COMP2310/COMP6310
Systems, Networks, & Concurrency

Convener: Prof John Taylor

After this lecture

- **You will be able to:**
  - Describe the steps to debug complex code failures
  - Identify ways to manage the complexity when programming
  - State guidelines for communicating the intention of the code

Outline

- **Debugging**
  - **Defects and Failures**
  - **Scientific Debugging**
  - **Tools**
- Design
  - Managing complexity
  - Communication
  - Naming
  - Comments

Atlas-Centaur

- **Centaur second stage failed after entering an uncontrolled spin**
  - Investigation - turbopumps relied on gas expansion and clogged from plastic remnants of scouring pads
  - Proposed Solution - Bake off plastic
- **Next launch – second stage failed after entering an …**
  - Further investigation – a valve had been leaking for years
    - Increased need for engine efficiency pushed this leak into failure range
- **What happened?**
  - The second time they reproduced the failure

https://www.thespacereview.com/article/1321/1

Defects, Errors, & Failures

1. The programmer creates a defect (or a fault)
2. The defect (maybe) causes an error

   wrong results in data values or control signals

3. The error propagates
4. The error causes a failure

   a component or system does not produce the intended result at an interface

Why is an error not necessarily a failure? Because errors can be masked or detected. Example: ECC memory.

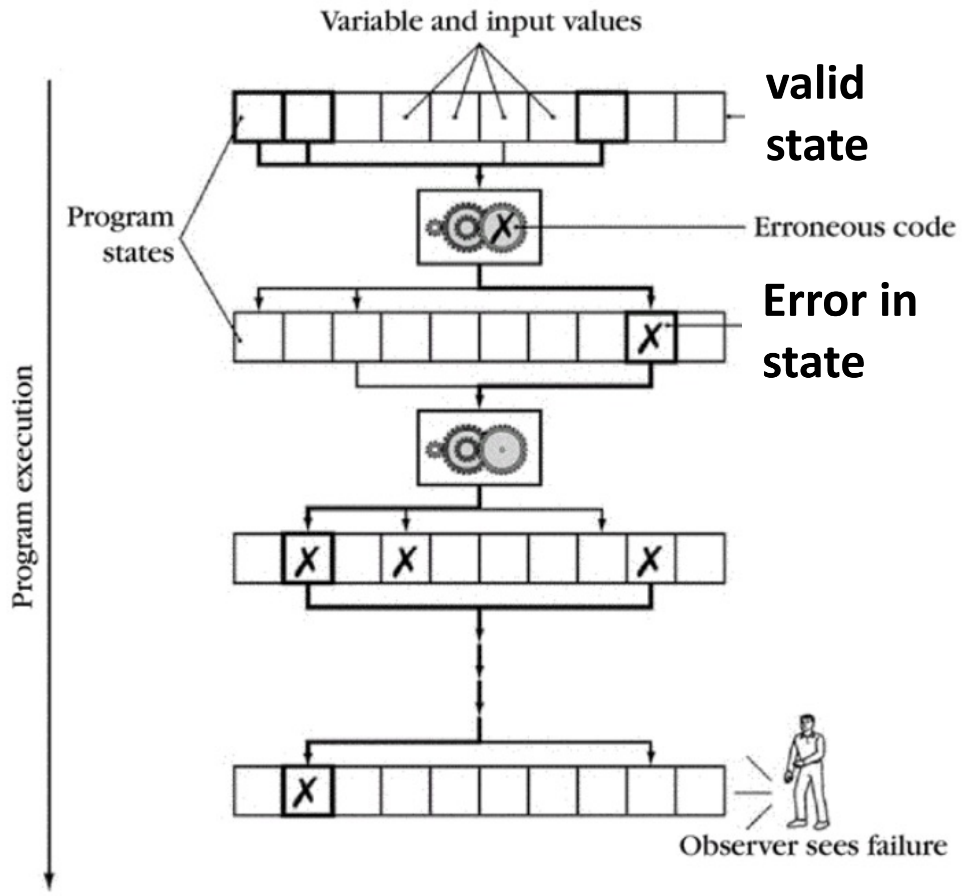
Curse of Debugging

- **Not every defect causes a failure!**
  - A defect can be **latent** or **active**
  - A defect in code that doesn’t get executed most of the time…

- **Testing can only show the presence of [defects] – not their absence. (Dijkstra 1972)**

