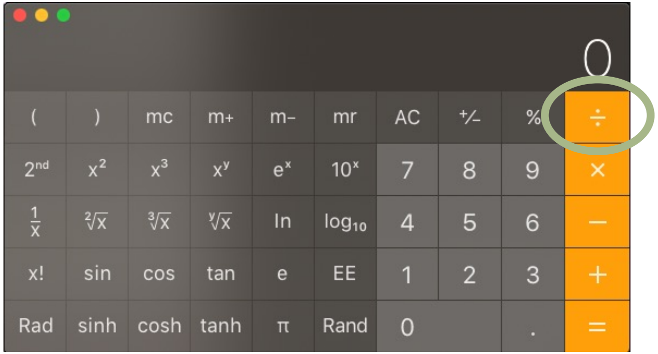
Behavior-First: Selecting Test Inputs based on Specs

How to create test cases that **systematically** cover the **input space** of an entity under test **without looking at implementation?**

Calculator: Integer Division

- Divide requires two integers, A and B and computes an integer A/C
  - Integer C := Divide(integer: A, integer: B)
- What values would you select to test this?



Possible values & observations

- Divide(4,2)
- Divide(10,5)
- Divide(10, 3)
- Divide(100, 44)
- Divide(-4,-2)
- Divide(4,0)
- Divide(“Hi”, ”There”)

Possible tests & observations

- Divide(4, 2)
- Divide(10, 5)
- Divide(10, 3)
- Divide(100, 44)
- Divide(-4, -2)
- Divide(4, 0)
- Divide(“Hi”, “There”)

Is there a meaningful difference between these two test cases?

Possible tests & observations

- Divide(4, 2)
- Divide(10, 5)
- Divide(10, 3)
- Divide(100, 44)
- Divide(-4, -2)
- Divide(4, 0)
- Divide(“Hi”, “There”)

Do we need these? Any difference between the two cases?

Why is this needed?

Possible tests & observations

- Divide(4, 2)
- Divide(10, 5)
- Divide(10, 3)
- Divide(100, 44)
- Divide(-4, -2)
- Divide(4, 0)
- Divide(“Hi”, ”There”)

Why is this needed?

What does this even mean?

Possible tests & observations

- Divide(4, 2)
- Divide(10, 5)
- Divide(10, 3)
- Divide(100, 44)
- Divide(-4, -2)
- Divide(4, 0)
- Divide(“Hi”, ”There”)

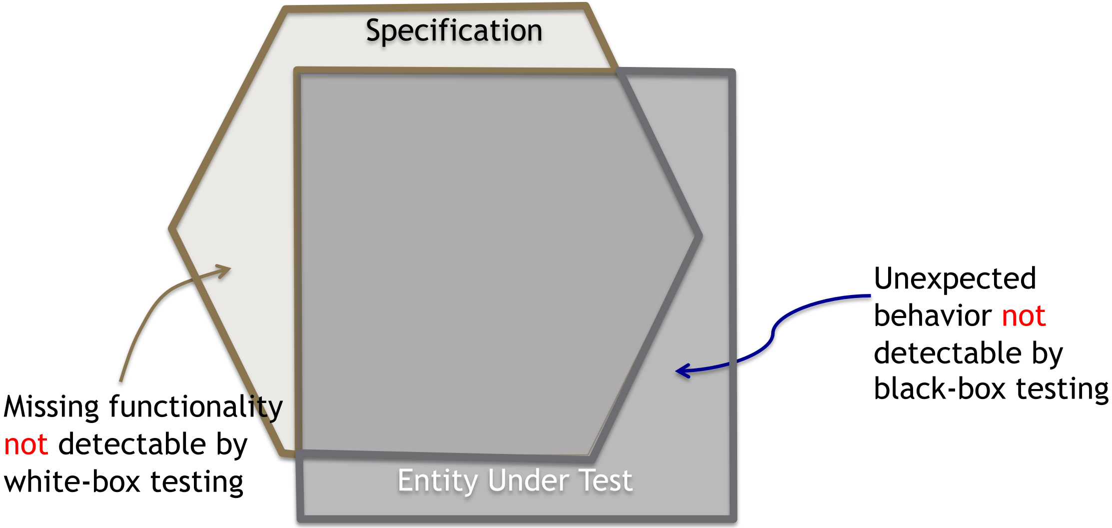
Intuitively know why these make sense or don’t, but we want to formulate this intuition as specific test design strategies

Recall the Principles

- Behavior-first: 100% code coverage is not the goal
- Tests must be comprehensive
  - Happy paths
  - Sad paths
  - *Corner cases* (unspecified exceptional or rare unusual cases)
    - *Boundary cases* (inputs at or near their valid ranges)
- Tests should not be redundant, or overprotective

