Functional Testing, Random Testing and Fuzzing

Functional testing

- Functional testing: Deriving test cases from program specifications
    - *Functional* refers to the source of information used in test case design, not to what is tested
- *Also known as*:
    - specification-based testing (from specifications)
    - black-box testing (no view of the code)
- Functional specification = description of intended program behavior
    - either formal or informal

Systematic vs Random Testing

- Random (uniform):
  - Pick possible inputs uniformly
  - Avoids designer bias
    - A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs as equally valuable
- Systematic (non-uniform):
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail *often* or *not at all*
- *Functional testing is systematic testing*

Why not random?

- Compute the probability of selecting a test case that reveals the fault in line 19 by randomly sampling the input domain, assuming that type double has range $-2^{31} \ldots 2^{31}-1$.
- Compute the probability of randomly selecting a test case that reveals a fault if lines 13 and 19 were both missing the condition a != 0.

```
1   /** Find the two roots of ax^2 + bx + c,
2    * that is, the values of x for which the result is 0.
3    */
4   class Roots {
5       double root_one, root_two;
6       int num_roots;
7       public roots(double a, double b, double c) {
8           double q;
9           double r;
10          // Apply the textbook quadratic formula:
11          // Roots = -b +- sqrt(b^2 - 4ac) / 2a
12          q = b*b - 4*a*c;
13          if (q > 0 && a != 0) {
14              // If b^2 > 4ac, there are two distinct roots
15              num_roots = 2;
16              r = (double) Math.sqrt(q) ;
17              root_one =   ((0-b) + r)/(2*a);
18              root_two =   ((0-b) - r)/(2*a);
19          } else if (q==0) { // (BUG HERE)
20              // The equation has exactly one root
21              num_roots = 1;
22              root_one = (0-b)/(2*a);
23              root_two = root_one;
24          } else {
25              // The equation has no roots if b^2 < 4ac
26              num_roots = 0;
27              root_one = -1;
28              root_two = -1;
29          }
30      }
31      public int num_roots() { return num_roots; }
32      public double first_root()   { return root_one; }
33      public double second_root() { return root_two; }
34  }
```

Why Not Random?

- Non-uniform distribution of faults
- *Example:* Java class "roots" applies quadratic equation

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$

Failing values are *sparse* in the input space — needles in a very big haystack. Random sampling is unlikely to choose a=0.0 and b=0.0

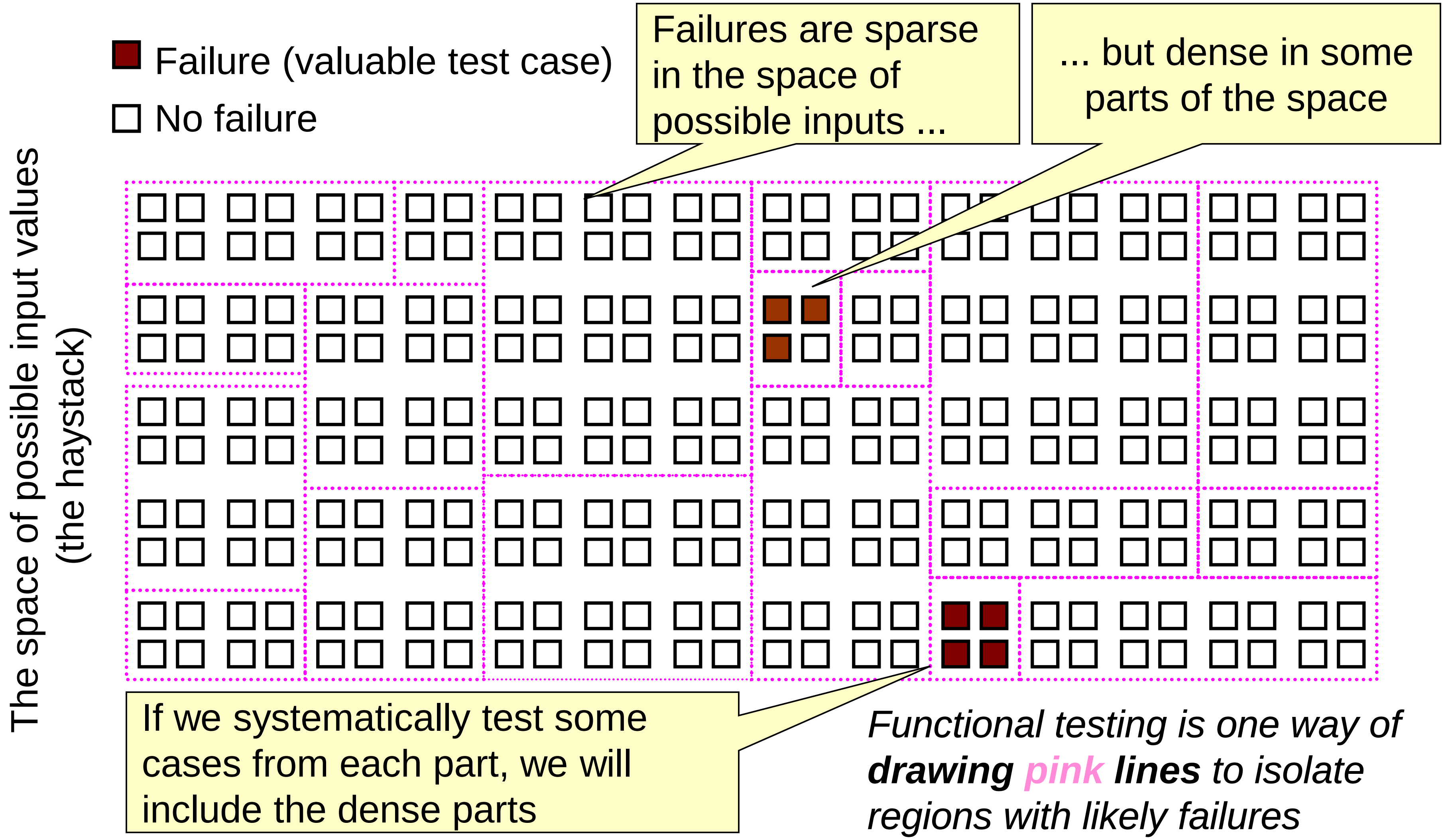
Consider the purpose of testing ...

