Functional Testing

SWE 261P

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 Inputs

- 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" usually implies **systematic** testing

avoid developer's bias

find test case which
are easy to fail

Why Not Random?

- Non-uniform distribution of faults
- *Example:* Java class "Roots" implements the quadratic equation

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

Why Not Random?

- Non-uniform distribution of faults
- *Example:* Java class "Roots" implements the 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 $a=0$

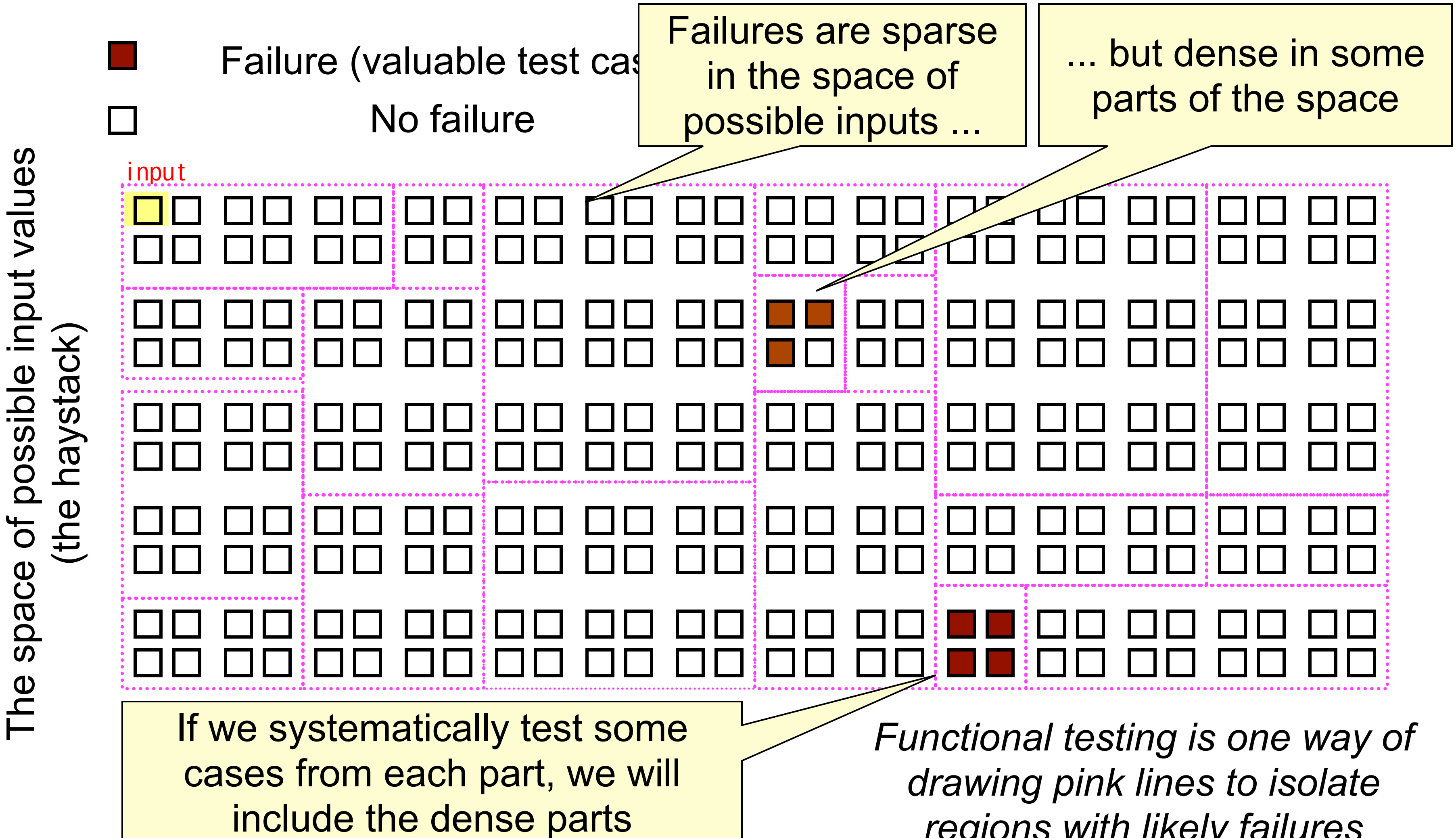
try to make sure the system is correct

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 ...