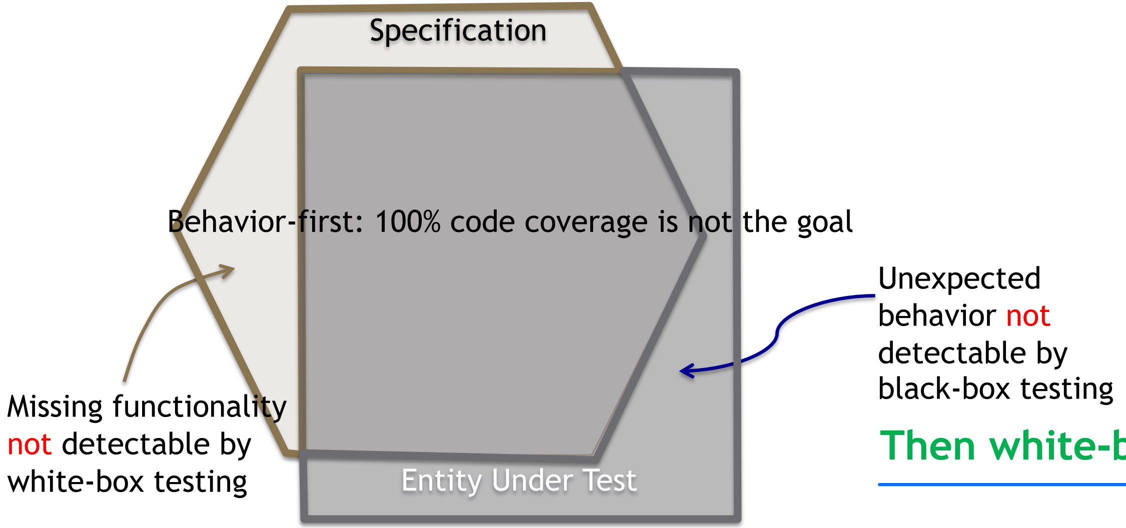
How to achieve these with more rigor?

Remember: Different testing strategies focus on different faults



Different testing strategies focus on different faults

What is another term for black-box testing?



Then white-box!

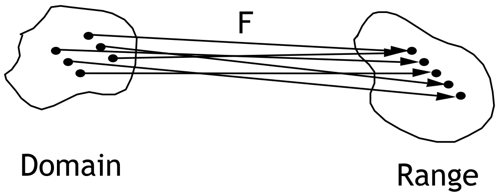
Interested in black-box first, or spec-based testing!

So spec-based testing is also known by other names: let's get this straight…

Also called:

- *black-box testing / behavior-first testing*
- *functional testing* ⚠
  - *because any program can be viewed as a function from a set of inputs to a set of outputs*

input:
- not necessarily a single value
- possibly sequence of actions



output:
- not necessarily a single value
- can be a side effect

Specification = description of intended behavior of EUT

– either *formal* or *informal*

Recall the Principles

- Behavior-first: 100% code coverage is not the goal
- Tests must be comprehensive
  - Happy paths
  - Sad paths
  - *Corner cases* (unspecified exceptional or rare unusual cases)
    - *Boundary cases* (inputs at or near their valid ranges)
- Tests should not be redundant, or overprotective

How to achieve these with more rigor?

Goals

- Distinguish spec-based testing from other systematic testing techniques
  - Understand the rationale for systematic vs. random or ad-hoc selection of test cases
- Use spec-based test selection as a primary, base-line technique
  - Apply the principle of **behavior-first** ("focus on behavior, not implementation")
  - Work with the basic concepts of combinatorial testing

In spec-based testing, we can pick test values in different ways

Systematic Selection

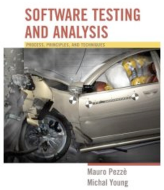
- Select inputs that are valuable and likely to reveal faults based on known strategies
  - Focus on inputs that have the highest potential to cause faults
  - Choose representative values of input classes that are likely to fail *together* or *not at all*
  - *Problem:* some faults (and programmer behavior) may not obey the general rules on which these strategies rely

Random Selection (Fuzzing)

- Pick possible inputs uniformly *or* perturb existing inputs in random ways
  - Assumption: All inputs as equally valuable
  - Removes bias
  - *Problem*: Faults are not uniformly distributed
    - What if failing values are very rare in the input space

Ad-Hoc Selection

- Try to select test case values based on knowledge, expertise and experience, guessing where faults could be
  - *Problem*: novice testers/devs will do a poor job
  - *Problem*: ester can make the same logical mistakes and bad assumptions as the programmer
    - Especially if they are the same person



Adapted from Software Testing and Analysis. Mauro Pezzè & Michal Young. 2007, *with corrections*