- To estimate the proportion of needles to hay, sample randomly
  - Reliability estimation requires unbiased samples for valid statistics. *But that's not our goal!*
- To find needles and remove them from hay, look systematically (non-uniformly) for needles
  - Unless there are a *lot* of needles in the haystack, a random sample will not be effective at finding them
  - We need to use everything we know about needles, e.g., are they heavier than hay? Do they sift to the bottom?

Systematic Partition Testing

- Partition testing separates the **input space** into **classes** whose union is the entire space.
- Divides the infinite set of possible test cases into a finite set of classes, with the purpose of drawing one or more test cases from each class.
- Usually produces fewer test cases than random testing

Systematic Partition Testing


🟥 Failure (valuable test case)
☐ No failure
Failures are sparse in the space of possible inputs ...
... but dense in some parts of the space
The space of possible input values (the haystack)
If we systematically test some cases from each part, we will include the dense parts

Functional testing is one way of ***drawing pink lines*** *to isolate regions with likely failures*

The partition principle

- Exploit some knowledge to choose samples that are more likely to include “special” or trouble-prone regions of the input space
  - Failures are sparse in the whole input space …
  - … but we may find regions in which they are dense
- (Quasi*-)Partition testing: separates the input space into classes whose union is the entire space
  - *Quasi because: The classes may overlap
- Desirable case: Each fault leads to failures that are dense (easy to find) in some class of inputs
  - sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault
  - seldom guaranteed; we depend on experience-based heuristics

Functional testing: exploiting the specification

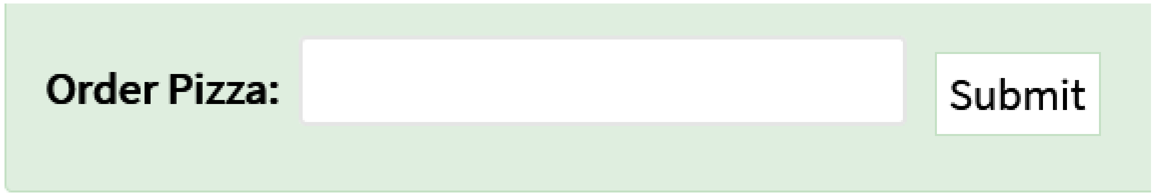
- Functional testing *uses the* ***specification*** (formal or informal) to **partition the input space**
  - E.g., for a quadratic equation, division between cases with zero, one, and two real roots
- Test each category, and *boundaries* between categories
  - No guarantees, but experience suggests failures often lie at the boundaries (as in the “roots” program)

Equivalence Partitioning

- A testing method that divides the input domain of a program into sets of data from which test cases can be derived.
- Equivalence partitioning strives to define a test case that uncovers a class of errors, thereby reducing the total number of test cases that must be developed.

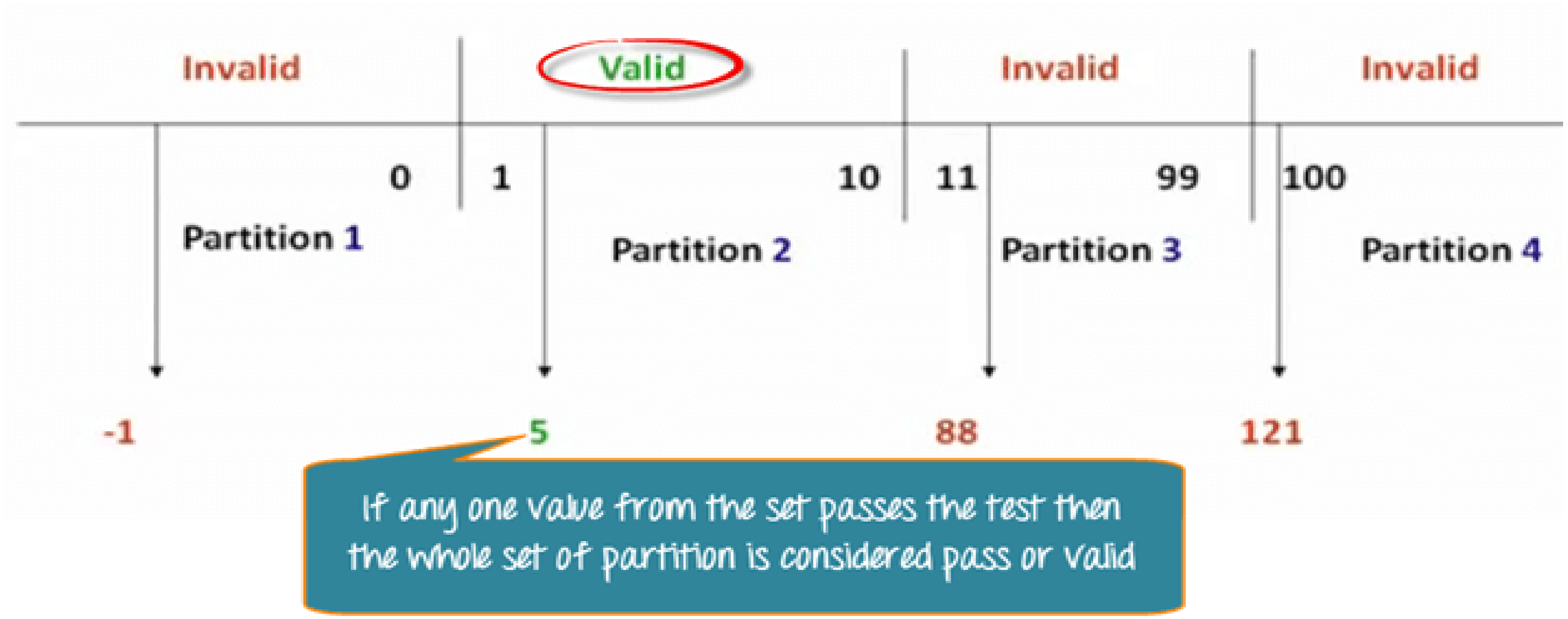
Equivalence Partitioning

- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, "Only 10 Pizza can be ordered"



