Topics for this Lecture

- Functional Testing vs Structural Testing
- Understand the rationale for systematic (non-random) selection of test cases
- How we do design test cases

Oregon State University

Testing Techniques

- **How to test software.**
  - How do we design tests?

Functional Testing	Structural Testing
Called “black-box” or “specification-based” testing	Called “white-box” testing
We ignore how the program is being written.	The program is the base.
Test based on the specification.	Test based on code.
Test covers as much *specified* behavior as possible.	Test covers as much *implemented* behavior as possible.

Why Functional?

- **Program code is not necessary.**
  - Only a description of intended behavior is needed
  - Send a program a stream of inputs, observe the outputs, decide if the system passed or failed the test.
- **Deriving test cases from program specifications**
  - Functional specification = description of intended program behavior
  - Functional refers to the source of information used in *test case design,* not to *what is tested*
- **Early functional test design has benefits**
  - Often reveals ambiguities and inconsistency in spec.
  - Gives additional explanation of spec.

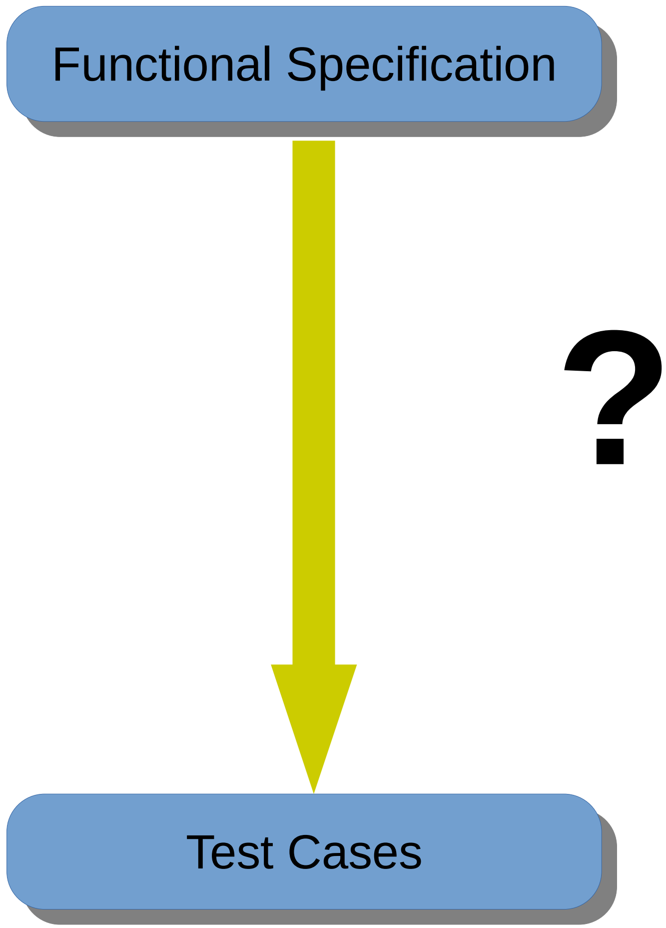
OSU
Oregon State
UNIVERSITY

Why Functional?

- **Best for missing logic faults**
  - Common problem: Some program logic was simply forgotten.
  - Structural (code-based) testing can not detect that some required feature is missing in the code.
- **Applies at all granularity levels**
  - Functional testing can be applied at any level where some for of specification are available. (unit tests • integration tests • system tests • regression tests)
  - Structural testing is tied to program structures, and applies to unit and integration testing.

Oregon State UNIVERSITY

From Specifications To Test Cases



The starting point of black box testing is a **description** of the software

How do we get from the **functional specification** to **test cases**?

The final result of black box testing is a set of **test cases**

Random vs Systematic Testing

- **“What test cases should I use to exercise my program?”**

1. Random (uniform):
   - Pick possible inputs uniformly
     - You select your inputs from a set where each input is equally probable.
   - 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)
   - Accidental bias may be avoided by choosing test cases from a random distribution. But treats all inputs as equally valuable
2. 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

Oregon State UNIVERSITY

Why Not Random?