Spec-based testing applied systematically focuses on bug-prone parts of a system

Random (aka Fuzzing)

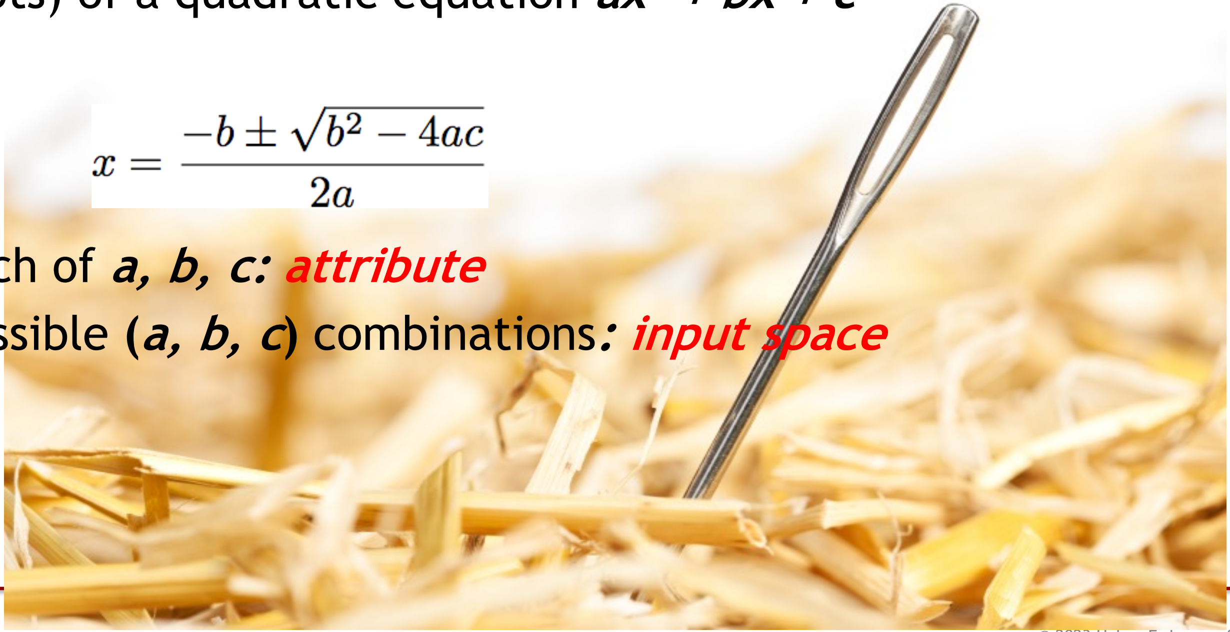
Ad-hoc (non-uniform)

Systematic (non-uniform)

- try to select inputs that are especially valuable/interesting based on known strategies
- usually by choosing a lot of representatives from behaviors that are likely to be fault-prone along with a few representatives from behaviors that are unlikely to be fault-prone

Why not random?
Because faults are not uniformly distributed

- *Example*: Java program that calculates the *real* roots (0, 1, or 2 roots) of a quadratic equation ***ax² + bx + c***



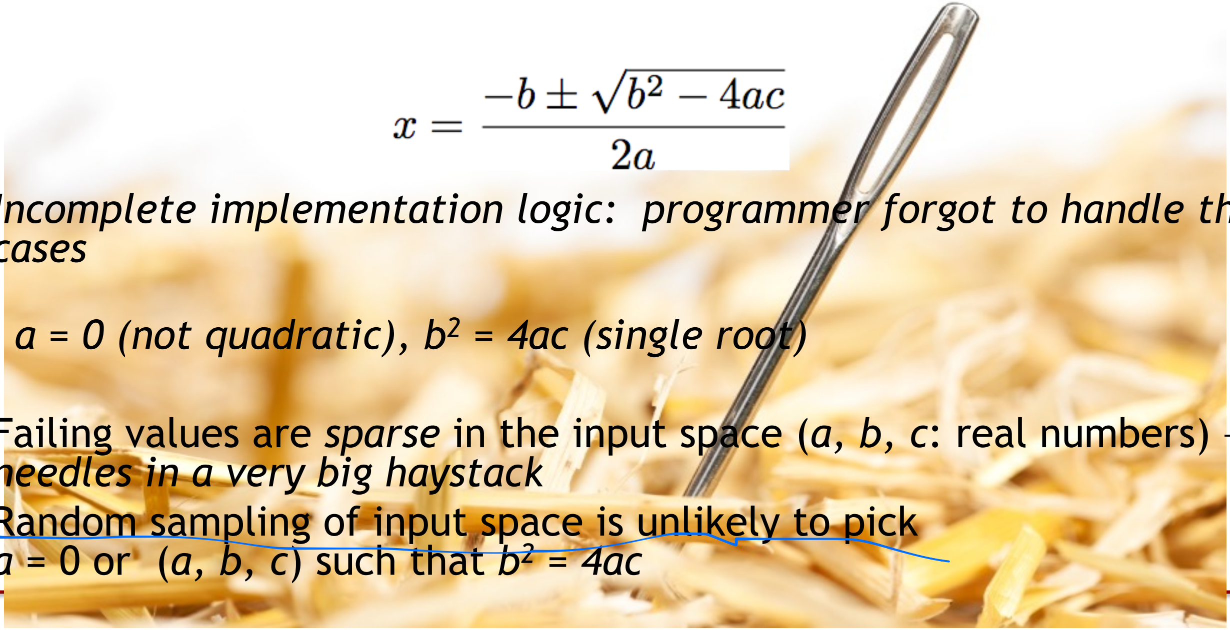
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Each of ***a, b, c***: ***attribute***
- Possible (***a, b, c***) combinations: ***input space***