Source:https://www.guru99.com/equivalence-partitioning-boundary-value-analysis.html

Equivalence Partitioning

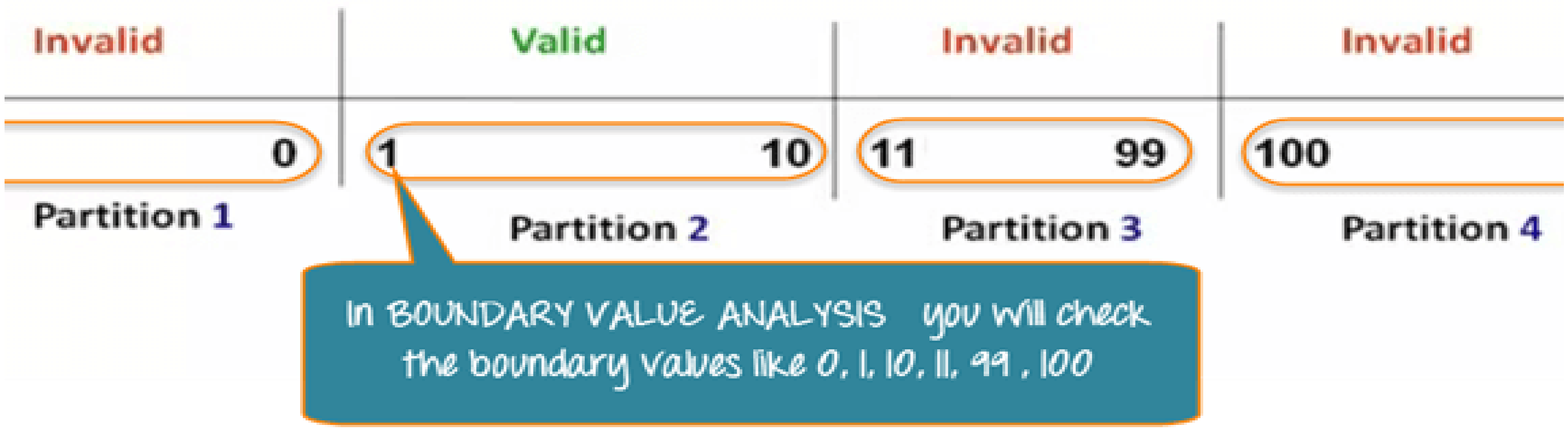

Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
-1
5
88
121
If any one value from the set passes the test then the whole set of partition is considered pass or valid

Boundary Value Analysis

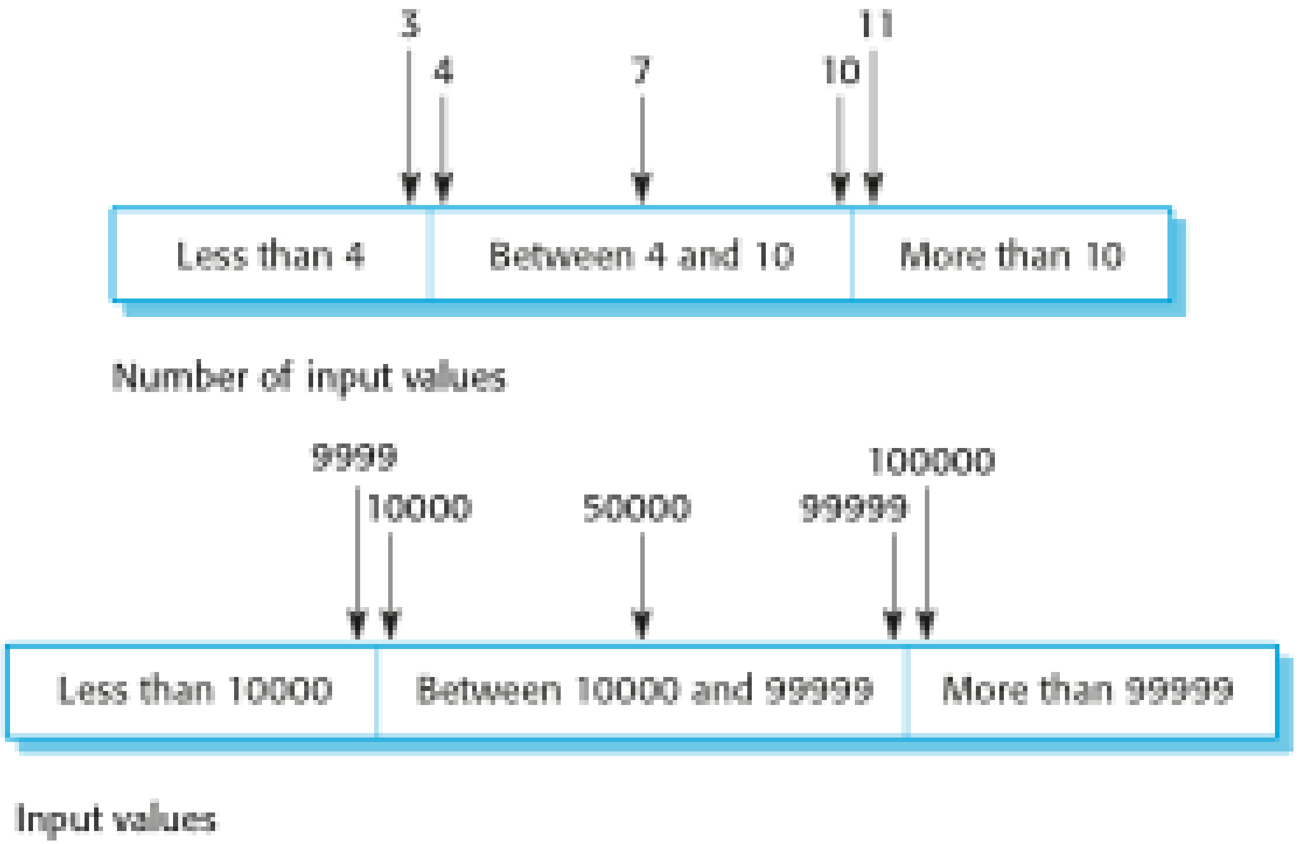
A test case design technique that complements equivalence partitioning.

