Systematic Functional Testing (Black Box Testing)

Credits:

Code Complete, 2nd Ed., Steve McConnell

Software Engineering, 5th Ed., Roger Pressman

Testing Computer Software, 2nd Ed., Cem Kaner, et. Al.

"Software Testing and Analysis", (Pezzè and Young), and Saarland University course on Software Testing (Fraser and Zeller)

Systematic Functional Testing

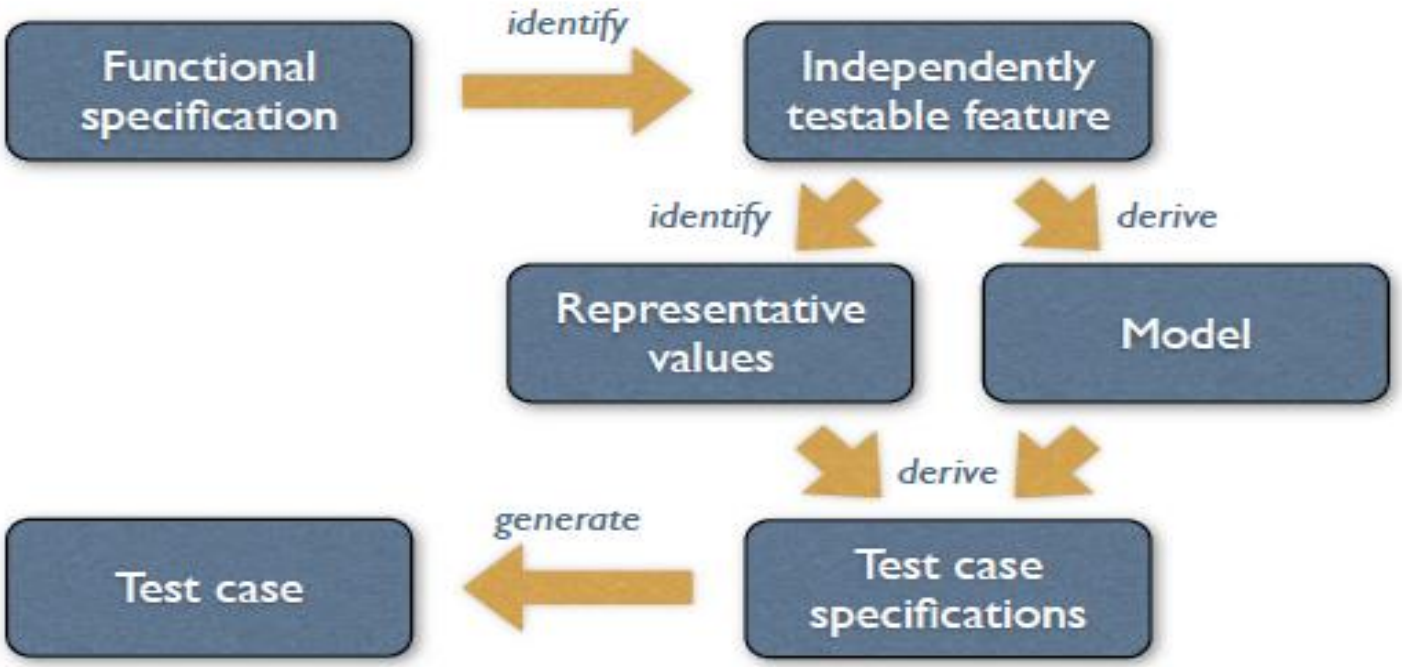
- Testing software against a specification of its external behavior without knowledge of internal implementation details
  - Can be applied to software “units” (e.g., classes) or to entire programs
  - External behavior is defined in API docs, Functional specs, Requirements specs, etc.
- Because this testing purposely disregards the program's control structure, attention is focused primarily on the information domain (i.e., data that goes in, data that comes out)
- Our Goal: Derive sets of input conditions (test cases) that fully exercise the external functionality

Systematic Functional Testing

- This testing tends to find different kinds of errors than white box testing
  - Missing functions
  - Usability problems
  - Performance problems
  - Concurrency and timing errors
  - Initialization and termination errors
  - Etc.