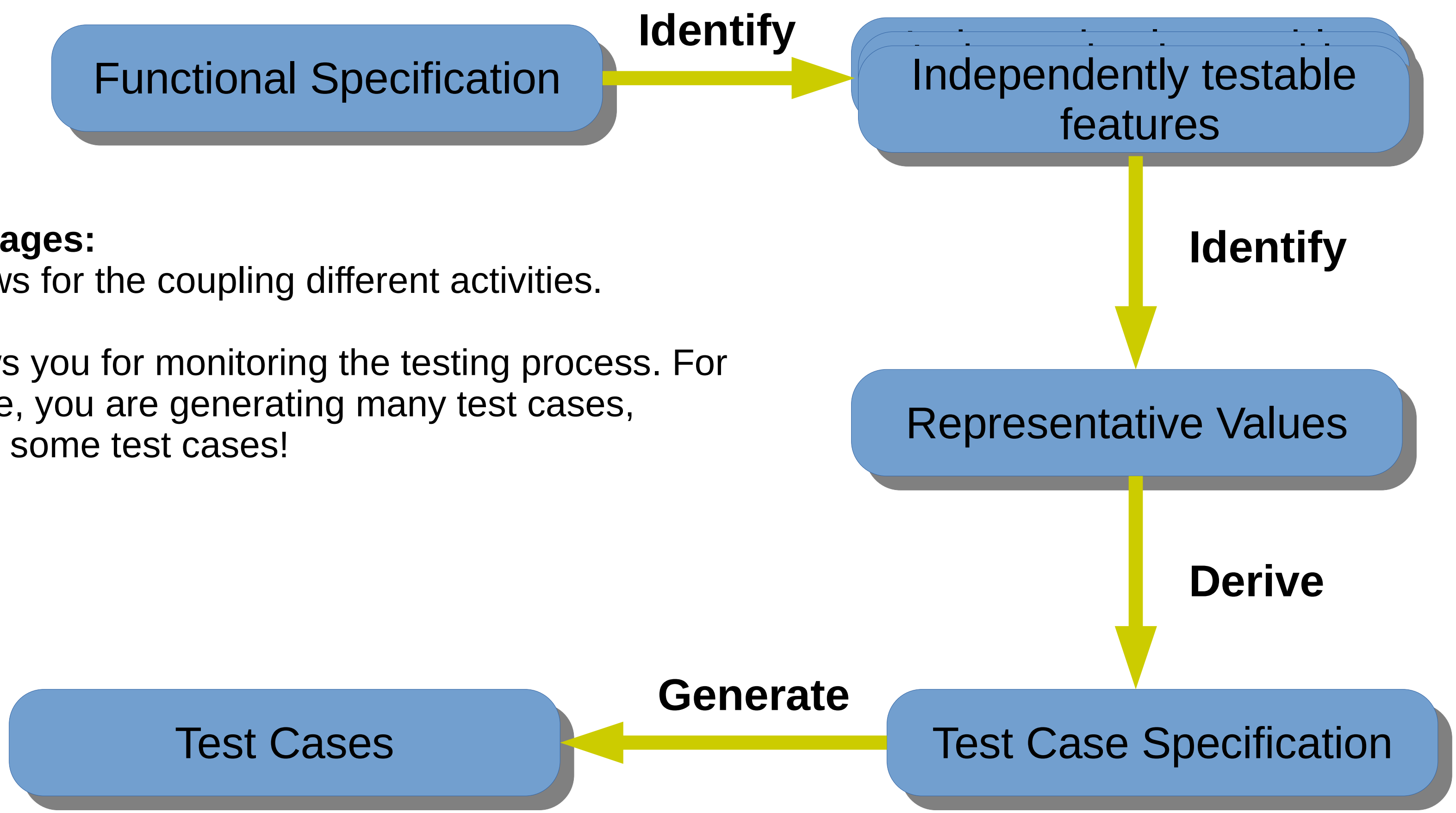
Why Not Random?

- Faults are not distributed uniformly
- **Example**: Assume a Java class "Root" finds the two roots of a quadratic equation.

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

  and fails if $b^2 - 4ac = 0$ and $a=0$
- Random sampling is unlikely to choose a=0.0 and b=0.0
- Failing values are sparse in the domain input space — needles in a very big haystack

A Systematic Functional-Testing Approach



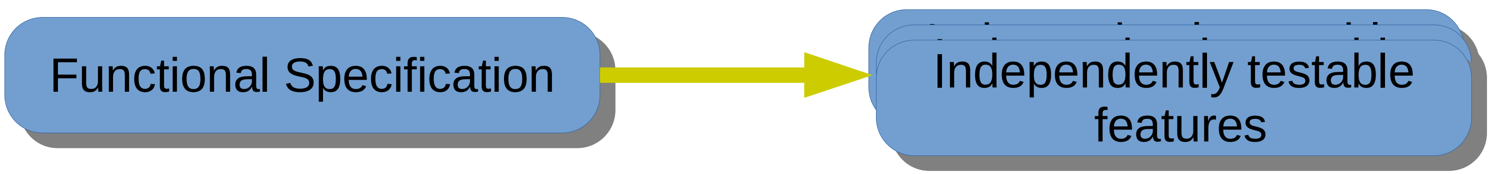
Advantages:
- It allows for the coupling different activities.