Image: http://econscribi.com/blog/wp-content/uploads/2015/04/Needle-in-Haystack.jpg

Why not random?
Because faults are not uniformly distributed

- *Example*: Java program that calculates the *real* roots (0, 1, or 2 roots) of a quadratic equation $\boldsymbol{ax^2 + bx + c}$



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

Incomplete implementation logic: programmer forgot to handle the cases

$a = 0$ (not quadratic), $b^2 = 4ac$ (single root)

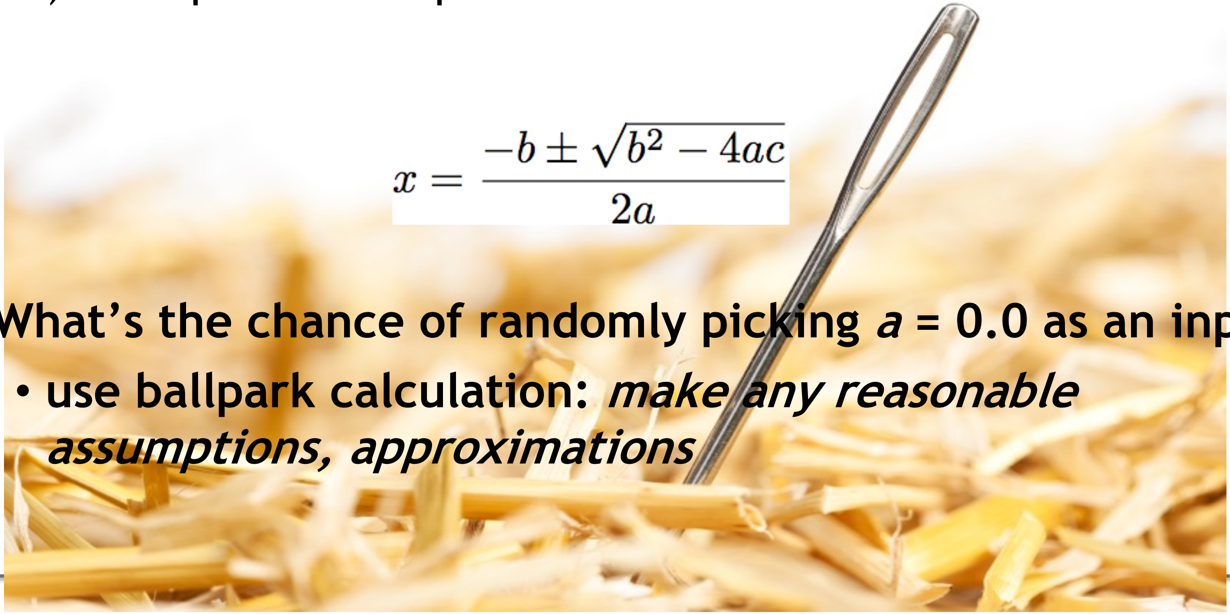
  - Failing values are *sparse* in the input space (a, b, c: real numbers) – *needles in a very big haystack*
  - Random sampling of input space is unlikely to pick $a = 0$ or (a, b, c) such that $b^2 = 4ac$

Image: http://econscribi.com/blog/wp-content/uploads/2015/04/Needle-in-Haystack.jpg

Why not random?

Mentimeter ?

- *Example:* Java program that calculates the *real* roots (0, 1, or 2 roots) of a quadratic equation $ax^2 + bx + c$



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

  – **What's the chance of randomly picking *a* = 0.0 as an input?**
    - **use ballpark calculation: *make any reasonable assumptions, approximations***

Image: http://econscribi.com/blog/wp-content/uploads/2015/04/Needle-in-Haystack.jpg

Why not random?