- Rather than selecting any element of an equivalence set, it selects test cases at the edges of the set.
- Boundary value analysis leads to a selection of test cases that exercise boundary values.

Boundary Value Analysis


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99 , 100

Equivalence partitions



Why functional testing?

- The baseline technique for designing test cases
  - Timely
    - Often useful in refining specifications and assessing testability *before* code is written
  - Effective
    - finds some classes of fault (e.g., missing logic) that can elude other approaches
  - Widely applicable
    - to any description of program behavior serving as spec
    - at any level of granularity from module to system testing.
  - Economical
    - typically less expensive to design and execute than structural (code-based) test cases

Early functional test design

- Program code is not necessary
  - Only a description of intended behavior is needed
  - Even incomplete and informal specifications can be used
    - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
  - Often reveals ambiguities and inconsistency in spec
  - Useful for assessing testability
    - And improving test schedule and budget by improving spec
  - Useful explanation of specification
    - or in the extreme case (as in XP), test cases are the spec

Functional versus Structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults
- Functional testing is best for ***missing logic*** faults
  - A common problem: Some program logic was simply forgotten
  - Structural (code-based) testing will *never focus on code that isn't there*!

Functional vs structural test: granularity levels

- Functional test applies at all granularity levels:

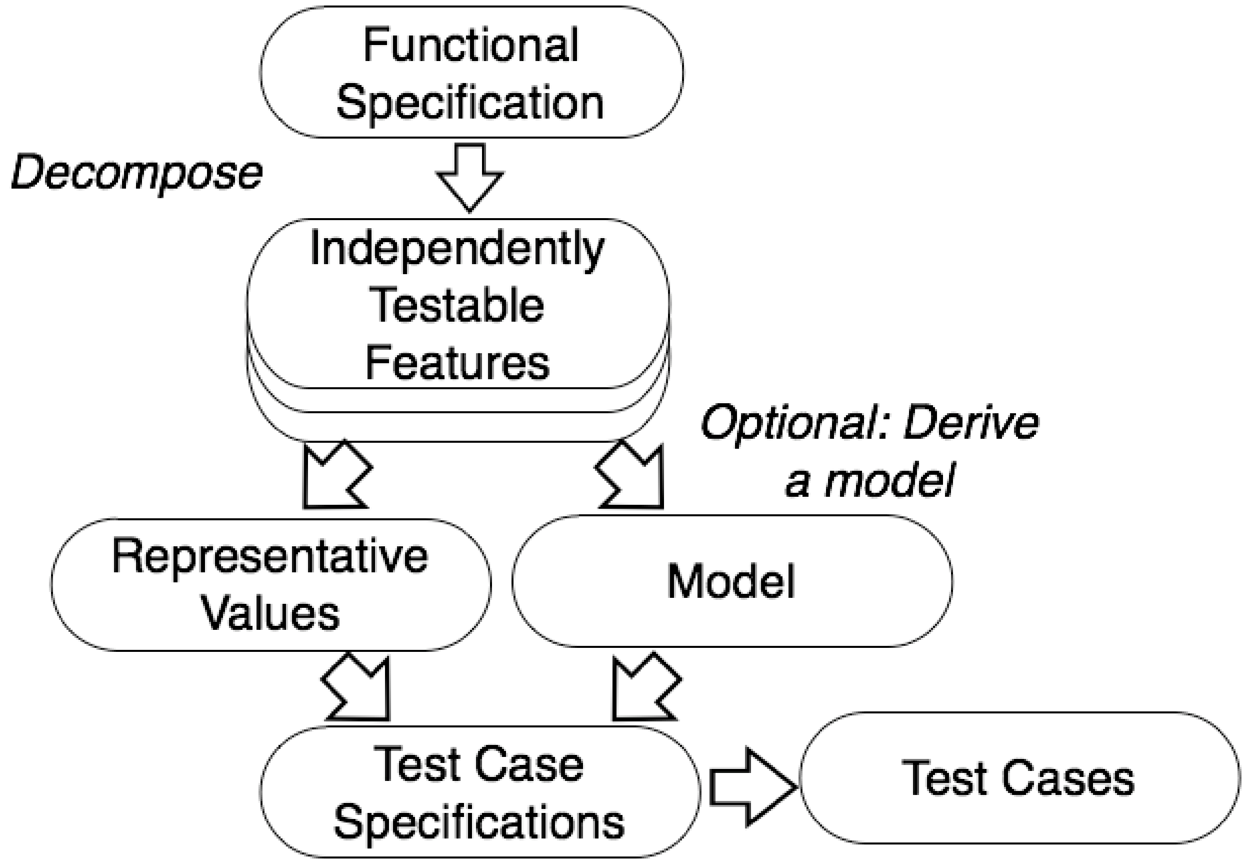
– Unit	(from module interface spec)
– Integration	(from API or subsystem spec)
– System	(from system requirements spec)
– Regression	(from system requirements + bug history)

- Structural (code-based) test design applies to relatively small parts of a system:
  - Unit
  - Integration