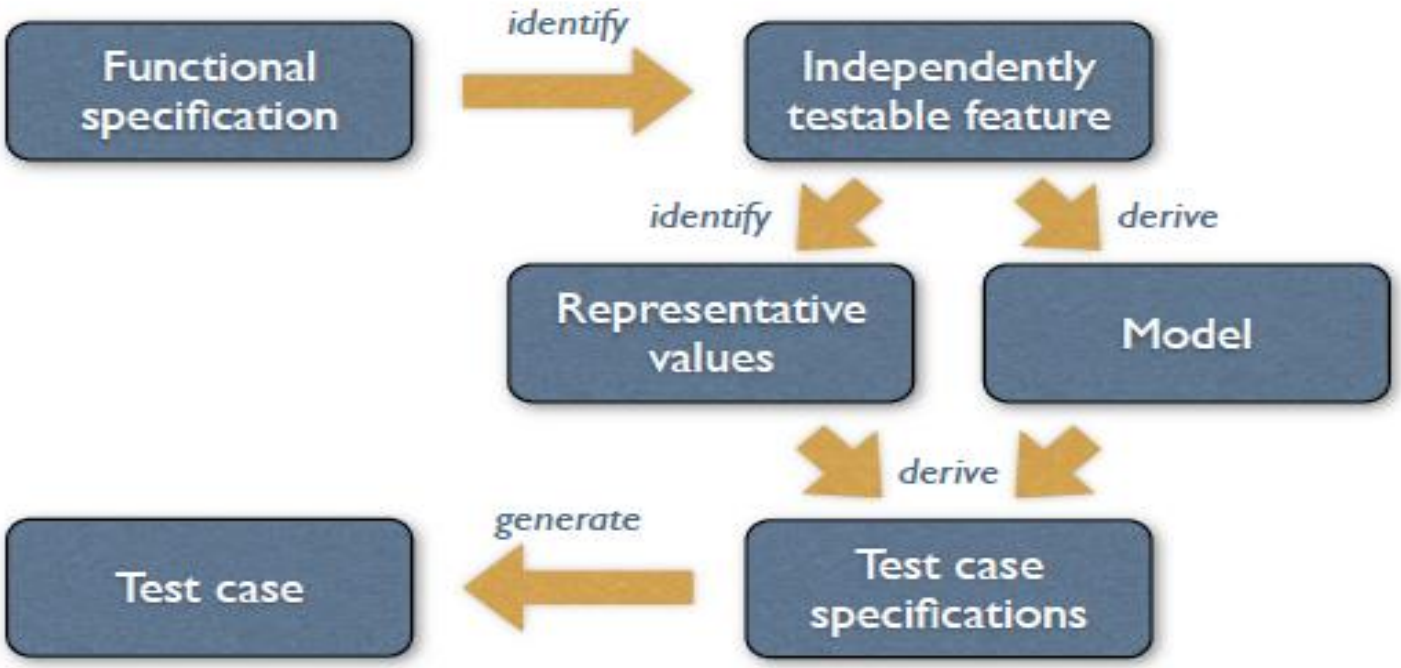
- Unlike white box testing, functional testing tends to be applied later in the development process