Defects to Failures

- **Code with defects will introduce erroneous state or control**
  - Correct code may propagate this state
  - Eventually an erroneous state is observed
- **Some executions will not trigger the defect**
  - Others will not propagate erroneous state
- **Debugging sifts through the code to find the defect**

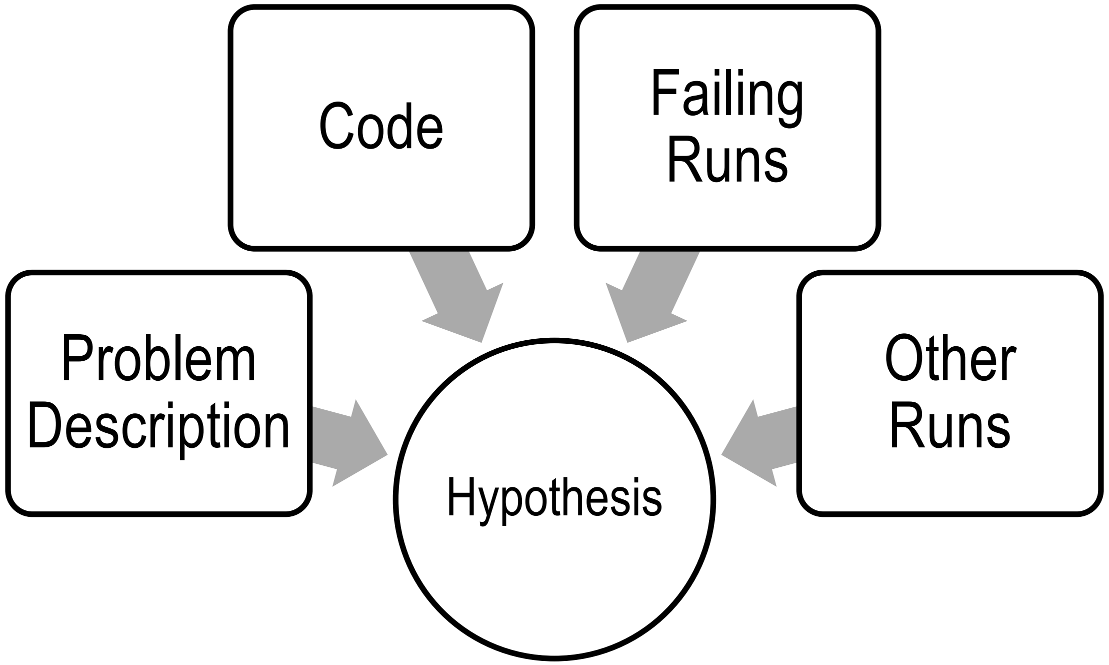


Explicit Debugging

- **Stating the problem**
  - Describe the problem aloud or in writing
    - A.k.a. “Rubber duck” or “teddy bear” method
  - Often a comprehensive problem description is sufficient to solve the failure

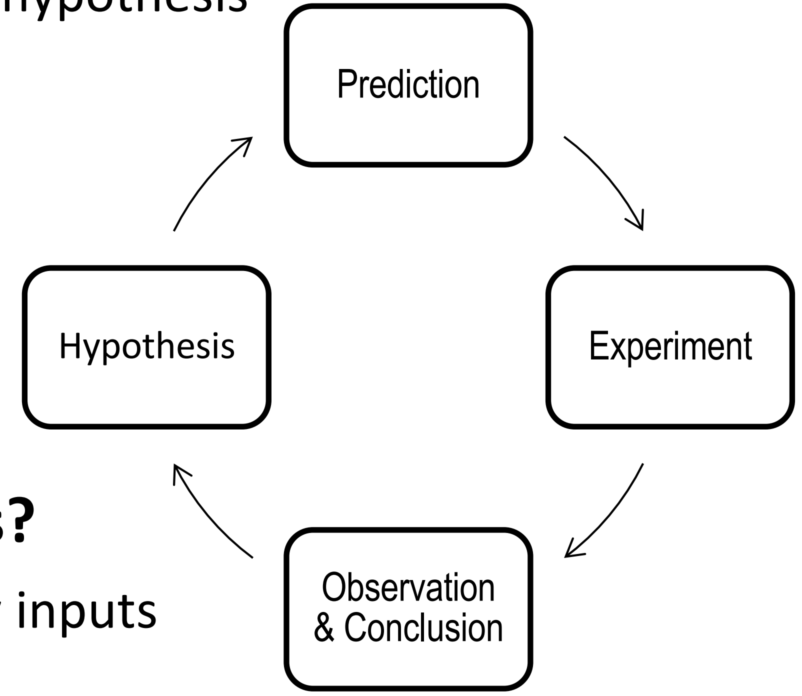
Scientific Debugging

- **Before debugging, you need to construct a hypothesis as to what is the defect**
  - Propose a possible defect and why it explains the failure conditions
  - Don’t have an idea? What experiments would give you useful info?
- **Occam’s Razor – given several hypotheses, pick the simplest / closest to current work**



