

# CS 3110

## Testing

Prof. Clarkson

Fall 2017

Today's music: Wrecking Ball by Miley Cyrus

# Review

## **Previously in 3110:**

- Modules
- Specification (functions, modules)

## **Today:**

- Validation
- Testing
  - Black box
  - Glass box
  - Randomized

# Validation

- **Validation:** does program behave as intended?
- **Testing:** a process for validation
- **Debugging:** determining cause of unintended behavior
- **Defensive programming:** implementation techniques for making validation and debugging easier

# Approaches to validation

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - Static analysis  
("lint" tools, FindBugs, ...)
  - Fuzzers
- Mathematical
  - Type systems
  - Formal verification



Less formal: Techniques may miss problems in programs

All of these methods should be used!

Even the most formal can still have holes:

- did you prove the right thing?
- do your assumptions match reality?

More formal: eliminate *with certainty* as many problems as possible.

# Testing vs. Verification

## Testing:

- Cost effective
- Guarantee that program is correct on **tested** inputs and in **tested** environments

## Verification:

- Expensive
- Guarantee that program is correct on **all** inputs and in **all** environments

# Edsger W. Dijkstra



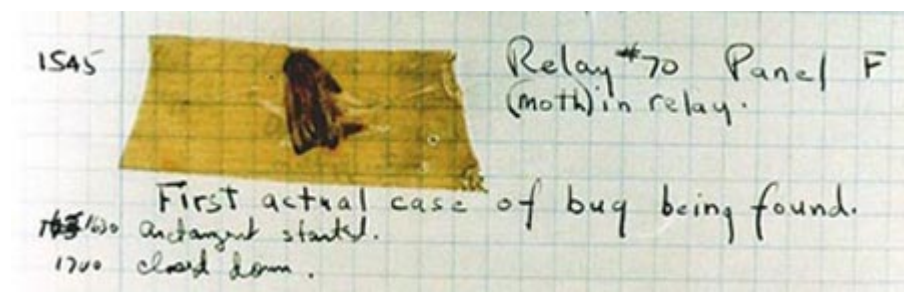
(1930-2002)

**Turing Award Winner (1972)**

*For eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness*

"Program testing can at best show the presence of errors but never their absence."

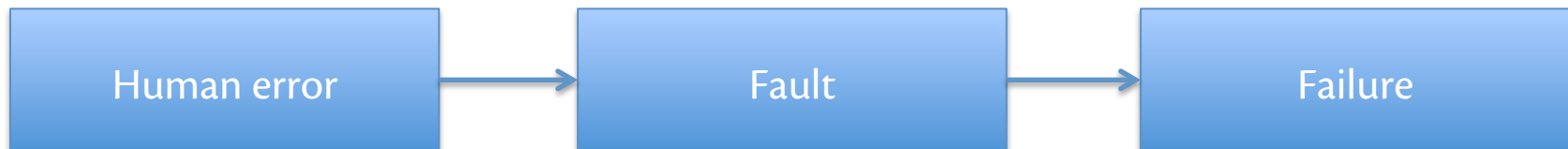
# Bugs



"bug": suggests something just wandered in

[IEEE 729]

- **Fault:** result of human error in software system
  - E.g., implementation doesn't match design, or design doesn't match requirements
  - Might never appear to end user
- **Failure:** violation of requirement
  - Something goes wrong for end user



# Testing

- Goal is to expose existence of faults, so that they can be fixed
- **Unit testing:** isolated components
- **Integration testing:** combined components
- **System testing:** functionality, performance, acceptance, installation



# Regression testing

- **Regression:** a previously fixed fault is reintroduced into the code
- **Regression testing:** running tests against new version of software to ensure no regressions
- If you ever find and fix a fault...
  - Put a test case into your suite for it
  - Run suite frequently to detect regressions

# Question

When should you stop testing?