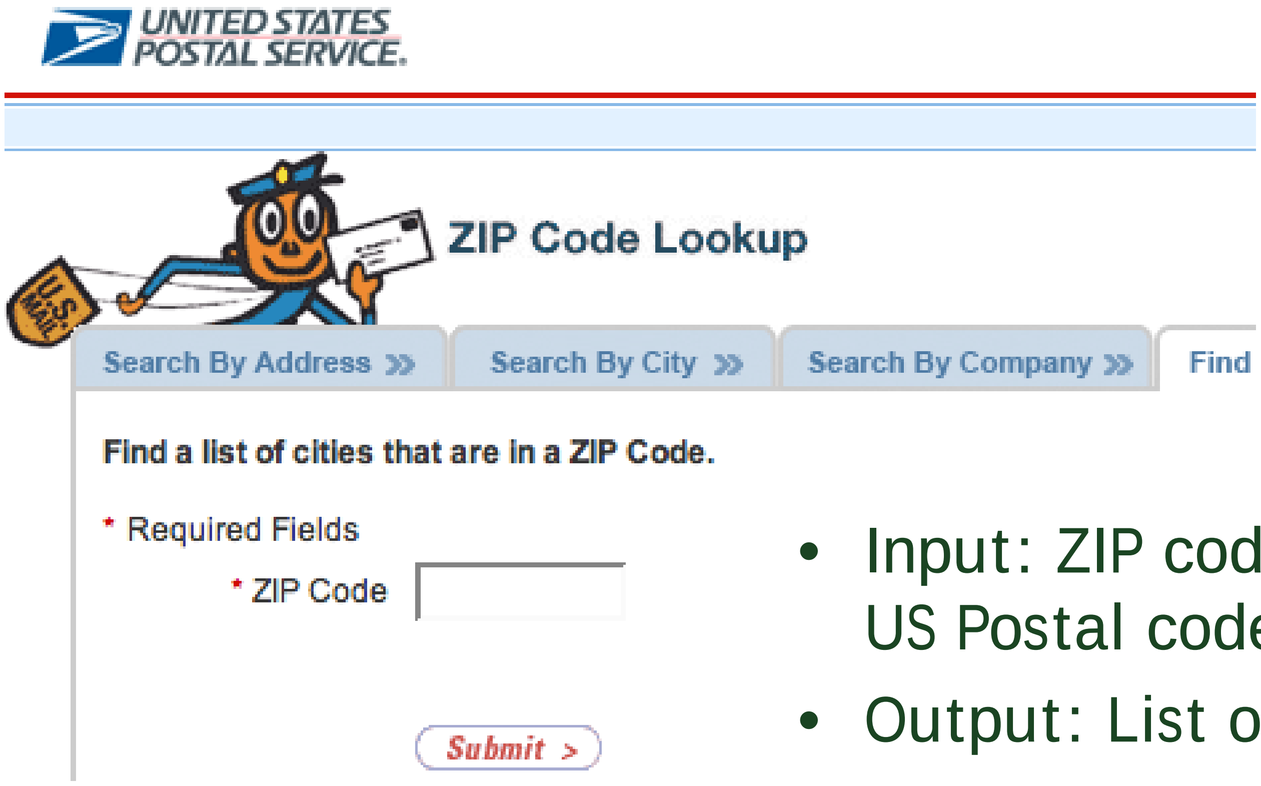
Steps: From specification to test cases

- 1. Decompose the specification
  - If the specification is large, break it into *independently testable features* to be considered in testing
- 2. Select representatives
  - Representative values of each input, or
  - Representative behaviors of a *model*
    - Often simple input/output tuples don’t describe a system. We use models in program specification, in program design, and in test design
- 3. Form test specifications
    - Typically: combinations of input values, or model behaviors
- 4. Produce and execute actual tests

From specification to test cases



Simple example: Postal code lookup


UNITED STATES POSTAL SERVICE

ZIP Code Lookup

Search By Address »

Search By City »

Search By Company »

Find

Find a list of cities that are in a ZIP Code.

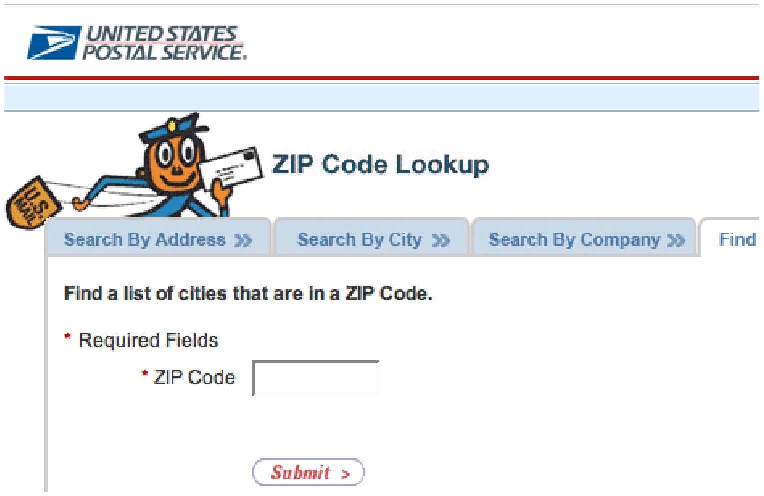
* Required Fields

* ZIP Code

- Input: ZIP code (5-digit US Postal code)
- Output: List of cities
- What are some representative values (or classes of value) to test?

Example: Representative values

Simple example with one input, one output


UNITED STATES POSTAL SERVICE.

ZIP Code Lookup

Find a list of cities that are in a ZIP Code.