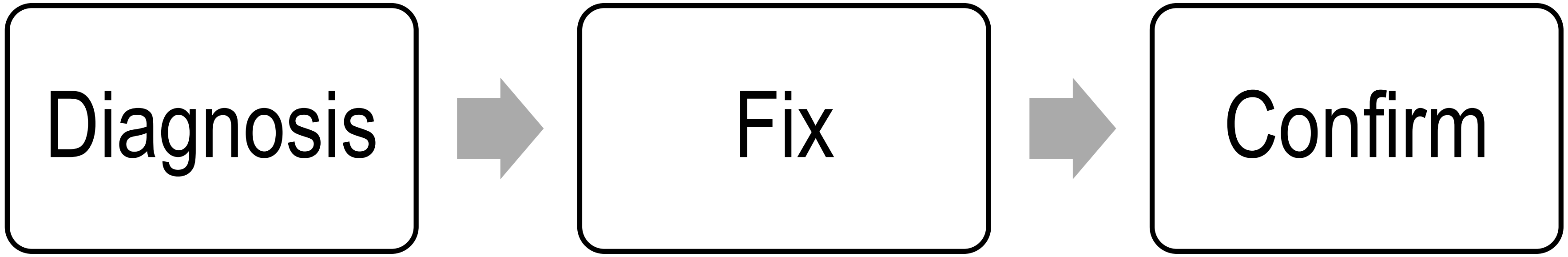
Scientific Debugging

- **Make predictions based on your hypothesis**
  - What do you expect to happen under new conditions
  - What data could confirm or refute your hypothesis
- **How can I collect that data?**
  - What experiments?
  - What collection mechanism?
- **Does the data refute the hypothesis?**
  - Refine the hypothesis based on the new inputs



Scientific Debugging

- **A set of experiments has confirmed the hypothesis**
  - This is the diagnosis of the defect
- **Develop a fix for the defect**
- **Run experiments to confirm the fix**
  - Otherwise, how do you know that it is fixed?
  - In the real world, you often add a test here



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.

How do we know it's a failure?
It violates the spec.
First, know what SHOULD happen.

Constructing a Hypothesis

- **Specification defined the first Fibonacci number as 1**
  - We have observed working runs (e.g., fib(2))
  - We have observed a failing run
  - We then read the code

- **fib(1) failed // Hypothesis**

Code	**Hypothesis**
for (i = 9; …)	Result depends on order of calls
while (n > 1) {	Loop check is incorrect
int f;	F is uninitialized

Brute Force Approach

- **First, compilation flags**
  - MUST include “-Wall”
  - Should include “-Werror”

```
Prompt> gcc -Wall -Werror -O3 -o badfib badfib.c
badfib.c: In function ‘fib’:
badfib.c:12:5: error: ‘f’ may be used uninitialized in this
function [-Werror=maybe-uninitialized]
     return f;
     ^
cc1: all warnings being treated as errors
```

Brute Force Approac

- **First, compilation flags: “–W**
  - MUST include “-Wall”
  - Should include “-Werror”
- **Second, other optimization**
  - Try at least –O3 and –O0

```
prompt>gcc  -O3 -o badfib badfib.c
prompt>./badfib
...
fib(2)=2
fib(1)=0
fib(0)=0
prompt>gcc  -O2 -o badfib badfib.c
prompt>./badfib
...
fib(2)=2
fib(1)=0
fib(0)=0
prompt>gcc  -O1 -o badfib badfib.c
prompt>./badfib
...
fib(2)=2
fib(1)=9
fib(0)=9
prompt>gcc  -O0 -o badfib badfib.c
prompt>./badfib
...
fib(2)=2
fib(1)=2
fib(0)=2
```