- A. When the assignment is due in 1 hour
- B. When all the examples in the writeup finally pass
- C. When you have at least five test cases for each function
- D. When all your own test cases finally pass
- E. None of the above

# Question

When should you stop testing?

- A. When the assignment is due in 1 hour
- B. When all the examples in the writeup finally pass
- C. When you have at least five test cases for each function
- D. When all your own test cases finally pass
- E. None of the above**

# Testing

When do you stop testing?

- **Bad answer:** when time is up
- **Bad answer:** what all tests pass

# Fun fact

*Pr[undetected faults]*  
increases  
with # *detected faults*

[Myers 1979, 2004]

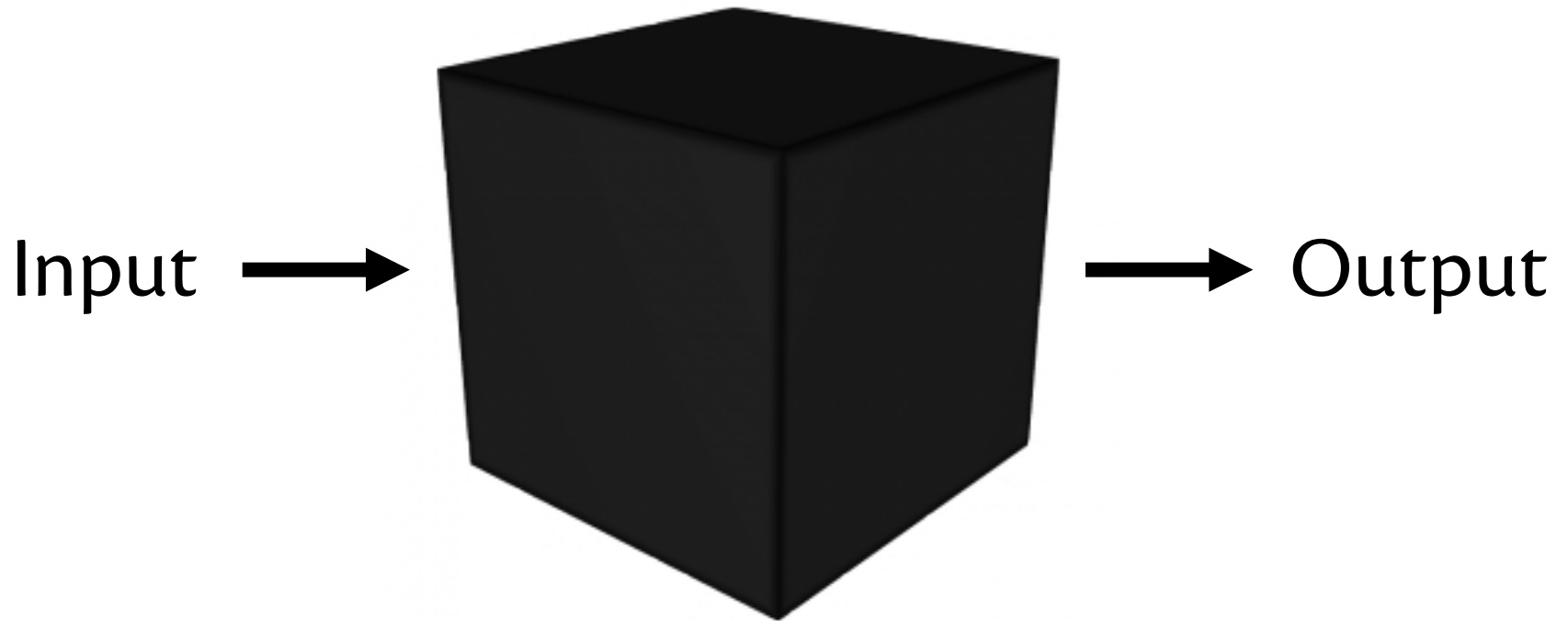
# Testing

When do you stop testing?