Systematic Functional Testing



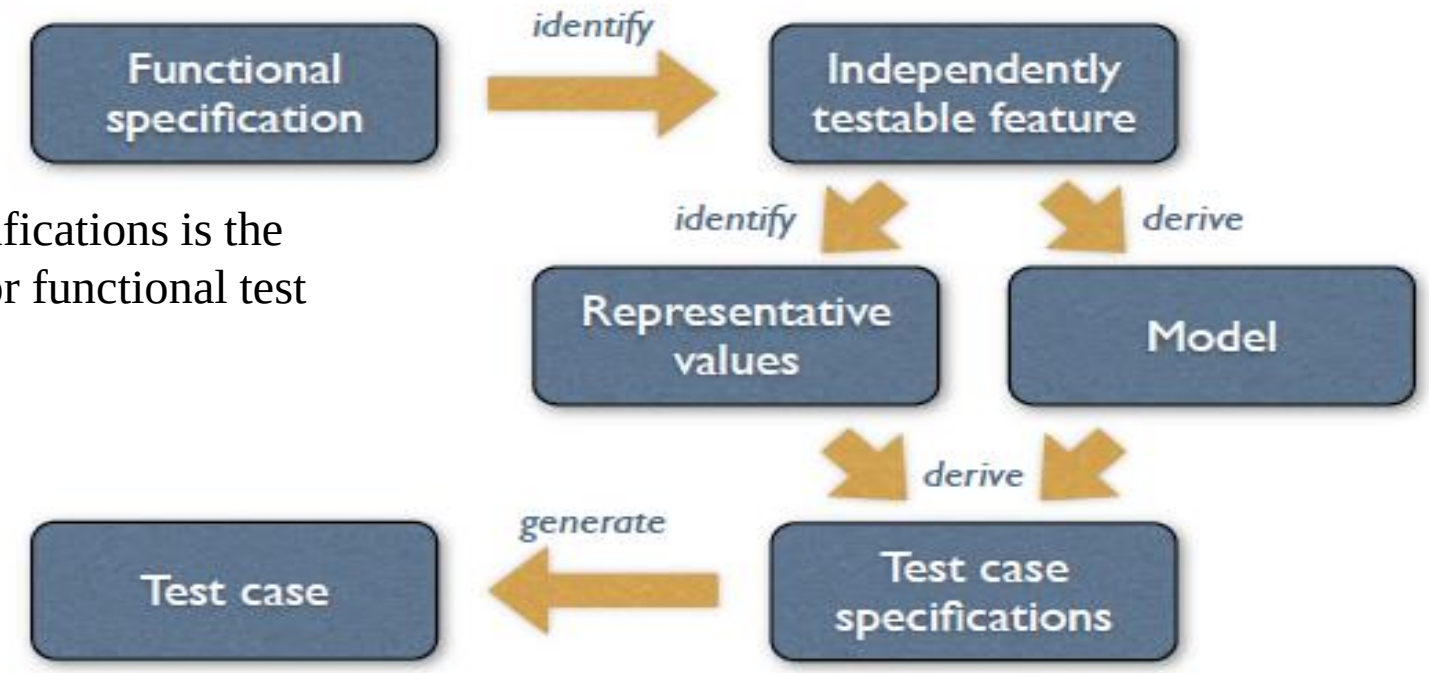
Source: "Software Testing and Analysis", (Pezzè and Young), and Saarland University course on Software Testing (Fraser and Zeller)

Systematic Functional Testing



This is a general approach that can be utilized at **all levels of testing**, but still needs to be adapted to the situation at hand and the software under test.

Systematic Functional Testing



functional specifications is the starting point for functional test design

Example 1: Unix cat command man page

Functional Specification

```
NAME  cat - concatenate files and print on the standard output
SYNOPSIS  cat [OPTION] [FILE]...
DESCRIPTION  Concatenate FILE(s), or standard input, to standard output.
       -A, --show-all equivalent to -vET
       -b, --number-nonblank number nonblank output lines
       -e     equivalent to -vE
       -E, --show-ends display $ at end of each line
       -n, --number number all output lines
       -s, --squeeze-blank never more than one single blank line
       -t     equivalent to -vT
       -T, --show-tabs display TAB characters as ^I
       -u     (ignored)
       -v, --show-nonprinting use ^ and M- notation, except for LFD and TAB
       --help display this help and exit
       --version output version information and exit
       With no FILE, or when FILE is -, read standard input.
EXAMPLES
       cat f - g Output f's contents, then standard input, then g's contents.
       cat    Copy standard input to standard output.
```

Example 2: Javadoc – Triangle Class

Class Triangle

```
java.lang.Object
  └Triangle
```

---

public class **Triangle**
extends java.lang.Object

Triangle. The main function takes 3 positive whole-number lengths to be typed in as command line arguments. The program responds with a description of the triangle, as follows:

- **equilateral** - if all three sides have equal length
- **isosceles** - if two sides have equal length
- **right-angled** - if one angle is a right angle
- **scalene** - all sides different lengths, no right angles
- **impossible** - if the given side lengths do not form a triangle

Area and perimeter of the triangle are calculated, too.