- *Example*: Java program that calculates the *real* roots (0, 1, or 2 roots) of a quadratic equation $ax^2 + bx + c$

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

Assume: 32-bit floating point number (all bit patterns are valid)

What's the chance of picking $a = 0.0$ as an input?

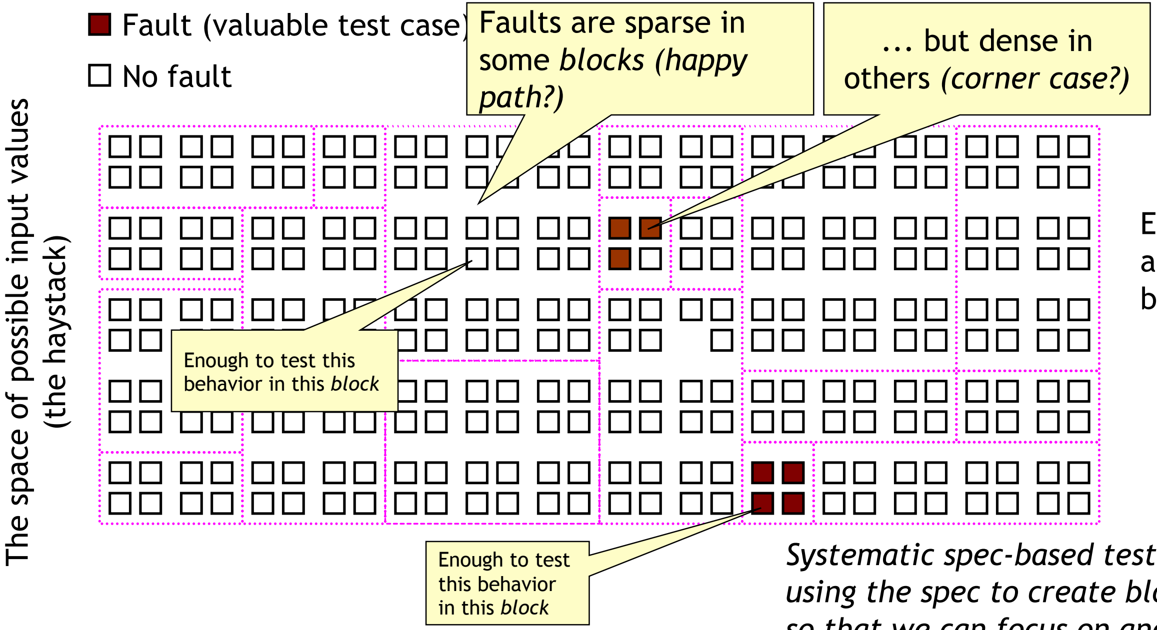
$= 1/2^{32} \approx 1$ in 4.3 billion

Purpose of spec-based testing is…

- 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 sink to the bottom?
    do magnets attract them?

Image: http://www.billfrymire.com/blog/finding-a-needle-in-the-haystack-image-of-the-week/

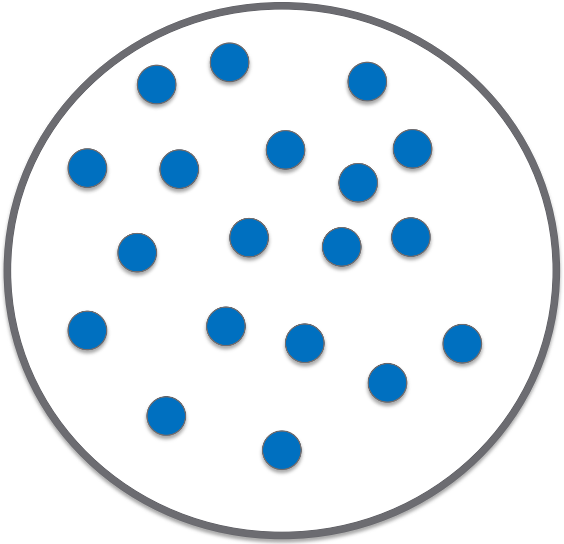
Bugs are rare: Partition the input space into blocks of related behaviors such that bugs are likely to be concentrated within a few blocks



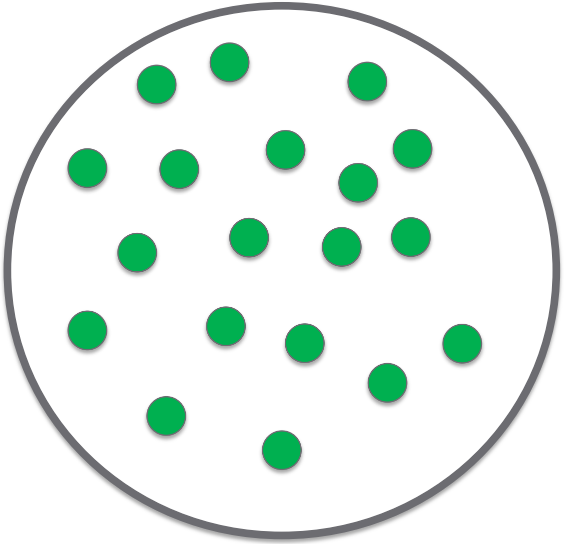
Each *block*: a cluster of related behaviors