- If our purpose was to estimate the proportion of needles to hay, sample randomly
  - Reliability estimation requires unbiased samples for valid statistics. *But, generally, 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



Functional testing is one way of drawing pink lines to isolate regions with likely failures no overlap in a group

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 — "Equivalence Partition"
      - » *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., specification of "square root" program suggests division between cases positive, imaginary
- Test each category, and boundaries between categories
  - No guarantees, but experience suggests failures often lie at the boundaries

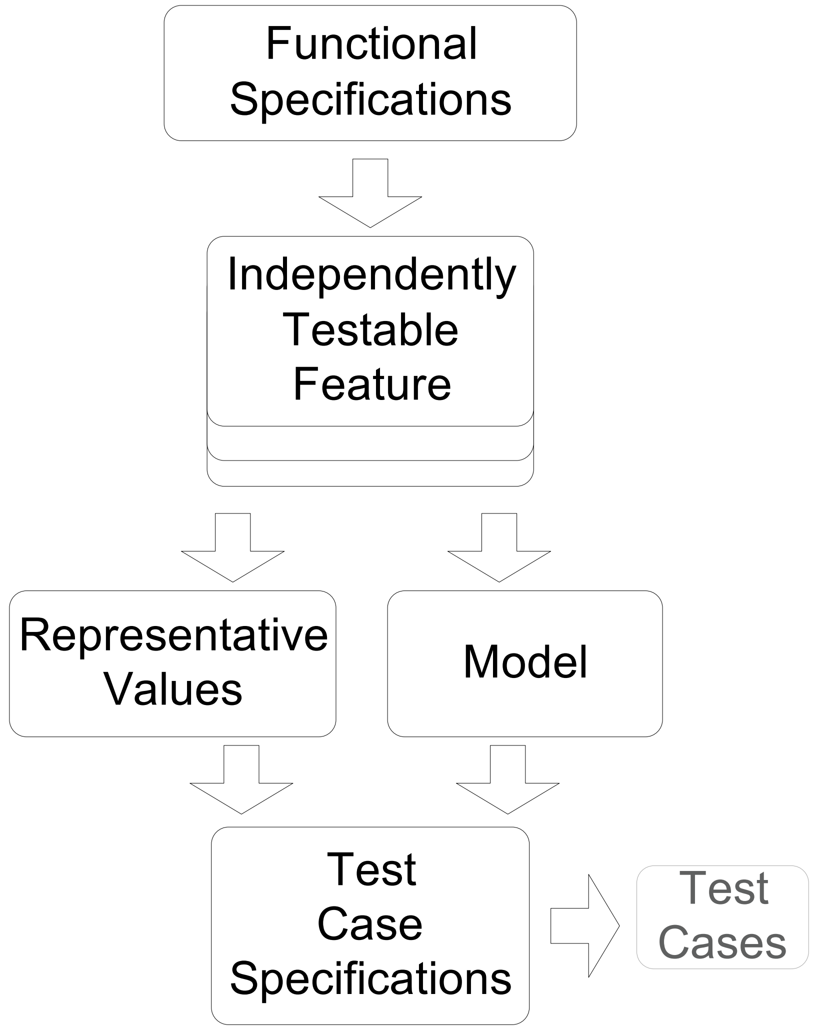
Functional versus Structural: Classes of faults

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

Steps: From specification to test cases

- 1. Decompose the specification into equivalence partitions
    - If the specification is large, break it into *independently testable features* to be considered in testing
- 2. Select representatives
    - Representative values (including boundary values) of each input, or
    - Representative behaviors of a *model*
        - Often simple input/output transformations 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



sometimes it doesn't make sense

Boundary Values

- Each input can be treated independently. For example, in the quadratic equation, "a," "b," and "c" should each be explored for each of their own possible boundary values
- Partitions for value ranges should be specified both in terms of the value range and the inclusivity and/or exclusivity of the extremes
  - For example, "Partition 1: $0 \leq x < 20$"
  - Or, another example, "Partition 1 for x is [0, 20)"
    - Brackets denote inclusion, Parentheses denote exclusion
- Boundary values expressed for a partition should account for the valid values that are ***within*** the partition (which must account for the data type of the variable)

Brainstorming Exercise

Simple example with one input, one output

Think of and suggest equivalence partitions
(that we could then test for)