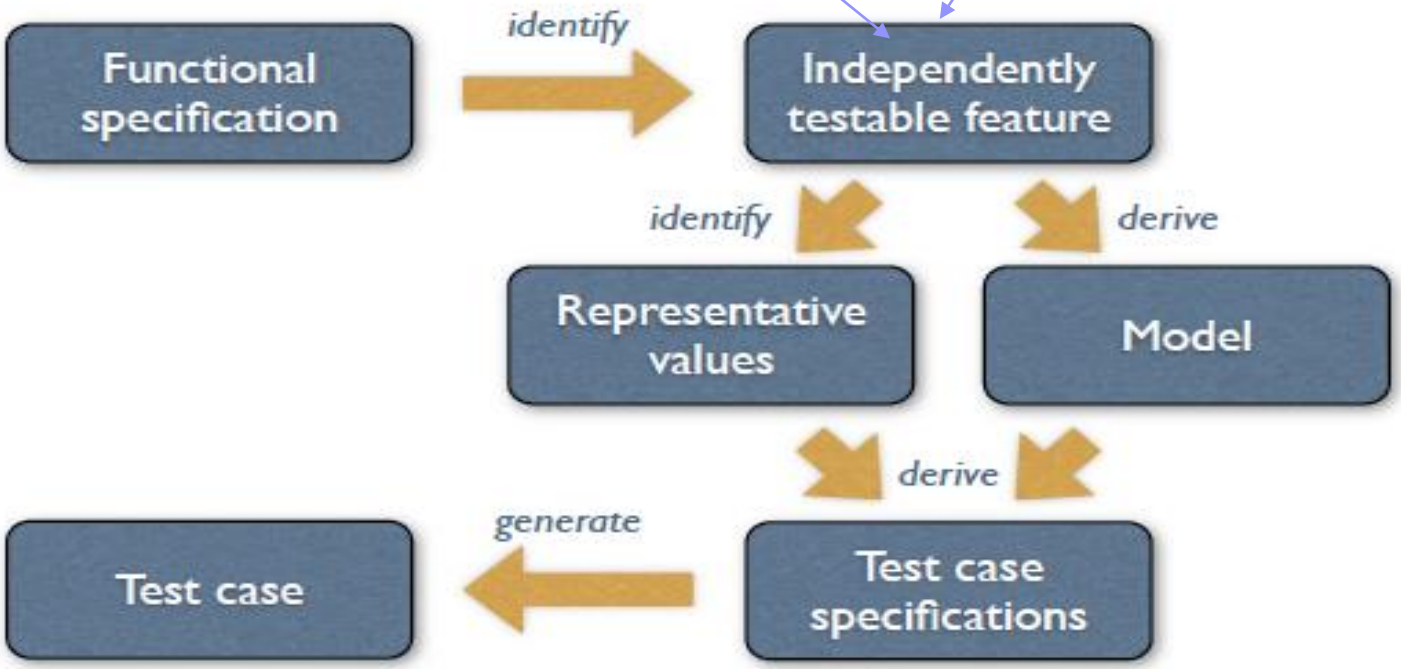
Constructor Summary

`Triangle(int s1, int s2, int s3)` Constuctor (without error checking)

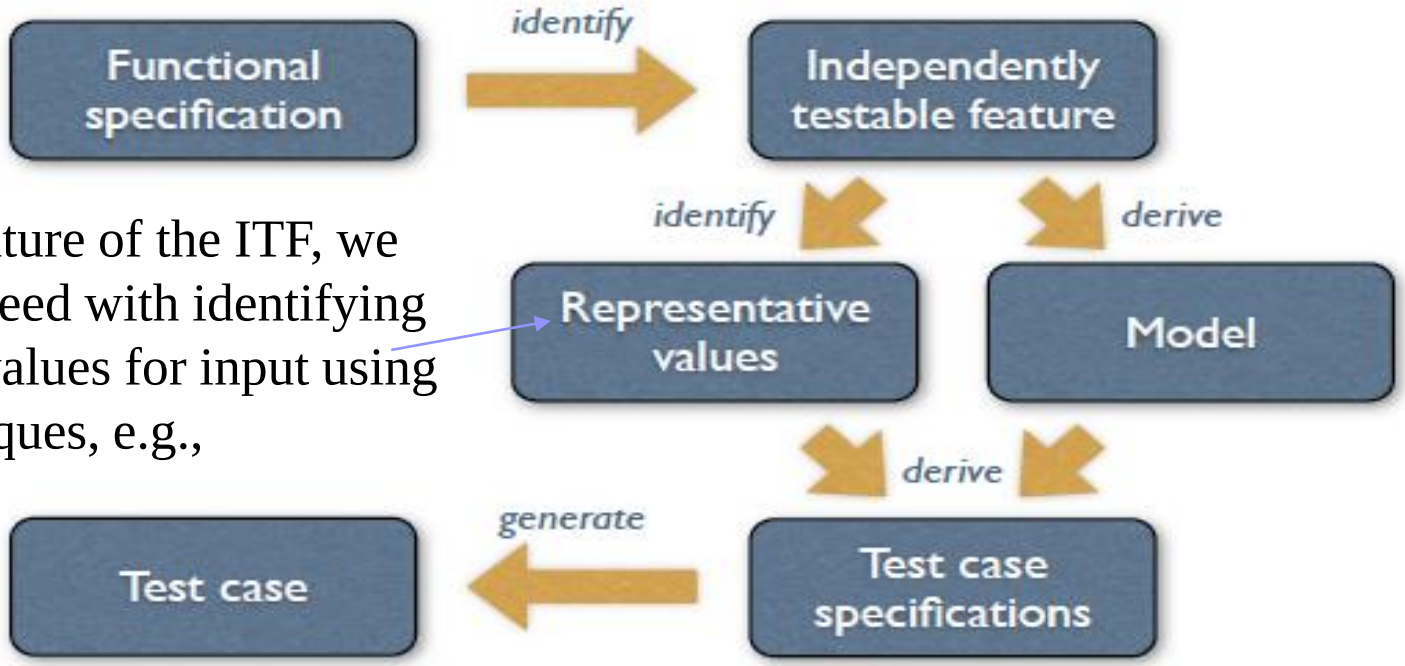
For example, the withdrawal function of an ATM could be an ITF (because this function is testable).