Brute Force Approach

- **First, compilation flags: “`-Wall -Werror`”**
  - MUST include “-Wall”
  - Should include “-Werror”
- **Second, other optimization levels**
  - Try at least –O3 and –O0
- **Valgrind (even if your program appears to be working!)**
  - Run on both –O3 and –O0
  - Only run after all warnings are gone!

```
prompt> gcc -g  -O3 -o badfib badfib.c
prompt> valgrind badfib
==1462== Memcheck, a memory error detector
==1462== Copyright (C) 2002-2017, and GNU GPL'd, by Julia
==1462== Using Valgrind-3.13.0 and LibVEX; rerun with -h
==1462== Command: badfib
==1462==
fib(9)=55
fib(8)=34
fib(7)=21
fib(6)=13
fib(5)=8
fib(4)=5
fib
fib(2)=2
fib(1)=0
fib(0)=0
==1462==
```

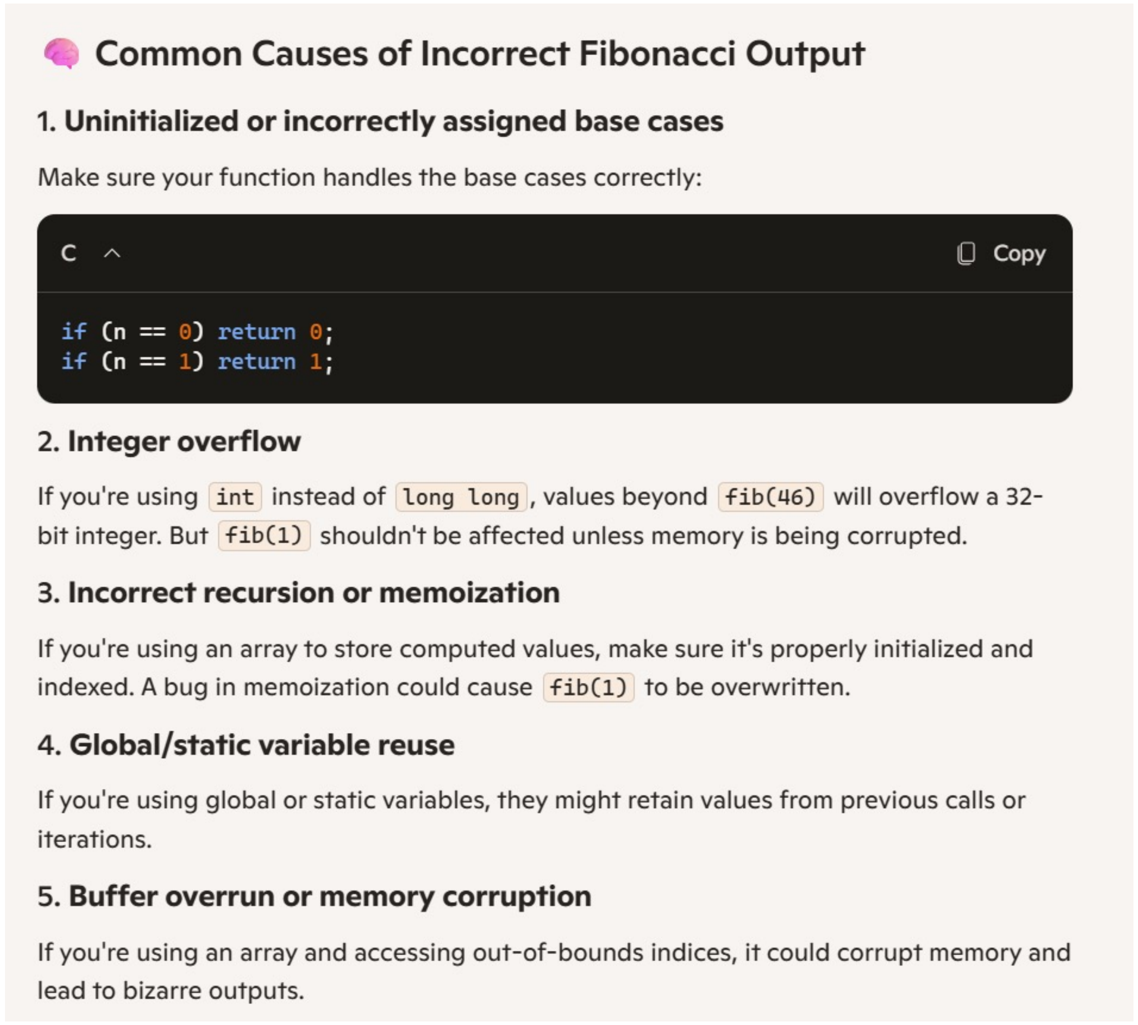
Valgrind is not perfect. On –O3 it finds no errors!

```
prompt> gcc -g  -O0 -o badfib badfib.c
prompt> valgrind badfib
==1561== Memcheck, a memory error detector
==1561== Copyright (C) 2002-2017, and GNU GPL'd, by Julia
==1561== Using Valgrind-3.13.0 and LibVEX; rerun with -h
==1561== Command: badfib
==1561==
fib(9)=55
fib(8)=34
fib(7)=21
fib(6)=13
fib(5)=8
fib(4)=5
fib
```