* Required Fields

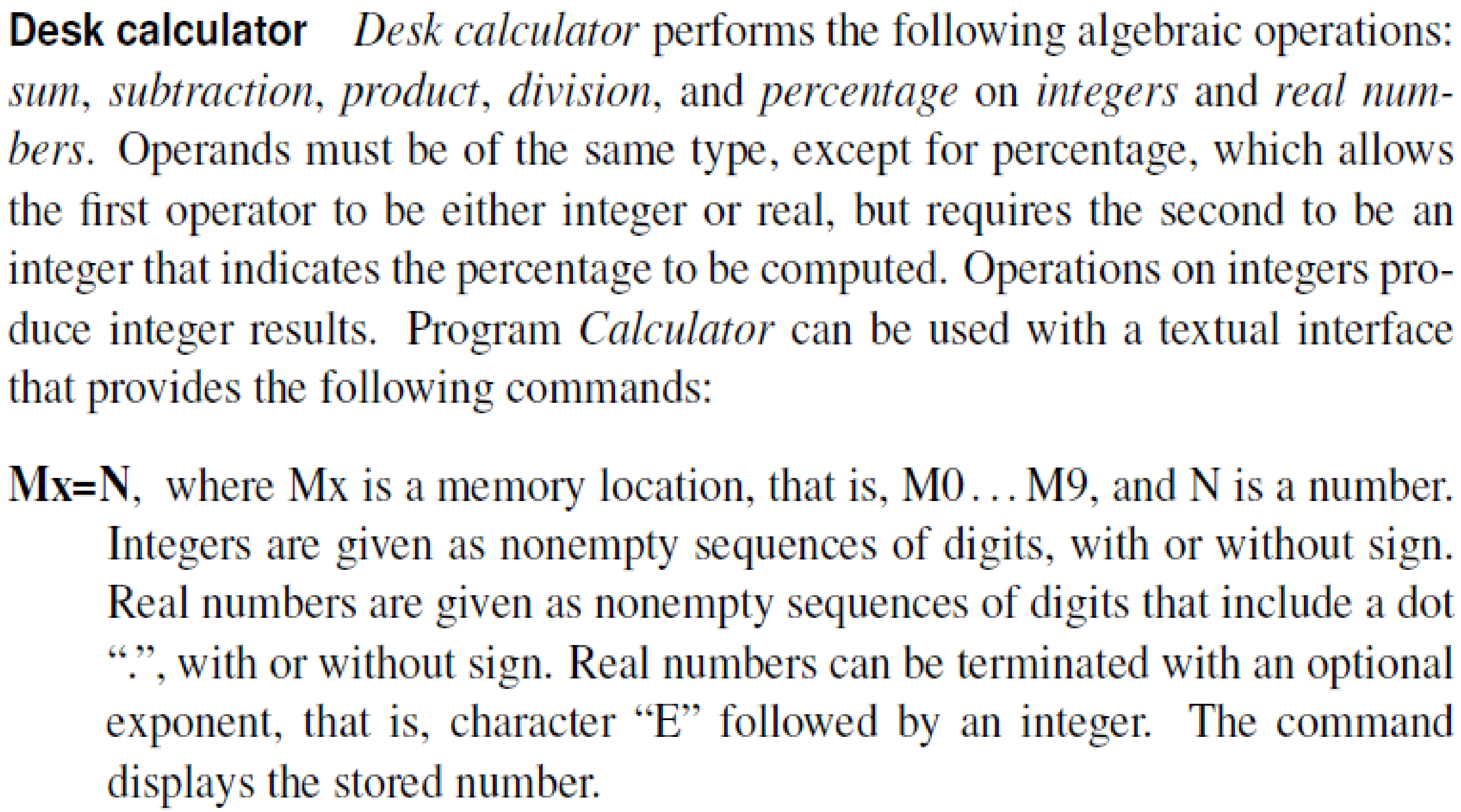
* ZIP Code

- Correct zip code
  - With 0, 1, or many cities
- Malformed zip code
  - Empty; 1-5 characters; 6 characters; very long
  - Non-digit characters
  - Non-character data

Note prevalence of boundary values (0 cities, 6 characters) and error cases

Pen and paper exercise

- Identify independently testable units in the following specification.



Desk calculator *Desk calculator* performs the following algebraic operations: *sum*, *subtraction*, *product*, *division*, and *percentage* on *integers* and *real numbers*. Operands must be of the same type, except for percentage, which allows the first operator to be either integer or real, but requires the second to be an integer that indicates the percentage to be computed. Operations on integers produce integer results. Program *Calculator* can be used with a textual interface that provides the following commands:

Mx=N, where Mx is a memory location, that is, M0...M9, and N is a number. Integers are given as nonempty sequences of digits, with or without sign. Real numbers are given as nonempty sequences of digits that include a dot ".", with or without sign. Real numbers can be terminated with an optional exponent, that is, character "E" followed by an integer. The command displays the stored number.

Mx=display, where Mx is a memory location and *display* indicates the value shown on the last line.

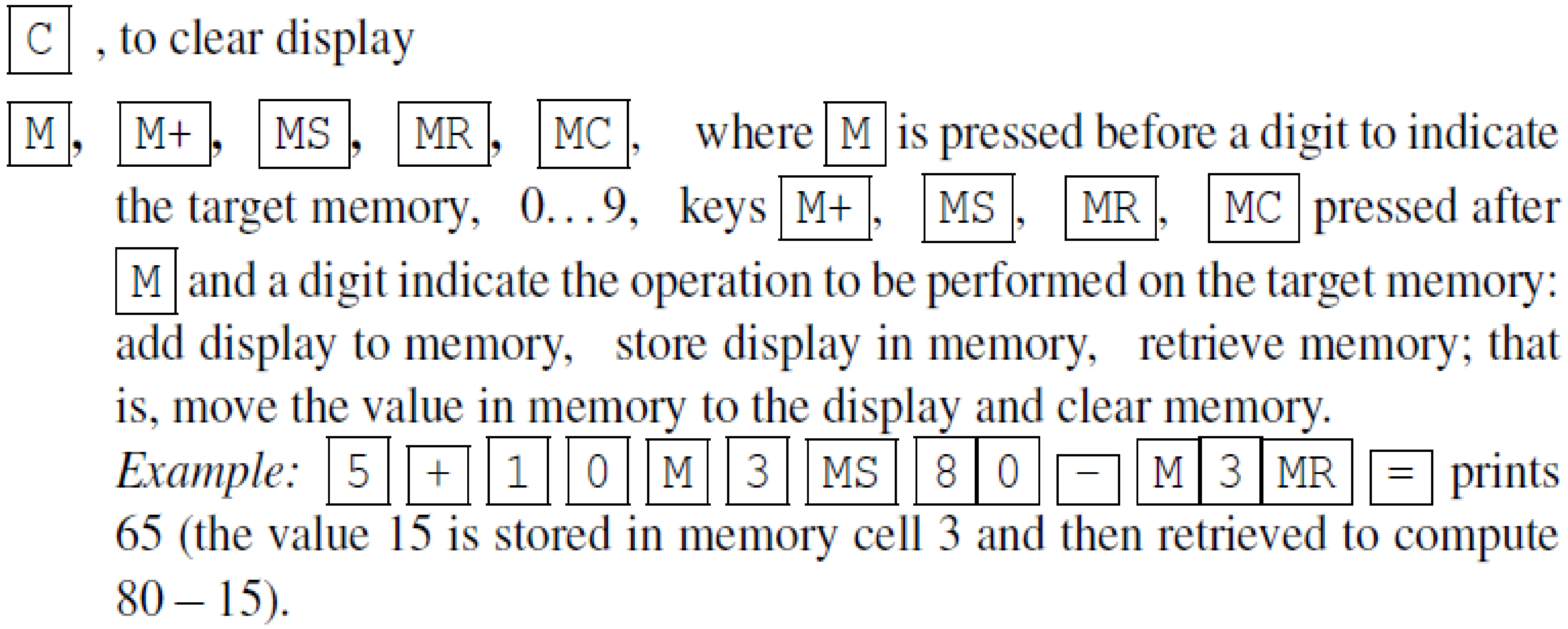
operand1 operation operand2, where *operand1* and *operand2* are numbers or memory locations or *display* and *operation* is one of the following symbols: "+", "-", "*", "/", "%", where each symbol indicates a particular operation. Operands must follow the type conventions. The command displays the result or the string *Error*.