Based on the functional specification, independently testable features (ITFs) are identified. An ITF is a part of the functionality of the software that can be tested separately from other functionalities. An important aspect of an ITF is its inputs.

Systematic Functional Testing



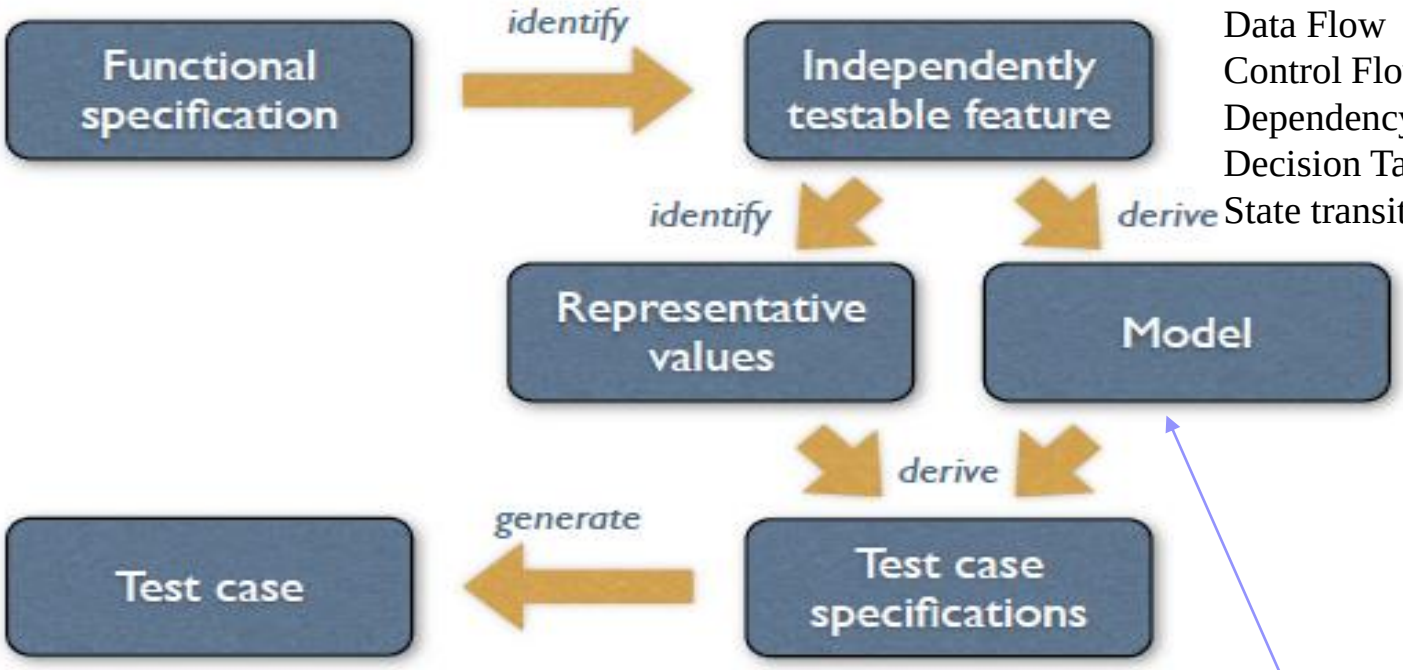
Systematic Functional Testing



Based on the nature of the ITF, we may either proceed with identifying representative values for input using different techniques, e.g.,