Valgrind is not perfect, but pretty good.

```
fib(2)=2
==1561== Conditional jump or move depends on uninitialise
==1561==    at 0x4E988DA: vfprintf (vfprintf.c:1642)
==1561==    by 0x4EA0F25: printf (printf.c:33)
```

LLM Gives Hypotheses Also



Common Causes of Incorrect Fibonacci Output

1. Uninitialized or incorrectly assigned base cases

Make sure your function handles the base cases correctly:

```c
if (n == 0) return 0;
if (n == 1) return 1;
```

2. Integer overflow

If you're using `int` instead of `long long`, values beyond `fib(46)` will overflow a 32-bit integer. But `fib(1)` shouldn't be affected unless memory is being corrupted.

3. Incorrect recursion or memoization

If you're using an array to store computed values, make sure it's properly initialized and indexed. A bug in memoization could cause `fib(1)` to be overwritten.

4. Global/static variable reuse

If you're using global or static variables, they might retain values from previous calls or iterations.

5. Buffer overrun or memory corruption

If you're using an array and accessing out-of-bounds indices, it could corrupt memory and lead to bizarre outputs.

Constructing a Hypothesis

- **Specification defined the first Fibonacci number as 1**
  - We have observed working runs (e.g., fib(2))
  - We have observed a failing run
  - We then read the code

- **fib(1) failed // Hypothesis**

Code	**Hypothesis**
for (i = 9; …)	Result depends on order of calls
while (n > 1) {	Loop check is incorrect
int f;	F is uninitialized

Prediction

- **Propose a new condition or conditions**
  - What will logically happen if your hypothesis is correct?
  - What data can be

- **fib(1) failed // Hypothesis**
  - // Result depends on order of calls
    - If fib(1) is called first, it will return correctly.
  - // Loop check is incorrect
    - Change to n >= 1 and run again.
  - // f is uninitialized
    - Change to int f = 1;

Experiment

- **Identical to the conditions of a prior run**
  - Except with one condition changed
- **Conditions**
  - Program input, using a debugger, altering the code

- **fib(1) failed // Hypothesis**
  - If fib(1) is called first, it will return correctly.
    - Fails.
  - Change to n >= 1
    - fib(1)=2
    - fib(0)=...
  - Change to int f = 1;
    - Works. Sometimes a prediction can be a fix.

Observation

- **What is the observed result?**
  - Factual observation, such as “Calling fib(1) will return 1.”
  - The conclusion will interpret the observation(s)
- **Don’t interfere.**
  - printf() can interfere for some kinds of bugs!
  - Like quantum physics, sometimes observations are part of the experiment
- **Proceed systematically.**
  - Update the conditions incrementally so each observation relates to a specific change
- **Do NOT ever proceed past first bug.**

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

Diagnosis

- **A scientific hypothesis that explains the current observations and makes future predictions becomes a theory**
  - We'll call this a diagnosis
- **Use the diagnosis to develop a fix for the defect**
  - Avoid *post hoc, ergo propter hoc* fallacy
  - Or correlation does not imply causation
- **Understand why the defect and fix relate**

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
    - N.b., Assertions must not have side effects
  - Extend checks to detect the fixed failure

- **Testing**
  - Every successful set of conditions is added to the test suite

Quick and Dirty

- **Not every problem needs scientific debugging**
  - Set a time limit: (for example)
    - 0 minutes – -Wall, valgrind
    - 1 – 10 minutes – Informal Debugging
    - 10 – 60 minutes – Scientific Debugging
    - > 60 minutes – Take a break / Ask for help