Systematic spec-based testing is a way of using the spec to create blocks of behaviors so that we can focus on and isolate fault-dense regions

A block in a partition represents a bunch of “related” behaviors

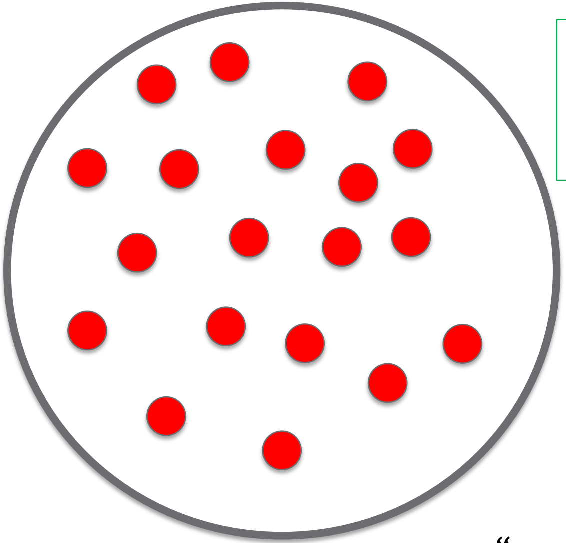


Inside a block: All or None



If you pick one to test, and it is ok, then the others in same block should be ok (or they have a high probability of passing as well) – ALL GREEN

Inside a block: All or None



If you pick one to test, and it fails, then the others in same block should fail too (or they have a high probability of failing as well) – ALL RED

"an equivalence class of behaviors"

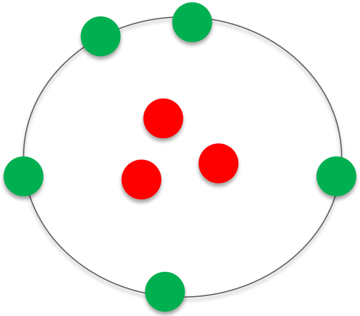
So what is the process we follow to get from a specification to a bunch of test cases

System Spec

Independently testable requirement	Independently testable requirement	Independently testable requirement
Input space of requirement: attributes	Input space of requirement: attributes	Input space of requirement: attributes
Characteristics derived from attributes and spec	Characteristics derived from attributes and spec	Characteristics derived from attributes and spec
Partition (set of blocks) for each characteristic	Partition for each characteristic	Partition for each characteristic
Valid/interesting combinations of blocks	Valid/interesting combinations of blocks	Valid/interesting combinations of blocks
Test Case Specs	Test Case Specs	Test Case Specs
Representatives for each block + oracle	Representatives for each block + oracle	Representatives for each block + oracle
Test Cases	Test Cases	Test Cases

We start with attributes: what obviously affect the requirement?
Attributes are more than just program inputs

Attribute: any explicit or implicit factor that may affect the underlying behavior of a piece of requirement: "*things that can vary*"



- Explicit (external stimuli)
  - an interface parameter
  - user input/action
  - sequence of user actions/inputs
  - database query
- Implicit (state-related)
  - state variable
  - global variable
  - variables appearing in pre- and post-conditions
  - persistent global data: state of input file, file system, database
  - environment variable
  - configuration: h/w or s/w platform, OS version, browser type, …

From attributes to characteristics --- attributes: *a, b, c*

An input domain: possible values/instantiations of an *attribute*

Example: quadratic root program

attributes: parameters *a*, *b*, *c*

domain of *a*, *b*, *c*: the set of real numbers R

D(*a*) = R

The input space: Cartesian product of input domains, or *composite input domain*

Example: input space of quadratic root program is R x R x R

R x R x R

A characteristic: a way to organize attributes and output properties for creating partitions into higher level abstractions

Example: quadratic root program

- Values of each attribute *a*, *b*, *c* may independently become a characteristic
- "multiplicity of real roots" (0, 1, or 2 roots; expressible only in terms of *all attributes*) – *relates to the properties of outputs/effects*

"multiplicity of roots": 0, 1, or 2

Simple example: reverse ZIP code lookup

Input: ZIP code (5-digit US Postal code)

Output: List of (0 or more matches, a ZIP code may touch multiple)

Spec? Spec is the above two lines

Attributes?

Input Space?

Characteristic?