• **Equivalence partitioning** (more context in later slides) of the input space • **Boundary value analysis** • Different types of random selection

Systematic Functional Testing



Some model tools

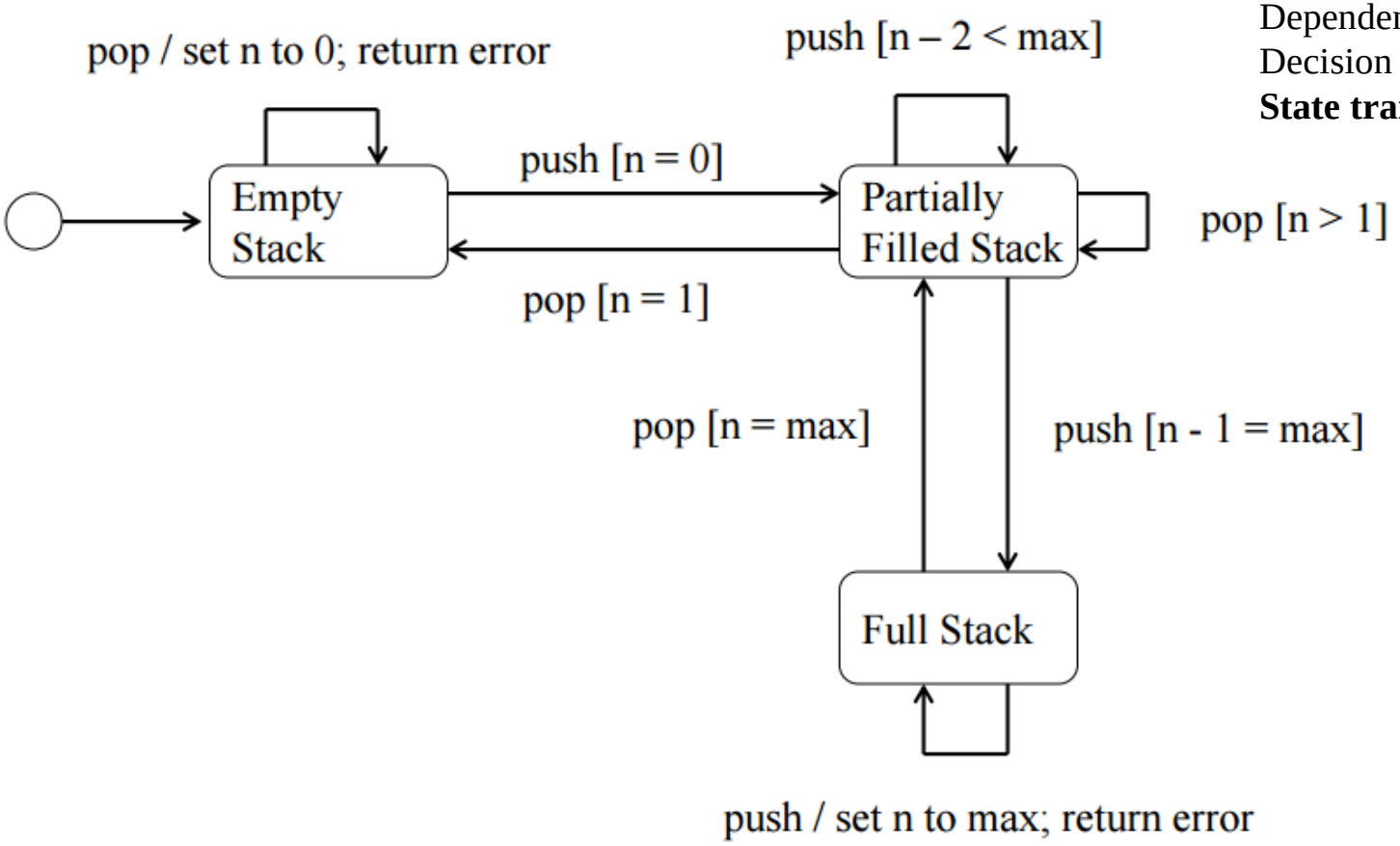
Data Flow
Control Flow
Dependency Graphs
Decision Tables
State transition machines

We may even derive a test Model that describes (parts of) the specification in a more formal way, and that helps us derive test cases