or with a graphical interface that provides a display with 12 characters and the following keys:

$\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$, $\boxed{8}$, $\boxed{9}$, the 10 digits

$\boxed{+}$, $\boxed{-}$, $\boxed{*}$, $\boxed{/}$, $\boxed{\%}$, the operations

$\boxed{=}$ to display the result of a sequence of operations



$\boxed{\mathtt{C}}$, to clear display

$\boxed{\mathtt{M}}$, $\boxed{\mathtt{M+}}$, $\boxed{\mathtt{MS}}$, $\boxed{\mathtt{MR}}$, $\boxed{\mathtt{MC}}$, where $\boxed{\mathtt{M}}$ is pressed before a digit to indicate the target memory, 0…9, keys $\boxed{\mathtt{M+}}$, $\boxed{\mathtt{MS}}$, $\boxed{\mathtt{MR}}$, $\boxed{\mathtt{MC}}$ pressed after $\boxed{\mathtt{M}}$ and a digit indicate the operation to be performed on the target memory: add display to memory, store display in memory, retrieve memory; that is, move the value in memory to the display and clear memory. *Example:* $\boxed{\mathtt{5}}$ $\boxed{\mathtt{+}}$ $\boxed{\mathtt{1}}$ $\boxed{\mathtt{0}}$ $\boxed{\mathtt{M}}$ $\boxed{\mathtt{3}}$ $\boxed{\mathtt{MS}}$ $\boxed{\mathtt{8}}\boxed{\mathtt{0}}$ $\boxed{\mathtt{-}}$ $\boxed{\mathtt{M}}\boxed{\mathtt{3}}\boxed{\mathtt{MR}}$ $\boxed{\mathtt{=}}$ prints 65 (the value 15 is stored in memory cell 3 and then retrieved to compute 80 – 15).

Random Testing

- Test a program by generating random, independent inputs.
- Using the specification to derive expected output and pass/fail criteria
- In the absence of specifications, exceptions thrown during test execution is an indication of faults.

Random Testing

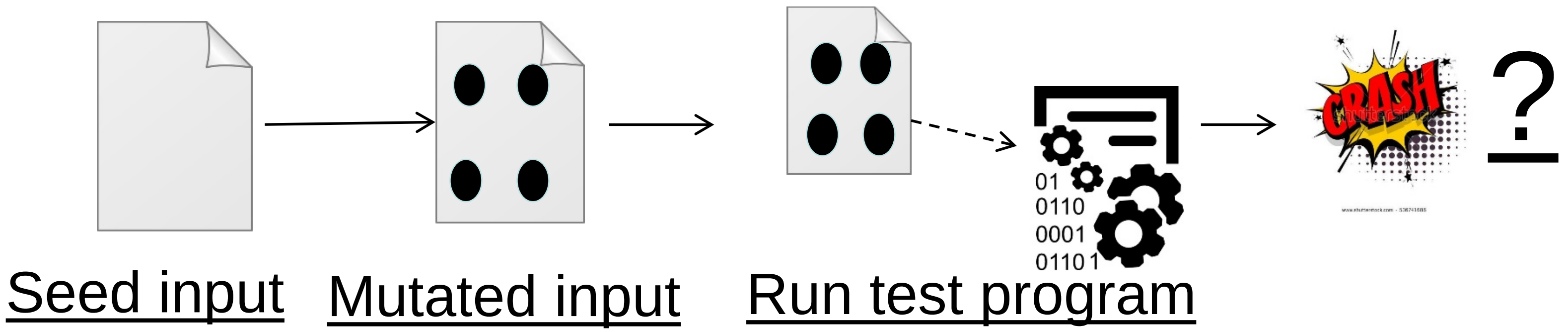
- Advantages:
  - Cheap to use
  - No bias (uniform random test)
  - Sometimes quick to find bug candidates
- Disadvantages:
  - Only finds basic bugs
  - Perform poorly with respect to other techniques to find bugs
  - Low coverage

Fuzzing

- Fuzzing is a random testing technique
  - Automatically generate test cases
  - Many slightly anomalous test cases are input into a target
- Generate invalid, unexpected, or random data as inputs to a computer program, and then observe for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.
- Different types of fuzzing:
  - Generation-based vs. Mutation-based
  - Dumb vs. smart
  - White-box vs. grey-box vs. black-box.