Simple example: reverse ZIP code lookup

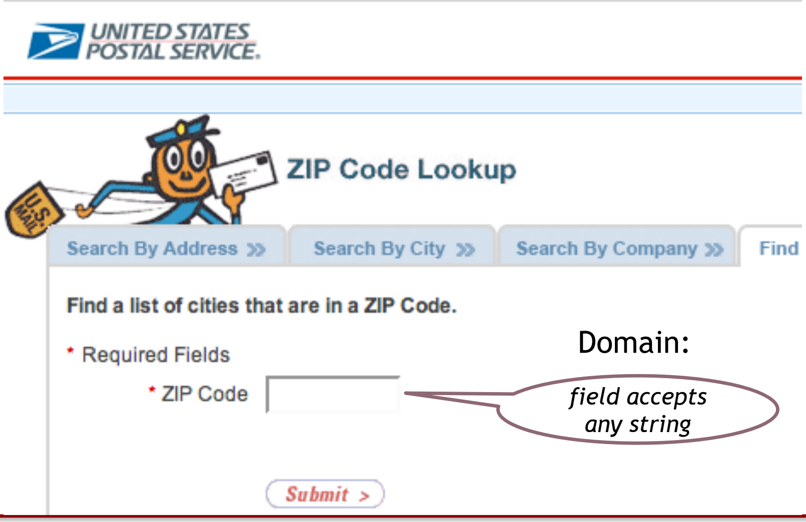
Input: ZIP code (5-digit US Postal code)

Output: List of (0 or more matches, a ZIP code may touch multiple)

Attributes?
- Single: Zip Code (a string)

Input Space?
- All Strings

Characteristics?
- No. of Matches
- Syntax (Validity of input)



Domain:

field accepts any string

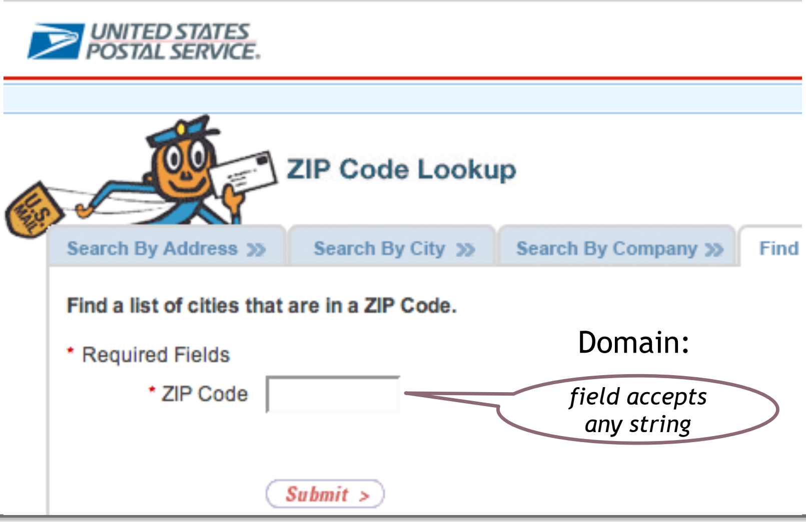
Simple example: reverse ZIP code lookup

Input: ZIP code (5-digit US Postal code)

Output: List of (0 or more matches)

Characteristics?
- No. of Matches
- Syntax (Validity of input)

- How should we partition these?
- What are some possible blocks to consider for testing for each characteristic?



Domain:

field accepts any string

Zip code example: partitions and blocks

Simple example with one input, one output

- No. of Matches (Multiplicity)
  - Unassigned zip code: empty set of
  - Contained zip code: yields exactly one town
  - Overlapping zip code: yields more than one town
- Syntax (Validity)
  - Empty input
  - Containing 1-4 digits
  - Containing 5 digits
  - Containing 6 digits
  - Containing non-digit characters
  - Very long input

Note the prominence of boundary values (0 , 1 town, 5 digits, 6 digits) and error cases

Zip code example:
formalized characteristics, partitions, and blocks

characteristic

Syntax (Validity)

Subclassified as valid/invalid

partition

1. 5 digits: *valid*
2. Empty: *invalid*
3. 1-4 digits: *invalid*
4. 6 digits: *invalid*
5. Non-digit characters: *invalid*
6. Long input (> 6 chars): *invalid*

block in partition

Multiplicity

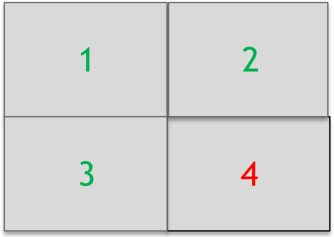
Completed by adding an "undefined" block

partition

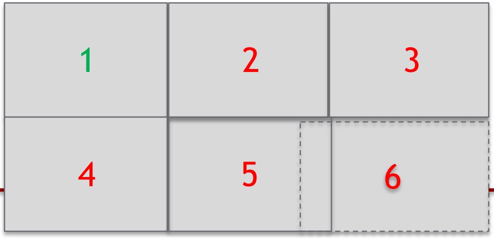
1. Unassigned zip code: empty set of towns
2. Contained zip code: yields exactly one town
3. Overlapping zip code: yields more than one town
4. Undefined

