Functional testing

Learning objectives

- Understand the rationale for systematic (non-random) selection of test cases
  - Understand the basic concept of partition testing and its underlying assumptions
- Understand why functional test selection is a primary, base-line technique
  - Why we expect a specification-based partition to help select valuable test cases
- Distinguish functional testing from other systematic testing techniques

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?

- Non-uniform distribution of faults
- *Example:* Java class "roots" applies quadratic equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

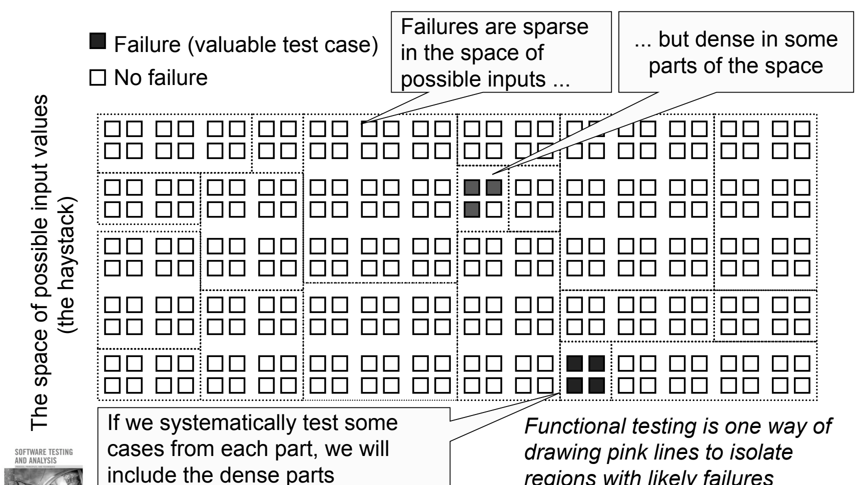
Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a=0$

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



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., specification of "roots" program suggests 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)

Why functional testing?

- The base-line 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!

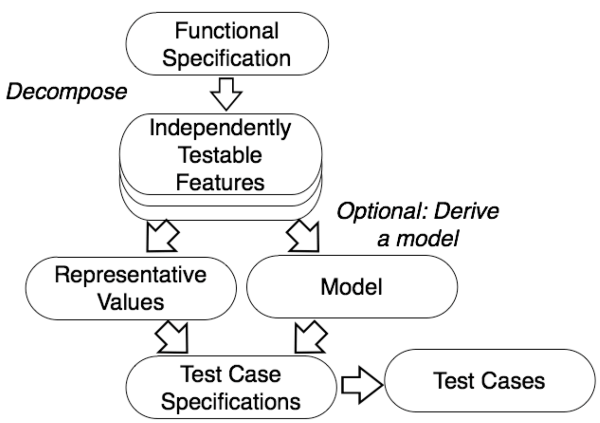
Functional vs structural test: granularity levels

- 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 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



Simple example: Postal code lookup

- 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

- Correct zip code
  - With 0, 1, or many cities
- Malformed zip code
  - Empty; 1-4 characters; 6 characters; very long
  - Non-digit characters
  - Non-character data

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

Summary

- Functional testing, i.e., generation of test cases from specifications is a valuable and flexible approach to software testing
  - Applicable from very early system specs right through module specifications
- (quasi-)Partition testing suggests dividing the input space into (quasi-)equivalent classes
  - Systematic testing is intentionally non-uniform to address special cases, error conditions, and other small places
  - Dividing a big haystack into small, hopefully uniform piles where the needles might be concentrated