- **Good answer:** when testing methodology is complete
- **Future answer:** statistical estimation says  $Pr[\textit{undetected faults}]$  is low enough (active research)

**TESTING**

# Black box testing



tester knows nothing about internals of functionality being tested

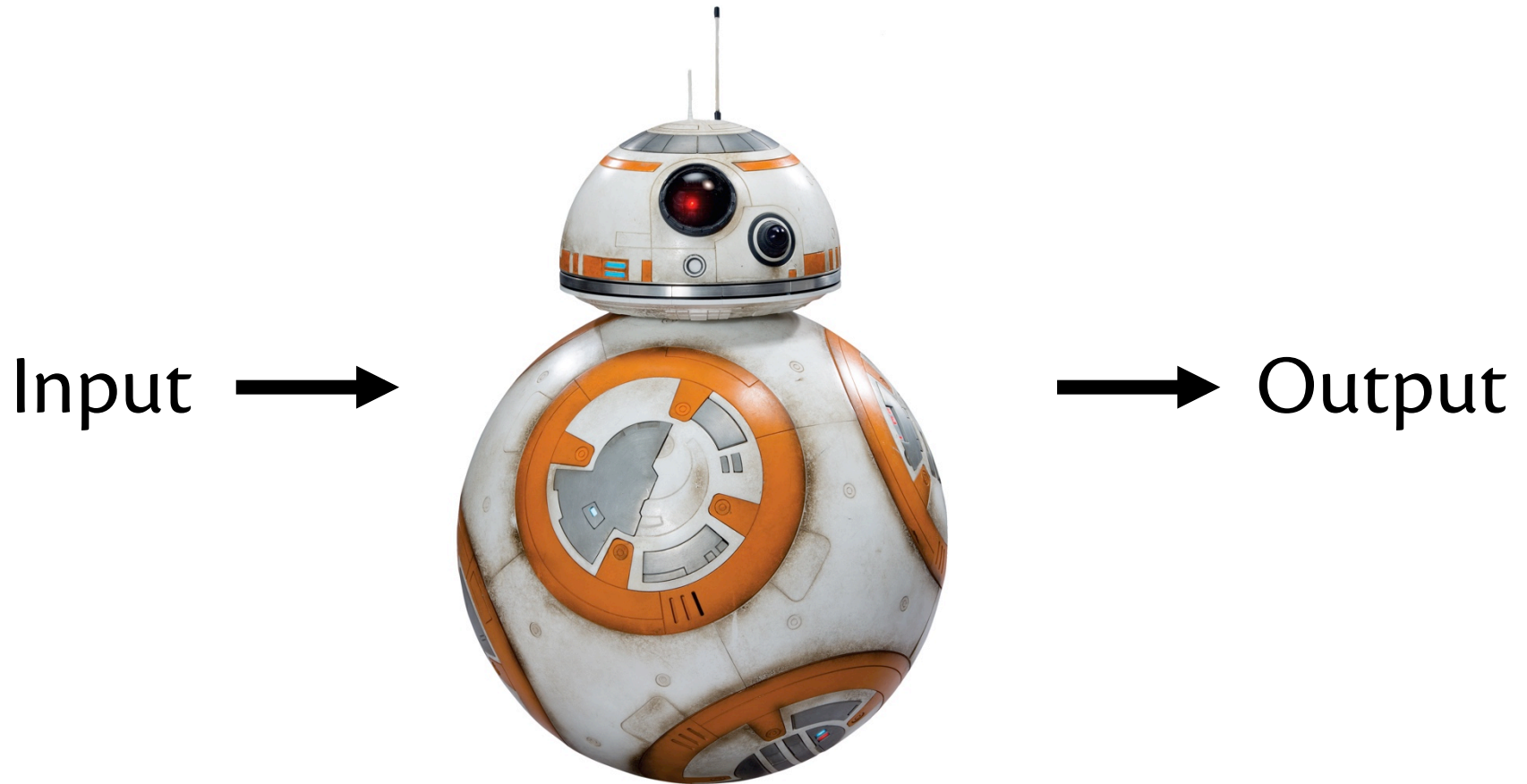


# Glass box testing



tester knows internals of functionality being tested

# Black box testing



tester knows nothing about internals of functionality being tested

# Glass box testing

Input →



→ Output

tester knows internals of functionality being tested

# Black box testing

- Tests are based on the **specification**
- **Advantages:**
  - Tester is not biased by assumptions made in implementation
  - Tests are robust w.r.t. changes in implementation
  - Tests can be read and evaluated by reviewers who are not implementers
- Main kinds of black box tests:
  - Example inputs provided by spec
  - Typical inputs
  - Boundary cases
  - Paths through spec

# Typical inputs

- Common, simple values of a type
  - **int**: small integers like 1 or 10
  - **char**: alphabetic letters, digits
  - **string**: whose length is a small integer and whose characters are typical
  - '**a list**': a small integer number of elements, each of which is a typical value of type '**a**'
  - **records/tuples**: each field/component with a typical value
  - **variants**: typical constructors, if there is such a thing

# Boundary cases



A QA Engineer walks into a bar.  
Orders a beer.  
Orders 0 beers.  
Orders 999999999 beers.  
Orders a lizard.  
Orders -1 beers.  
Orders a sdfeljknesv.

@sempf

# Boundary cases

- aka *corner cases* or *edge cases*
- Atypical or extremal values of a type, and values nearby
  - **int**: 0, 1, -1, **min\_int**, **max\_int**
  - **char**: '**\000**', '**\255**', '**\032**' (space), '**\127**' (delete)
  - **string**: empty string, string with a single character, unreasonably long string