Example state diagram for Stack ADT

Some model tools:

Data Flow
Control Flow
Dependency Graphs
Decision Tables
State transition machines



Example Decision Table

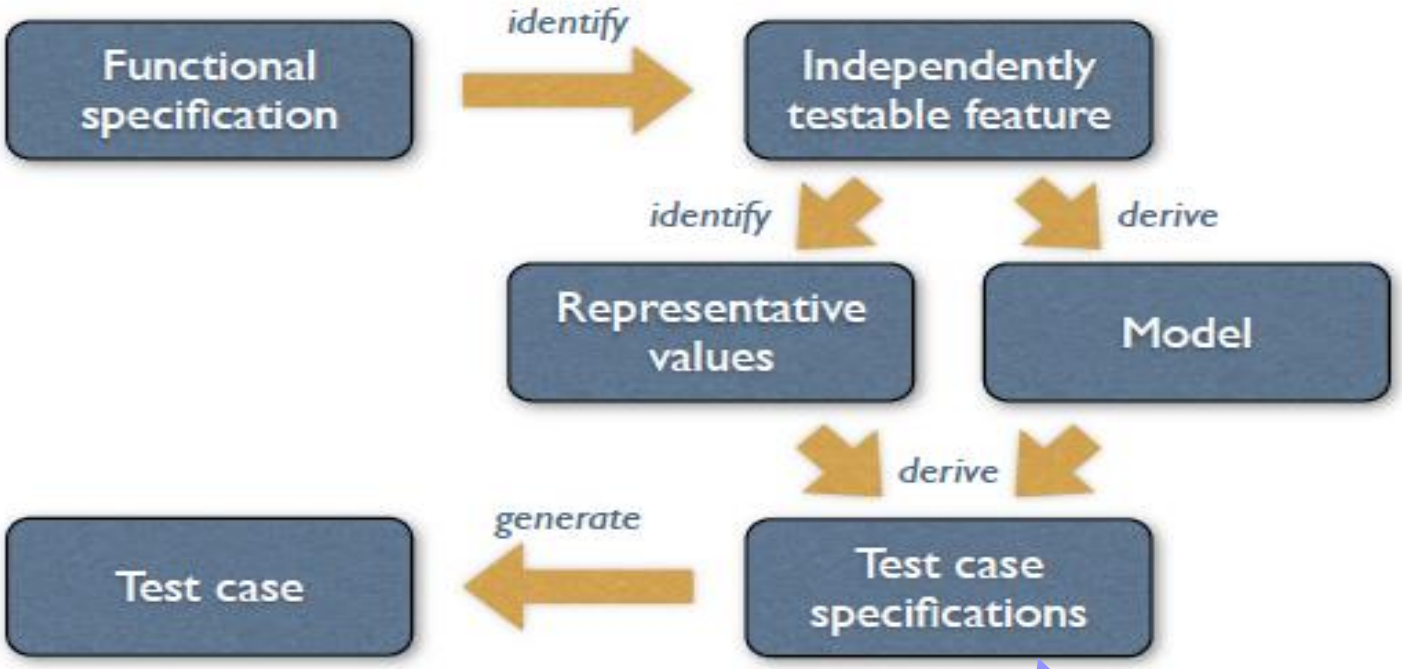
Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Credit status = accepted	–	Y	–	–	Y
Credit status <> accepted	–	–	–	Y	–
Credit limit = 0	–	Y	–	–	N
Credit limit > 0	–	–	–	–	Y
Credit limit = blank/unknown	Y	–	–	N	N
Next review date = blank/unknown	–	–	Y	N	N
Next review date = valid date	–	–	–	Y	Y
Outcomes					
Run credit rules				X	X
Fail credit processing	X	X	X		

Some model tools:

Data Flow
Control Flow
Dependency Graphs
Decision Tables
State transition machines