Mutation-based Fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: Mutation-based Fuzzing

Fuzzing a PDF viewer:

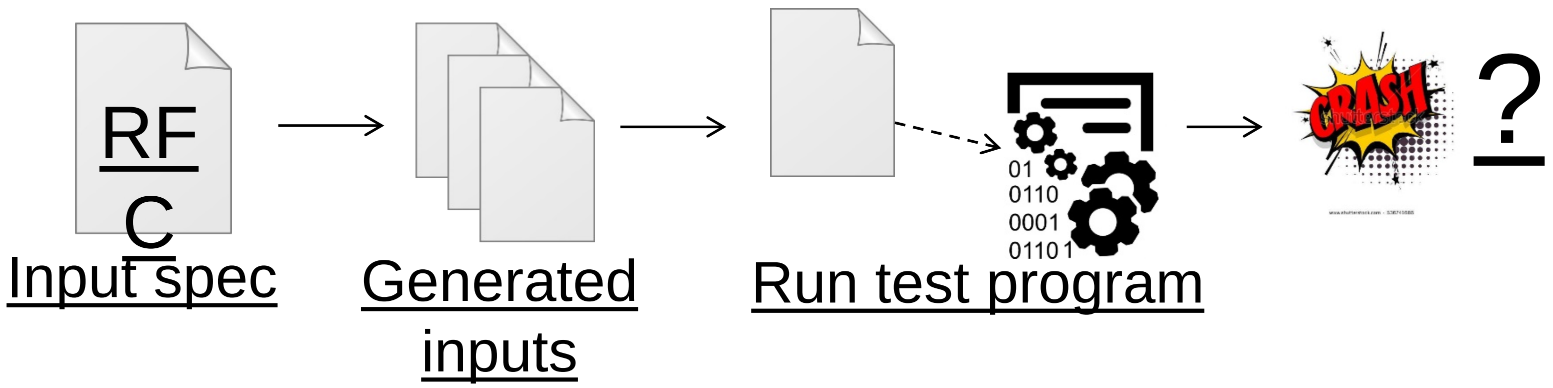
1. Google for .pdf (about 1 billion results)
2. Crawl pages to build a corpus
3. Use fuzzing tool (or script)
   - Collect seed PDF files
   - Mutate that file
   - Feed it to the program
   - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

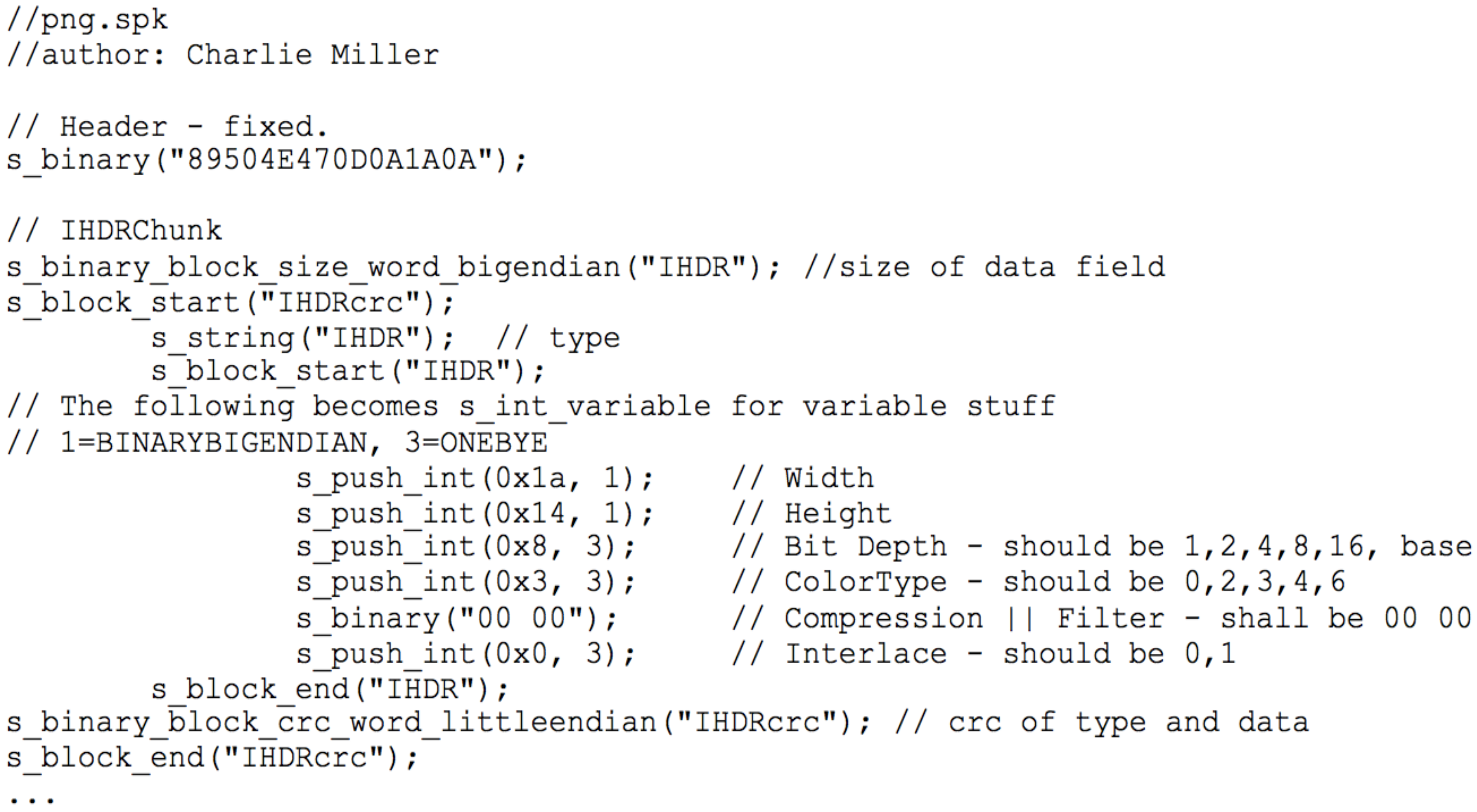
- Easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Generation-Based Fuzzing

- Test cases are generated from some description of the input format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Generation-Based Fuzzing



```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, base
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Sample PNG spec

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependencies (e.g, checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they’re all run and no bugs are found?

Code coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer