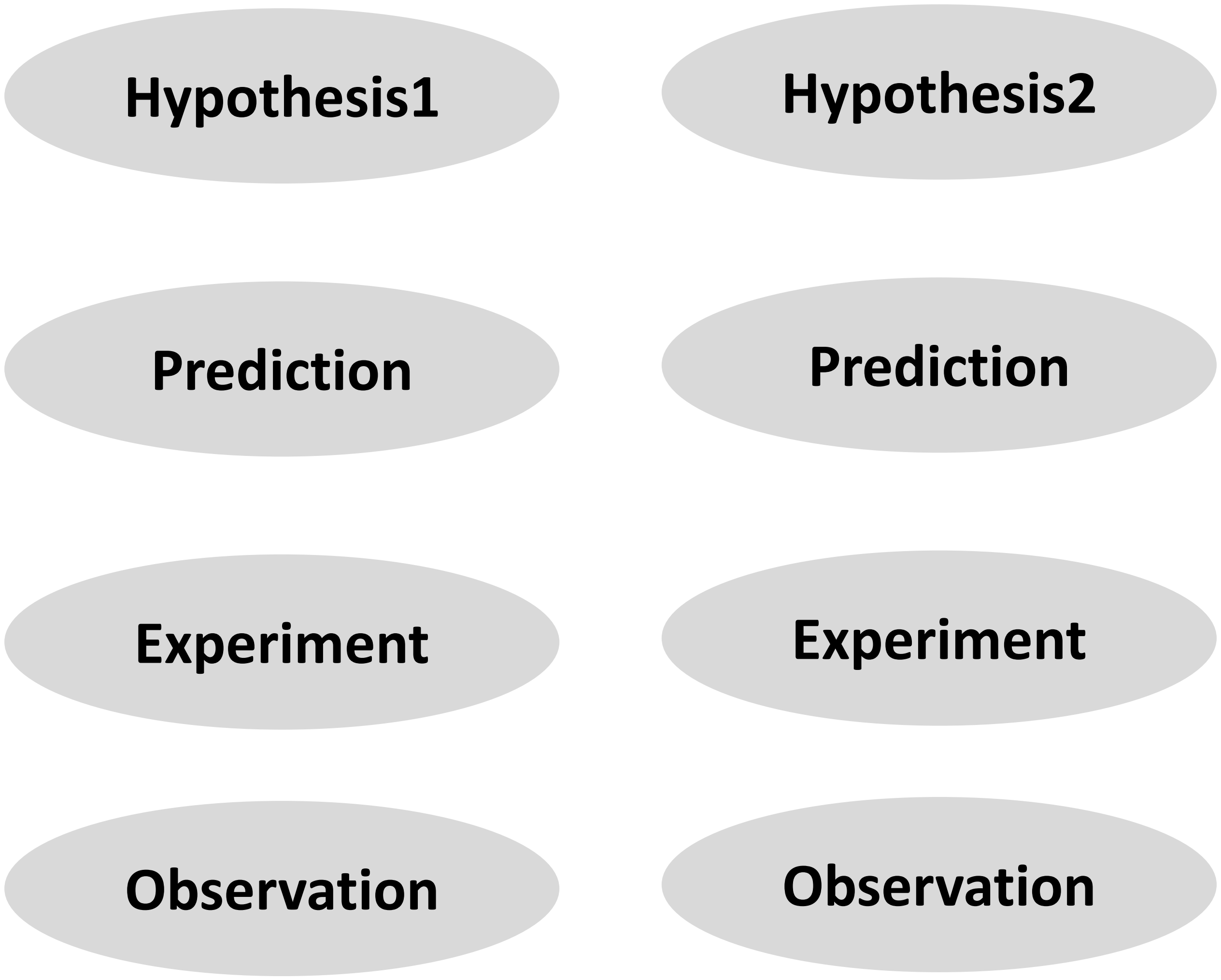
Walkthrough: Code with a Bug

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
..
  for (i = 9; i > 0; i--)
    printf("fib(%d)=%d\n",
            i, fib(i));
```

```
$ gcc -o fib fib.c
fib(9)=55
fib(8)=34
...
fib(2)=2
fib(1)=134513905
```

A defect has caused a failure.


Hypothesis1
Hypothesis2
Prediction
Prediction
Experiment
Experiment
Observation
Observation

Debugging Tools

- **Observing program state can require a variety of tools**
  - Debugger (e.g., gdb)
    - What state is in local / global variables (if known)
    - What path through the program was taken
  - Valgrind
    - Does execution depend on uninitialized variables
    - Are memory accesses ever out-of-bounds

Fix and Confirm

- **Confirm that the fix resolves the failure**
- **If you fix multiple perceived defects, which fix was for the failure?**
  - Be systematic

Learn

- **Common failures and insights**
  - Why did the code fail?
  - What are my common defects?
- **Assertions and invariants**
  - Add checks for expected behavior
  - Extend checks to detect the fixed failure
- **Testing**
  - Every successful set of conditions is added to the test suite

Common bugs (not comprehensive)

Common ways in which code is likely to have bugs, either already or in the future

- Use of uninitialized variables
- Unused values
- Unreachable code
- Duplicated code
- Bloated functions/methods
- Memory leaks
- Interface misuse
- Null pointers

Outline

- Caches (review of previous lecture)
  - Using blocking to improve temporal locality
- Debugging
  - Defects and Failures
  - Scientific Debugging
  - Tools
- **Design**
  - **Managing complexity**
  - **Communication**
  - **Naming**
  - **Comments**

Design

- **A good design needs to achieve many things:**
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

Design

- **A good design needs to achieve many things:**
  - Performance
  - Availability
  - Modifiability, portability
  - Scalability
  - Security
  - Testability
  - Usability
  - Cost to build, cost to operate

But above all else: it must be readable

Design

Good Design does:

Complexity Management &

Communication

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- Abstraction
- Information Hiding
- ...

Complexity Management: Step 0

- Make a plan!
- Break down the complex problem into simpler chunks
- Visualize or whiteboard the system “architecture” and key “functions” before writing any code!
- Write example test cases/traces even before coding!

Managing Complexity

- **Given the many ways to manage complexity**
  - Design code to be testable
  - Try to reuse testable chunks

Complexity Example

- **Split a cache access into three+ testable components**
  - State all of the steps that a cache access requires
  - Which steps depend on the operation being a load or a store?

Complexity Example

- **Split a cache access into three+ testable components**
  - State all of the steps that a cache access requires
    - Convert address into tag, set index, block offset
    - Look up the set using the set index
    - Check if the tag matches any line in the set
    - If so, hit
    - If not a match, miss, then
      - Find the LRU block
      - Evict the LRU block
      - Read in the new line from memory
    - Update LRU
    - Update dirty if the access was a store

  - Which steps depend on the operation being a load or a store?

Designs need to be testable

- **Testable design**
  - Testing versus Contracts*
  - These are complementary techniques
- **Testing and Contracts are**
  - Acts of design more than verification
  - Acts of documentation: executable documentation!

* A contract specifies in a precise and checkable way interfaces for software components: preconditions, postconditions, and object invariants.

Testing Example

- **For your cache simulator, you can write your own traces**
  - Write a trace to test for a cache hit

    ```
    L 50, 1
    L 50, 1
    ```

  - Write a trace to test dirty bytes in cache

    ```
    S 100, 1
    ```

Testable design is modular

- **Modular code has: separation of concerns, encapsulation, abstraction**
  - Leads to: reusability, extensibility, readability, testability

- **Separation of concerns**
  - Create helper functions so each function does “one thing”
  - Functions should neither do too much nor too little
  - Avoid duplicated code

- **Encapsulation, abstraction, and respecting the interface**
  - Each module is responsible for its own internals
  - No outside code “intrudes” on the inner workings of another module

Trust the Compiler!

- Use plenty of temporary variables
- Use plenty of functions
- Let compiler do the math

Communication

When writing code, the author is communicating with:

- The machine
- Other developers of the system
- Code reviewers
- Their future self

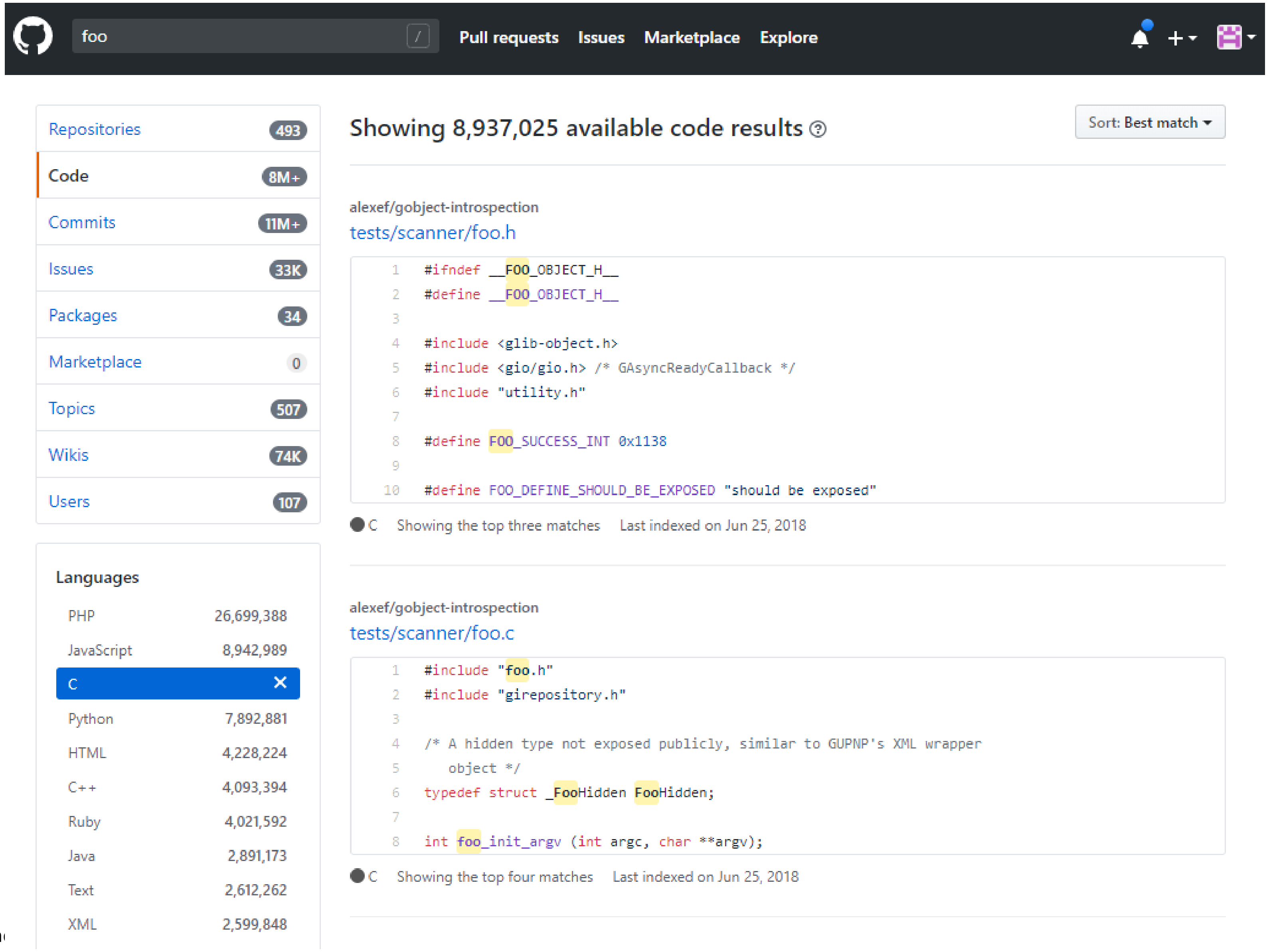
Communication

There are many techniques that have been developed around code communication:

- Tests
- Naming
- Comments
- Commit Messages
- Code Review
- Design Patterns
- …

Naming

Avoid deliberately meaningless names:



foo

Pull requests Issues Marketplace Explore

Repositories	493
Code	8M+
Commits	11M+
Issues	33K
Packages	34
Marketplace	0
Topics	507
Wikis	74K
Users	107

Languages

PHP	26,699,388
JavaScript	8,942,989
C	
Python	7,892,881
HTML	4,228,224
C++	4,093,394
Ruby	4,021,592
Java	2,891,173
Text	2,612,262
XML	2,599,848

Showing 8,937,025 available code results

Sort: Best match

alexef/gobject-introspection
tests/scanner/foo.h