-It allows you for monitoring the testing process. For example, you are generating many test cases, missing some test cases!

Oregon State UNIVERSITY

Systematic Functional Testing

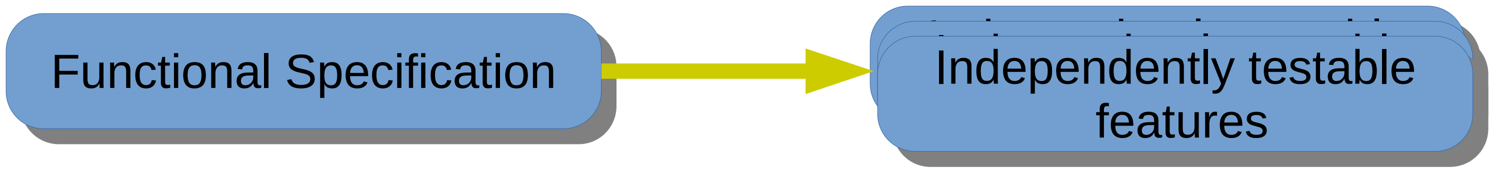
- **Identify Independently Testable Features**



  - The preliminary step of functional testing consists in partitioning the specifications into features can be tested separately. Decompose the software under test (**SUT**) into independently testable features
  - An Independently Testable Feature (**ITF**) is a functionality that can be tested independently of other functionalities of the **SUT**.
  - An ITF need not correspond to a unit test or subsystem of the software
    - For testing, programs can be decomposed : **Features**: observable behavior vs **Units**, **subsystems** and **components**: the structure of the software system.

Oregon State UNIVERSITY

Systematic Functional Testing



What are the independently testable features?

- Consider a file sorting utility that maybe capable of merging two sorted files.

- Consider the Root class solves the two roots of a quadratic equation $ax^2+bx+c=0$ and produces the two values of x (i.e., root_one and root_two)

Systematic Functional Testing

- **What are the independently testable features?**

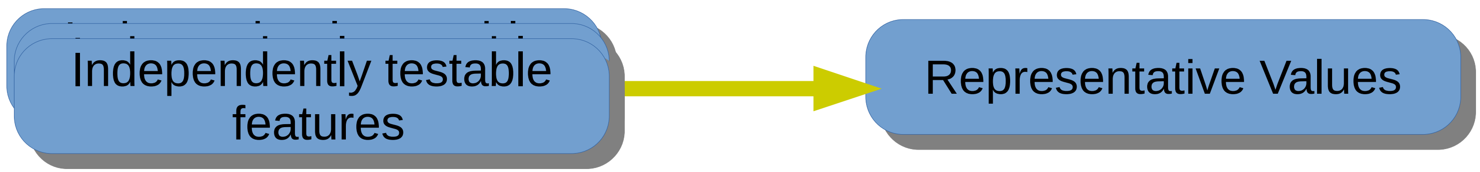
Functional Specification → Independently testable features

- Consider a file sorting utility that maybe capable of merging two sorted files.
    - **TWO ITFs**: we might test the sorting and merging functionalities separately.

- Consider the root class solves the two roots of a quadratic equation $ax^2+bx+c=0$ and produces the two values of x (i.e., root_one and root_two)
    - Just **ONE ITF**: Root class is a unit and thus provides exactly single testable feature.