To complete the partition: what happens when ZIP Code is invalid: then there is no multiplicity per se

Multiplicity



Syntax



Zip code example: now we must combine blocks from different characteristics and try to write a test case for each to cover them

characteristic

But we cannot combine all blocks with all other blocks

partition

Syntax

1. 5 digits: *valid*
2. Empty: *invalid*
3. 1-4 digits: *invalid*
4. 6 digits: *invalid*
5. Non-digit characters: *invalid*
6. Long input (> 6 chars): *invalid*

block in partition

partition

Multiplicity (constraint: Validity = *valid*)

1. Unassigned zip code: empty set of towns
2. Contained zip code: yields exactly one town
3. Overlapping zip code: yields more than one town
4. Undefined

Don't have 4 x 6 = 24 combinations

- Some combinations are impossible by definition
- In reality, we have:
  1 x 3 + 1 x 5 = 8 combinations
  (green x green) + (red x red)

Combinatorial constraints

- Syntax: for block 1, Multiplicity must be *defined*
- Multiplicity: for block 4, Syntax must be *invalid*

Zip code example: one last refinement/correction

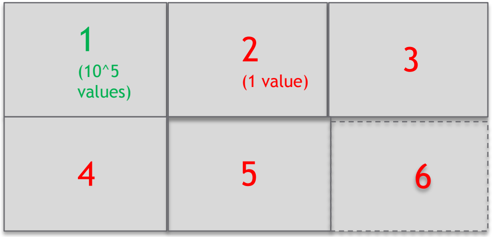
Simple example with one input, one output

Syntax
1. 5 digits: *valid*
2. Empty: *invalid*
3. 1-4 digits: *invalid*
4. 6 digits: *invalid*
5. Non-digit characters (1 to 5 chars): *invalid*
6. Long input (> 6 chars): *invalid*

No longer overlap between 5 and 6

Multiplicity (constraint: Validity = *valid*)
1. Unassigned zip code: empty set of towns
2. Contained zip code: yields exactly one town
3. Overlapping zip code: yields more than one town
4. Undefined

Syntax



Not equally sized!

Zip code example: write the 8 test cases for this input space model

Simple example with one input, one output

Syntax

1. 5 digits: *valid*
2. Empty: *invalid*
3. 1-4 digits: *invalid*
4. 6 digits: *invalid*
5. Non-digit characters (1 to 5 chars): *invalid*
6. Long input (> 6 chars): *invalid*

Multiplicity (constraint: Validity = *valid*)

1. Unassigned zip code: empty set of towns
2. Contained zip code: yields exactly one town
3. Overlapping zip code: yields more than one town
4. Undefined

Example Zip codes:

- 15200 => no match
- 15201 => {Pittsburgh}
- 15106 => {Heidelberg, Carnegie}

Zip code example: write the 8 test cases for this input space model

Simple example with one input, one output

Syntax

1. 5 digits: *valid*
2. Empty: *invalid*
3. 1-4 digits: *invalid*
4. 6 digits: *invalid*
5. Non-digit characters (1 to 5 chars): *invalid*
6. Long input (> 6 chars): *invalid*

Multiplicity (constraint: Validity = *valid*)

1. Unassigned zip code: empty set of towns
2. Contained zip code: yields exactly one town
3. Overlapping zip code: yields more than one town
4. Undefined

- “15200” => no match
- “15201” => {Pittsburgh}
- “15106” => {Heidelberg, Carnegie}
- “” => invalid
- “123” => invalid
- “123456” => invalid
- “a$12” => invalid
- “12345abcdef” => invalid

Summary

- Systematic spec-based testing is generation of test cases from specifications in a focused, smart, and flexible way
  - *widely applicable at all levels*
- This involves partitioning the input space into blocks that preferably represent "equivalence classes"
- Systematic testing is intentionally non-uniform to address special cases, error conditions, and other small blocks of inputs
  - *metaphor: dividing a big haystack into variable-size, hopefully internally-uniform piles, where the needles might be concentrated in smaller piles*

Sources

- Pezze & Young, *Software Testing & Analysis*
- Jorgensen, *Software Testing: A Craftsman's Approach*
- Ammann & Offutt: *Introduction to Software Testing*

Several illustrations and examples have been adopted from these sources and accompanying instructor materials, with various adaptations