Common Bugs…

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Outline

- 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

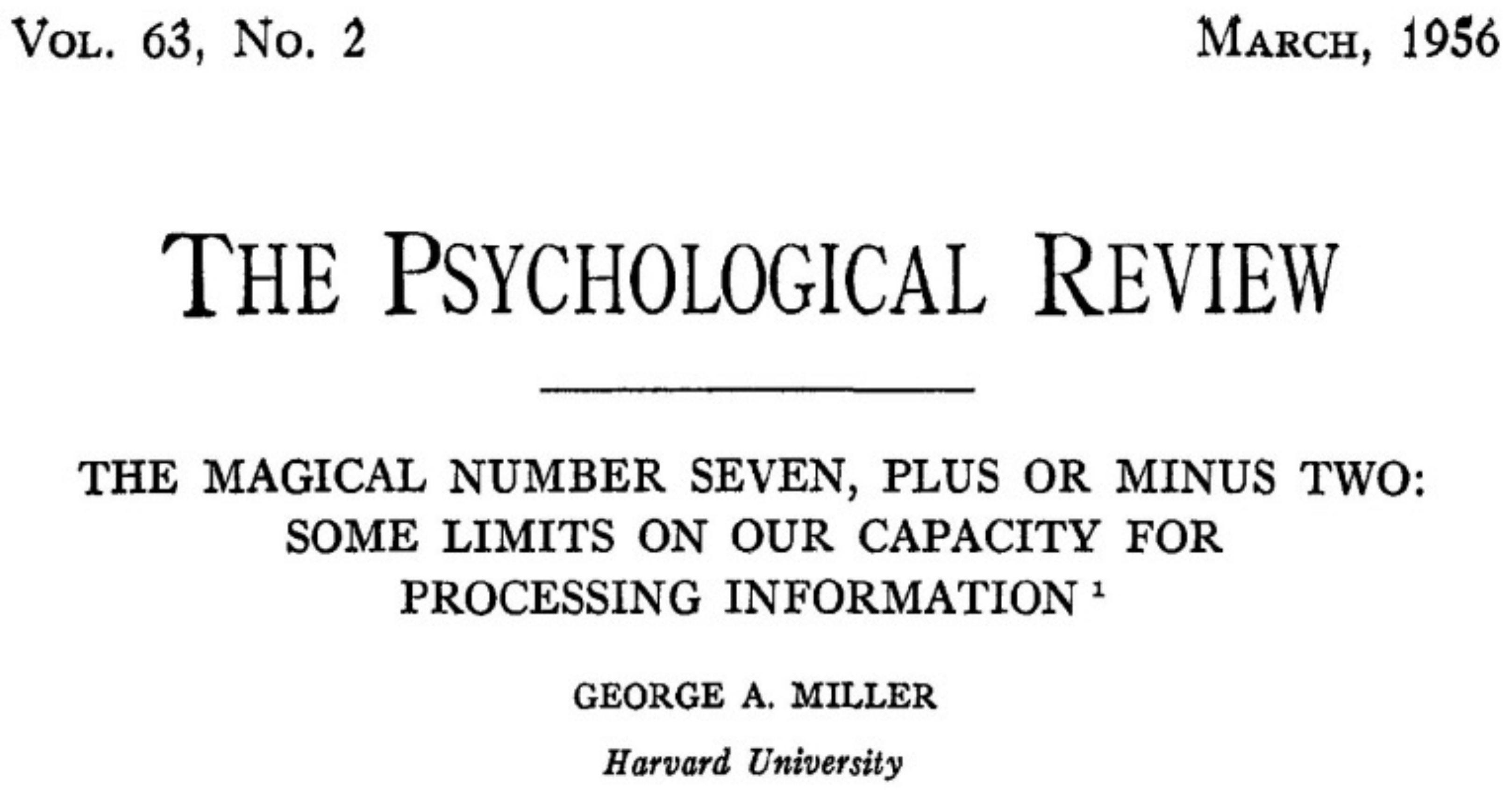
Good Design does:

Complexity Management &

Communication

Complexity

- There are well known limits to how much complexity a human can manage easily.