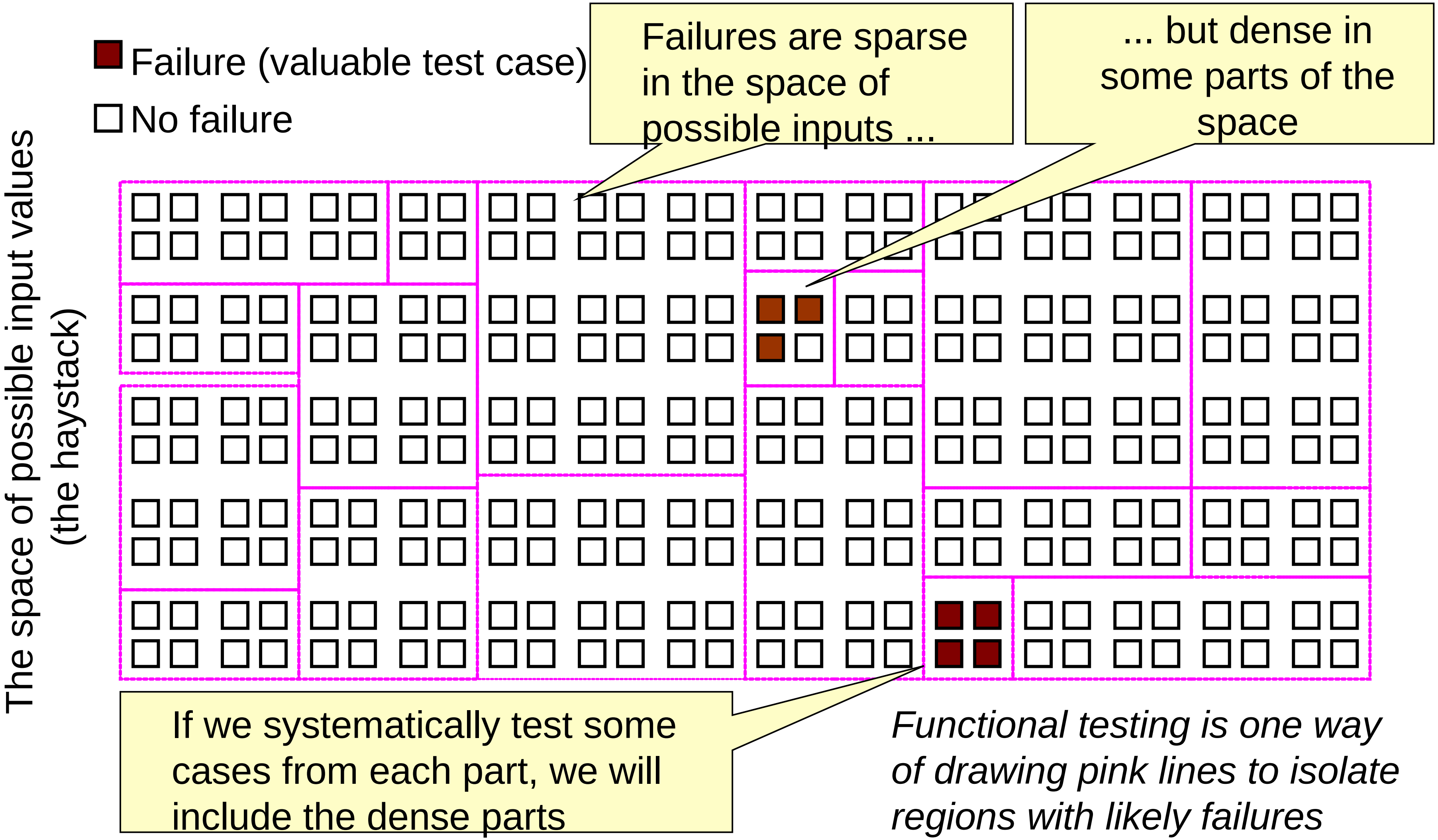
Systematic Functional Testing

- **Representative Values**



  - The next step is identifying which values of each input should be selected to form test cases
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of equivalence classes that are opt to fail often or not all.

Systematic Partition Testing



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

Equivalence Partitioning

- ***Equivalence partitioning***, sometimes called ***equivalent classing****,* is the process of reducing the huge (infinite) set of possible test cases into a much smaller, but still equally effective, set.
- An ***equivalence partition*** or ***equivalence class*** is a set of test cases that tests the same thing or reveals the same bug.
- In other words, divide your input conditions into groups (classes)
  - Input in the same class should behave similarly in the program
- **How do we choose *equivalent partitions/classes*?**
  - The key is to examine input conditions from the spec. and think about ways to group similar inputs, similar outputs and similar operations of the software under test.
  - Each input condition induces an equivalence class– valid and invalid inputs.