  - '**a list**': empty list, list with a single element, list with enough elements to trigger stack overflow on non-tail-recursive functions
  - **records/tuples**: combinations of atypical values
  - **variants**: all constructors

# Paths through spec

## Representative inputs for classes of outputs

```
(* [is_prime n] is true iff [n] is prime *)  
val is_prime: int -> bool
```

two classes of output:

- true: representative input: n=13
- false: representative input: n=42

other examples:

- **compare** functions have three classes of output
- functions that return variants have several classes of output



# Paths through spec

Representative inputs for each way of satisfying the precondition

```
(* [sqrt x n] is the square root of [x]  
  * computed to an accuracy of [n]  
  * significant digits  
  * requires: x >= 0 and n >= 1 *)  
val sqrt : float -> int -> float
```

(i)  $x=0.0$ ,  $n=1$ , (ii)  $x=1.0$ ,  $n=1$ ,  
(iii)  $x=0.0$ ,  $n=2$ , (iv)  $x=1.0$ ,  $n=2$

# Paths through spec

Representative inputs for each way of raising and not raising exception

```
(* [pos x lst] is the 0-based position of  
 * the first element of [lst] that equals [x].  
 * raises:  Not_found if [x] is not in [lst]. *)  
val pos: 'a -> 'a list -> int
```

(i)  $x=1$ ,  $lst=[1]$ , (ii)  $x=0$ ,  $lst[1]$

# Glass box testing

- aka *white box testing*
- **Advantages:**
  - can determine whether a new test case really yields additional information about correctness of implementation
  - can address likely errors that are not apparent from specification
- **Supplements** black-box testing; does not **replace** examination of specification
- Main kind of glass box test cases:
  - *paths through implementation* aka *path coverage*

# Paths through implementation

All execution paths through implementation are tested

```
let max3 x y z =  
  if x>y then  
    if x>z then x else z  
  else  
    if y>z then y else z
```

Testing according to black-box specification might lead to all kinds of inputs

But there are really only four paths through implementation!