Vol. 63, No. 2 March, 1956

THE PSYCHOLOGICAL REVIEW

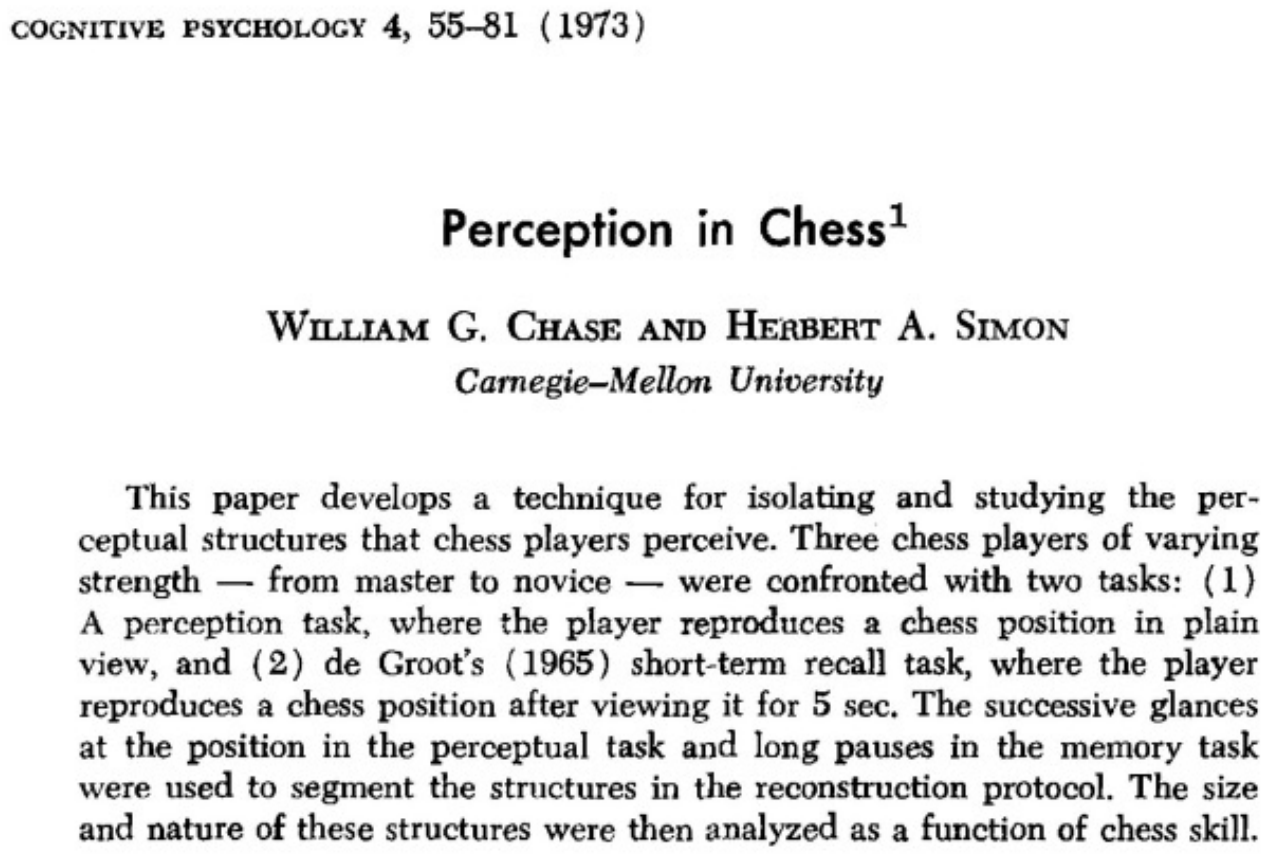
THE MAGICAL NUMBER SEVEN, PLUS OR MINUS TWO: SOME LIMITS ON OUR CAPACITY FOR PROCESSING INFORMATION [1]

GEORGE A. MILLER

Harvard University

Complexity Management

- However, patterns can be very helpful...



COGNITIVE PSYCHOLOGY 4, 55–81 (1973)

Perception in Chess[1]

WILLIAM G. CHASE AND HERBERT A. SIMON

Carnegie–Mellon University

This paper develops a technique for isolating and studying the perceptual structures that chess players perceive. Three chess players of varying strength — from master to novice — were confronted with two tasks: (1) A perception task, where the player reproduces a chess position in plain view, and (2) de Groot's (1965) short-term recall task, where the player reproduces a chess position after viewing it for 5 sec. The successive glances at the position in the perceptual task and long pauses in the memory task were used to segment the structures in the reconstruction protocol. The size and nature of these structures were then analyzed as a function of chess skill.

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- Don’t repeat yourself - DRY
- Abstraction
- Information Hiding
- ...

Managing Complexity

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

Complexity Example

- **Split a wordle tool access into three+ testable components**
  - State all of the steps that the tool requires

Complexity Example

- **Split a wordle tool access into three+ testable components**
  - State all of the steps that the tool requires
    - Read in dictionary file
      - Create struct to hold each word
      - Is that struct appropriate for complexity / input?
    - Read in guess + green / yellow from stdin
      - Search for other words that would fit
      - Sort words based on maximum information gained

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

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!