```
#ifndef __FOO_OBJECT_H__
#define __FOO_OBJECT_H__

#include <glib-object.h>
#include <gio/gio.h> /* GAsyncReadyCallback */
#include "utility.h"

#define FOO_SUCCESS_INT 0x1138

#define FOO_DEFINE_SHOULD_BE_EXPOSED "should be exposed"
```

● C Showing the top three matches Last indexed on Jun 25, 2018

alexef/gobject-introspection
tests/scanner/foo.c

```
#include "foo.h"
#include "girepository.h"

/* A hidden type not exposed publicly, similar to GUPNP's XML wrapper
   object */
typedef struct _FooHidden FooHidden;

int foo_init_argv (int argc, char **argv);
```

● C Showing the top four matches Last indexed on Jun 25, 2018

Naming is understanding

"If you don't know what a thing should be called, you cannot know what it is.

If you don't know what it is, you cannot sit down and write the code."

- Sam Gardiner

Better naming practices

1. Start with meaning and intention
2. Use words with precise meanings (avoid “data”, “info”, “perform”)
3. Prefer fewer words in names
4. Avoid abbreviations in names
5. Use code review to improve names
6. Read the code out loud to check that it sounds okay
7. Actually rename things

Naming guidelines – Use dictionary words

- **Only use dictionary words and abbreviations that appear in a dictionary.**
  - For example: FileCpy -> FileCopy
  - Avoid vague abbreviations such as acc, mod, auth, etc..

Avoid using single-letter names

- **Single letters are unsearchable**
  - Give no hints as to the variable’s usage
- **Exceptions are loop counters**
  - Especially if you know why i, j, etc were originally used

Limit name character length

“Good naming limits individual name length, and reduces the need for specialized vocabulary” – Philip Relf

Limit name word count

- Keep names to a four word maximum
- Limit names to the number of words that people can read at a glance.

- Which of each pair do you prefer?

```
a1) arraysOfSetsOfLinesOfBlocks

a2) cache


b1) evictedData

b2) evictedDataBytes
```

Describe Meaning

- Use descriptive names.
- Avoid names with no meaning: a, foo, blah, tmp, etc

- There are reasonable exceptions:

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Use a large vocabulary

- **Be more specific when possible:**
  - Person -> Employee

- **What is size in this binaryTree?**

```
struct binaryTree {
  int size;
  …
};
```

```
height
numChildren
subTreeNumNodes
keyLength
```

Use problem domain terms

- **Use the correct term in the problem domain's language.**
  - Hint: as a student, consider the terms in the assignment

- **In cachelab, consider the following:**

```
line

element
```

Use opposites precisely

- **Consistently use opposites in standard pairs**
  - first/end -> first/last

Comments

Don’t Comments

- **Don’t say what the code does**
  - because the code already says that
- **Don’t explain awkward logic**
  - improve the code to make it clear
- **Don’t add too many comments**
  - it’s messy, and they get out of date

Awkward Code

- **Imagine someone (TA, employer, etc) has to read your code**
  - Would you rather rewrite or comment the following?

```
(*(void **)((*(void **)(bp)) + DSIZE)) = (*(void **)(bp + DSIZE));
```

  - How about?

```
bp->prev->next = bp->next;
```

  - Both lines update program state in the same way.

Do Comments

- Answer the question: why the code exists

- When should I use this code?
- When shouldn’t I use it?
- What are the alternatives to this code?

Why does this exist?

- **Explain why a magic number is what it is.**

```
// Each address is 64-bit, which is 16 + 1 hex characters
const int MAX_ADDRESS_LENGTH = 17;
```

- **When should this code be used? Is there an alternative?**

```
unsigned power2(unsigned base, unsigned expo){
    unsigned i;
    unsigned result = 1;
    for(i=0;i<expo;i++){
        result+=result;
    }
    return result;
}
```

How to write good comments

1. **Write short comments of what the code will do.**
   1. Single line comments
   2. Example: Write four one-line comments for quick sort

```
// Initialize locals
// Pick a pivot value
// Reorder array around the pivot
// Recurse
```

How to write good comments

1. **Write short comments of what the code will do.**
   1. Single line comments
   2. Example: Write four one-line comments for quick sort

2. **Write that code.**

3. **Revise comments / code**
   1. If the code or comments are awkward or complex
   2. Join / Split comments as needed

4. **Maintain code and comments**

Commit Messages

- **Committing code to a source repository is a vital part of development**
  - Protects against system failures and typos:
    - cat foo.c versus cat > foo.c
  - The commit messages are your record of your work
    - Communicating to your future self
    - Describe in one line what you did

  “Parses command line arguments”

  “fix bug in unique tests, race condition not solved”

  “seg list finished, performance is …”
- **Use branches**

Summary

- **Programs have defects**
  - Be systematic about finding them
- **Programs are more complex than humans can manage**
  - Write code to be manageable
- **Programming is not solitary, even if you are communicating with a grader or a future self**
  - Be understandable in your communication

Acknowledgements

- **https://www.debuggingbook.org/**
- **Some debugging content derived from:**
  - http://www.whyprogramsfail.com/slides.php
- **Some code examples for design are based on:**
  - "The Art of Readable Code". Boswell and Foucher. 2011.
- **Lecture originally written by**
  - Michael Hilton and Brian Railing