Example

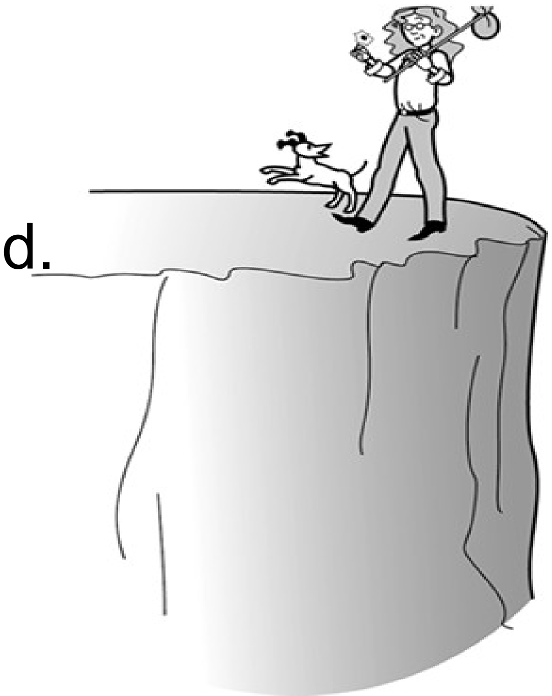
public int **Split** (String str, int size)// takes a string and split it into sub string, into chunks of size characters each

Some possible partitions:

- size < 0 “incorrect size”
- size = 0 “partition with a single element”
- size > 0 “ a standard case”
- str with length < size
- str with length in [size, size x 2]
- str with legth > size x 2
- ….

Boundary Conditions

- **How do we choose representatives from equivalence classes?**
  - If you can safely and confidently walk along the edge of a cliff without falling off, you can almost certainly walk in the middle of a field.
  - If software can operate on the boundary edge of its capabilities, it will almost certainly operate well under normal conditions.
  - You need to create two equivalence partitions:
    - The first should contain data that you would expect to work properly- test the valid data just inside the boundary of an equivalence class.
    - The second partition should contain data that you expect to cause an error – test the invalid data just outside and at the boundary.



Example

public int **Split** (String str, int size)// takes two inputs,a string and size, and splits the string into substrings, into chunks of size characters each.

Some possible partitions:

- size < 0 “incorrect size”
- size = 0 “partition with a single element”
- size > 0 “ a standard case”
- str with length < size
- str with length in [size, size x 2]
- str with legth > size x 2

Some possible inputs:

- size =-1
- size = 1
- size = MaxInt “boundary”

- string with length size-1
- string with length size
- …..

Example

public int **Split** (String str, int size)// takes a string and split it into sub string, into chunks of size characters each

Some possible inputs:

- size =-1 - string with length size-1
- size = 1 - string with length size
- size = MaxInt "boundary" - …..

Test Case Specifications:

- size =-1 - string with length -2
- size =-1 - string with length -1
- size = 1 - string with length 0

Example

public int **Split** (String str, int size)// takes a string and split it into sub string, into chunks of size characters each

Some possible inputs:

- size =-1 - string with length size-1
- size = 1 - string with length size
- size = MaxInt "boundary" - …..

Test Case Specifications:

- ~~size =-1~~ ~~- string with length -2~~
- ~~size =-1~~ ~~- string with length -1~~
- size = 1 - string with length 0

Example: Search Routine Specification

procedure Search (Key : *in* ELEM ; T: **in** ELEM_ARRAY;
    Found : **out** BOOLEAN; L: **out** ELEM_INDEX) ;

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)
or
-- the element is not in the array
(**not** Found and
not (**exists** i, 1 <= i <= N, T (i) = Key))

Example: Search Routine - input partitions

- Inputs which conform to the pre-conditions
- Inputs where a pre-condition does not hold
- Inputs where the key element is a member of the array
- Inputs where the key element is not a member of the array

Example: Search Routine - input partitions

Array	Element
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array

Example: Search Routine – Test Cases

Input array (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 6
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Partition Testing vs. Random Testing

- Partition testing typically more **expensive** than random generating data.
- Partition testing usually produces **fewer** test cases than random testing for the same expenditure of time and money.
- Partitioning can therefore be **advantageous** only if the average value (fault detection effectiveness) is greater
- Generally, random inputs are **easier** to generate, but less likely to cover parts pf the specification or the code.
- Gutjahr's states that Partition testing is more effective than random testing. "*Gutjahr, W. J. (1999). Partition Testing vs. Random Testing: The Influence of Uncertainty. IEEE Transactions on Software Engineering, 25(5), 661-674.*"
- Given a fixed budget, the optimum not lie in only partition testing or random testing, but some mix that use of available knowledge.

OSU
Oregon State
UNIVERSITY

References:
Pezze + Young, “Software Testing and Analysis”, Chapter 10 & 11
Patton, Ron. "Software Testing." (2000). Chapter 4 & 5
Sommerville, I., Software Engineering, Sixth Edition, Addison-Wesley, 2001
Chapter 20

Oregon State UNIVERSITY