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

The compiler can be helpful!

- 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**

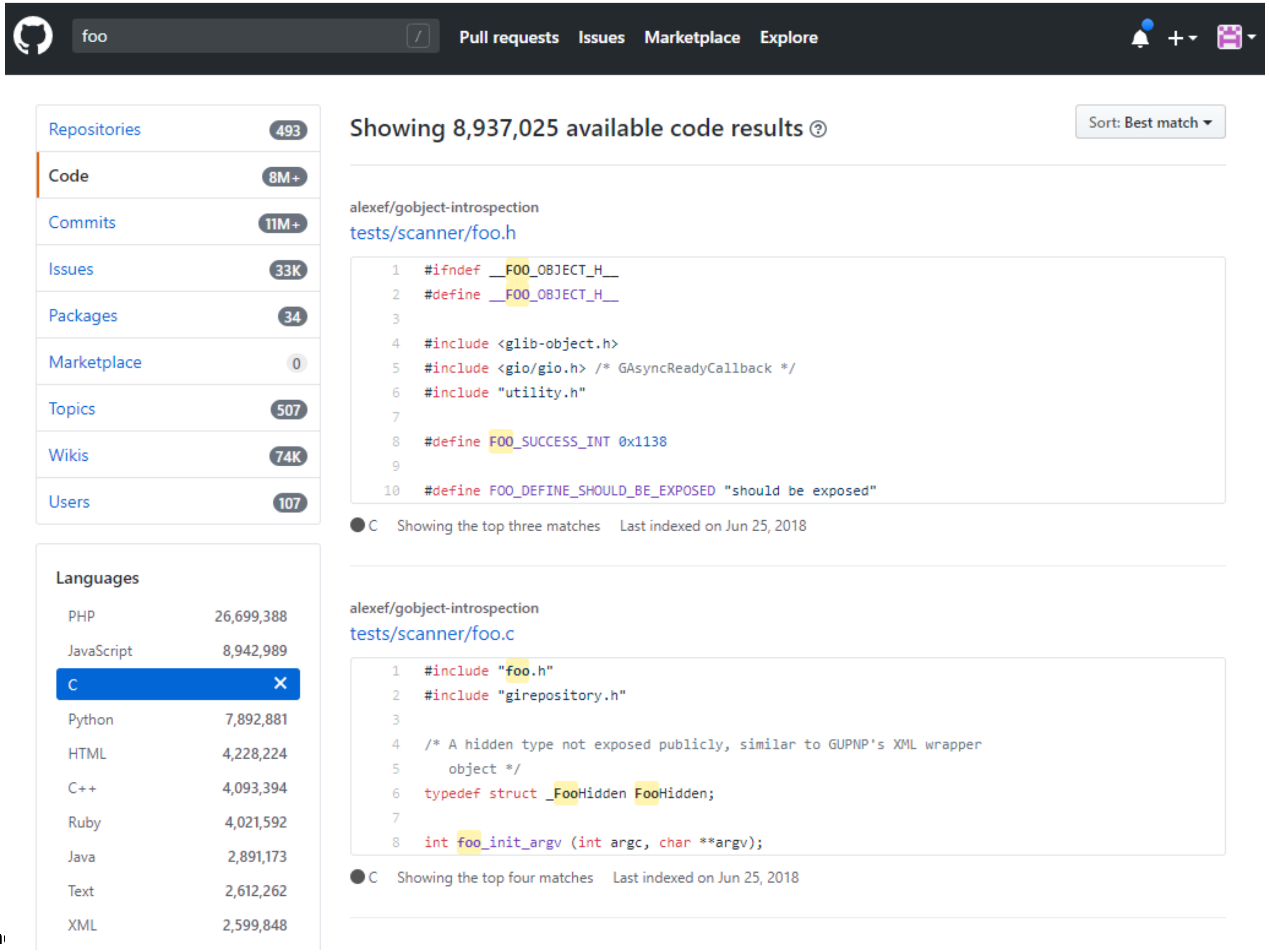
Communication

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:



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

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 opposites precisely

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

Comments

Don’t Comments

- **Don’t restate 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?

How to write good comments

1. **Code by commenting!**
   Write short comment
   1. Helps you think about design & overcome blank-page problem
   2. Single line comments
   3. 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 revised 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

- **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