Representatives: (i) 3 2 1, (ii) 3, 2, 4, (iii) 1, 2, 1, (iv) 1, 2, 3

# Achieving path coverage

- Include test cases for:
  - each branch of each (nested) if expression
  - each branch of each (nested) pattern match
- Particularly watch out for:
  - base cases of recursive function
  - recursive calls in recursive function
  - every place where an exception might be raised

# Testing data abstractions

- Some functions of a data abstraction *produce* a value of it
  - **empty** produces an empty set
  - **union** produces a set
- Other functions *consume* a value
  - **size** consumes a dictionary and produces an integer
  - **bindings** consumes a dictionary and produces a list
- For every possible path through spec and impl of producers... test how a consumer handles it
  - e.g. if producers of a set handle sets of size 0, 1, and >1 differently...
  - then test each such set with every consumer
- For every value returned by abstraction, check the RI

# Randomized testing

- *"It was a dark and stormy night..."*
- Generate **random inputs** and feed them to programs:
  - Crash? hang? terminate normally?
  - Of ~90 utilities in '89, crashed about 25-33% in various Unixes
  - Crash => buffer overflow potential
- Since then, "fuzzing" has become a standard practice for security testing
- Results have been repeated for X-windows system, Windows NT, Mac OS X
  - **Results keep getting worse in GUIs** but better on command line

# Upcoming events

- N/A

*This is saving your code from being rekt.*

**THIS IS 3110**



Advice on

# DEBUGGING

# Debugging

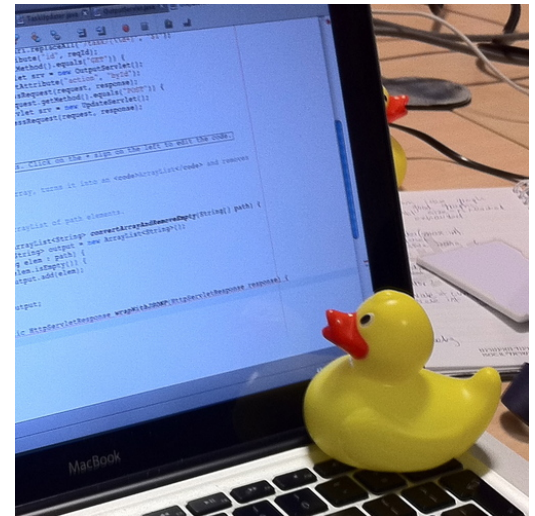
- *Testing* reveals a fault in program
- *Debugging* reveals the cause of that fault
- "Bug" is misleading
  - it didn't just crawl into program
  - **programmer put it there**
- Debugging takes more time than programming
  - get it right the first time!
  - understand exactly why you expect code to work before testing/debugging it

# Debugging advice

- Follow the scientific method:
  - formulate a falsifiable hypothesis
  - create an experiment that can refute that hypothesis
  - run that experiment
  - keep a lab notebook
- Find the simplest possible input that causes fault
- Find the first manifestation of inappropriate behavior

# Debugging advice

- Invest effort in writing code to help you understand intermediate results
- The bug is probably not where you think it is...ask yourself where it cannot be
- Get someone else to help you



# Debugging advice

- If all else is failing, doubt your sanity: do you have the right compiler? the right source code
- Don't debug when angry or tired: give it a break; come back refreshed
- Think through the fix carefully: "fixing" a bug often leads to new bugs

# Defensive programming

- *Proactive debugging*: make it easier to detect faults by writing fault-detection code during implementation
- Techniques:
  - Asserting preconditions
  - Asserting (rep) invariants
  - Exhaustive conditionals
    - check for all the possible values in an `if` or `match`
    - don't assume that `x <> a` means `x = b`

# Defensive programming

**Q:** “Isn’t this expensive?”

**A:** It only seems that way!

- For **implementer**: the defensive code you write tends to pay off in the faults it catches early
- For **performance**: the faults you catch in production might save more money than the run-time cost of the checks