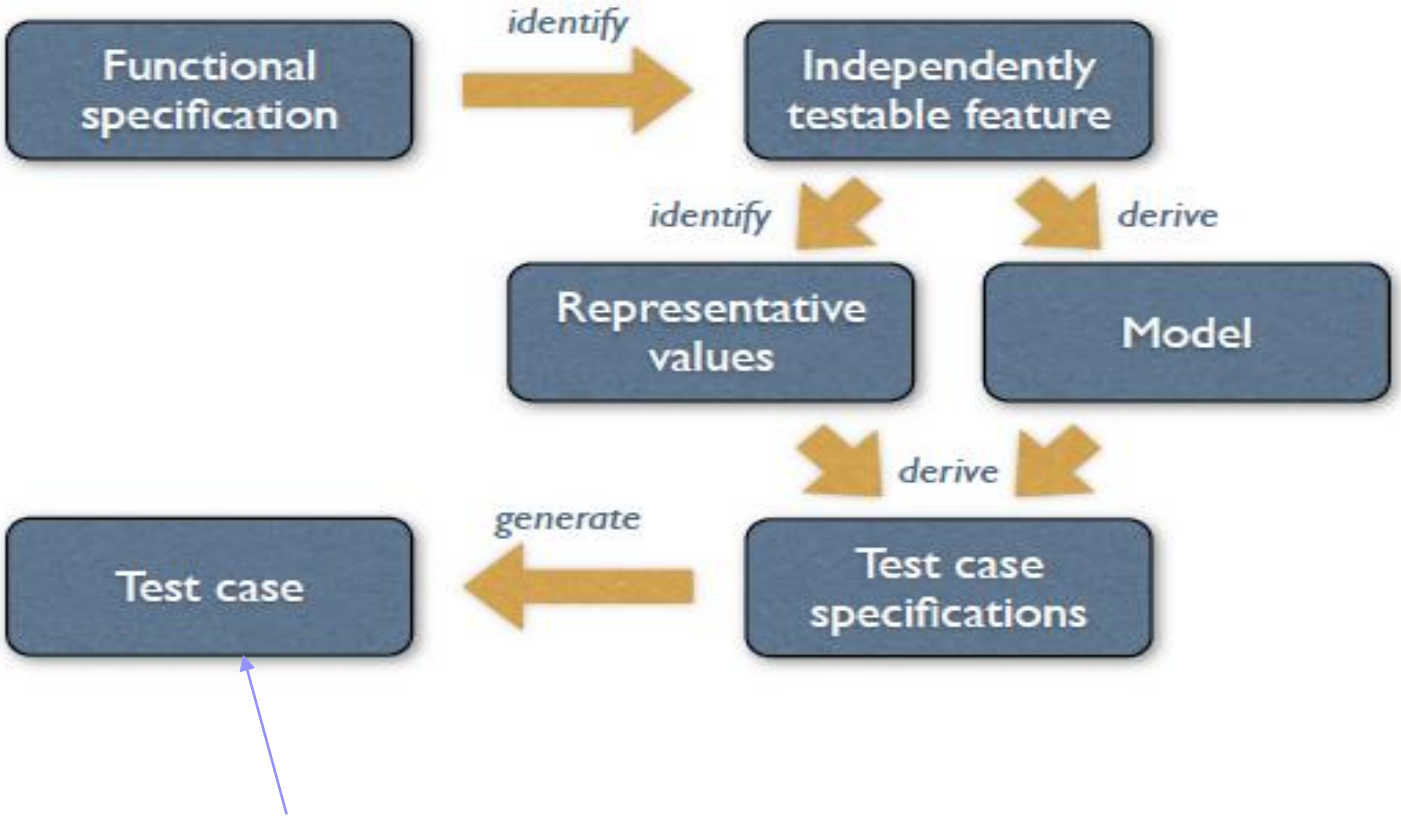
Source: http://www.seilevel.com/requirements/decision-tables-simplify

Systematic Functional Testing



Based on the representative values, or/and the test model, **test specifications** are derived. Test specifications are abstract test requirements that must be met during testing. Note: Often a given test specification could be met by a **large number of possible sets** of test cases.

Systematic Functional Testing



As a last step in the test design process, we create test cases that meet the test specifications. Here that a test case not only requires an input, but also an expected output in order to be able to provide a verdict. The expected output is typically derived based on the specifications.

Summary: Functional Testing

1. Identify independently-testable features
2. Defining all the inputs to the features
3. Identify representative classes of values/Build Model
4. Generate test case specifications
5. Generate and instantiate test cases

The Information Domain: inputs and outputs

- Defining the input domain
  - Boolean value
    - T or F
  - Numeric value in a particular range
    - 99 <= N <= 99
    - Integer, Floating point
  - One of a fixed set of enumerated values
    - {Jan, Feb, Mar, …}
    - {Visa, MasterCard, Discover, …}
  - Formatted strings
    - Phone numbers
    - File names
    - URLs
    - Credit card numbers
    - Regular expressions

A Challenge

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

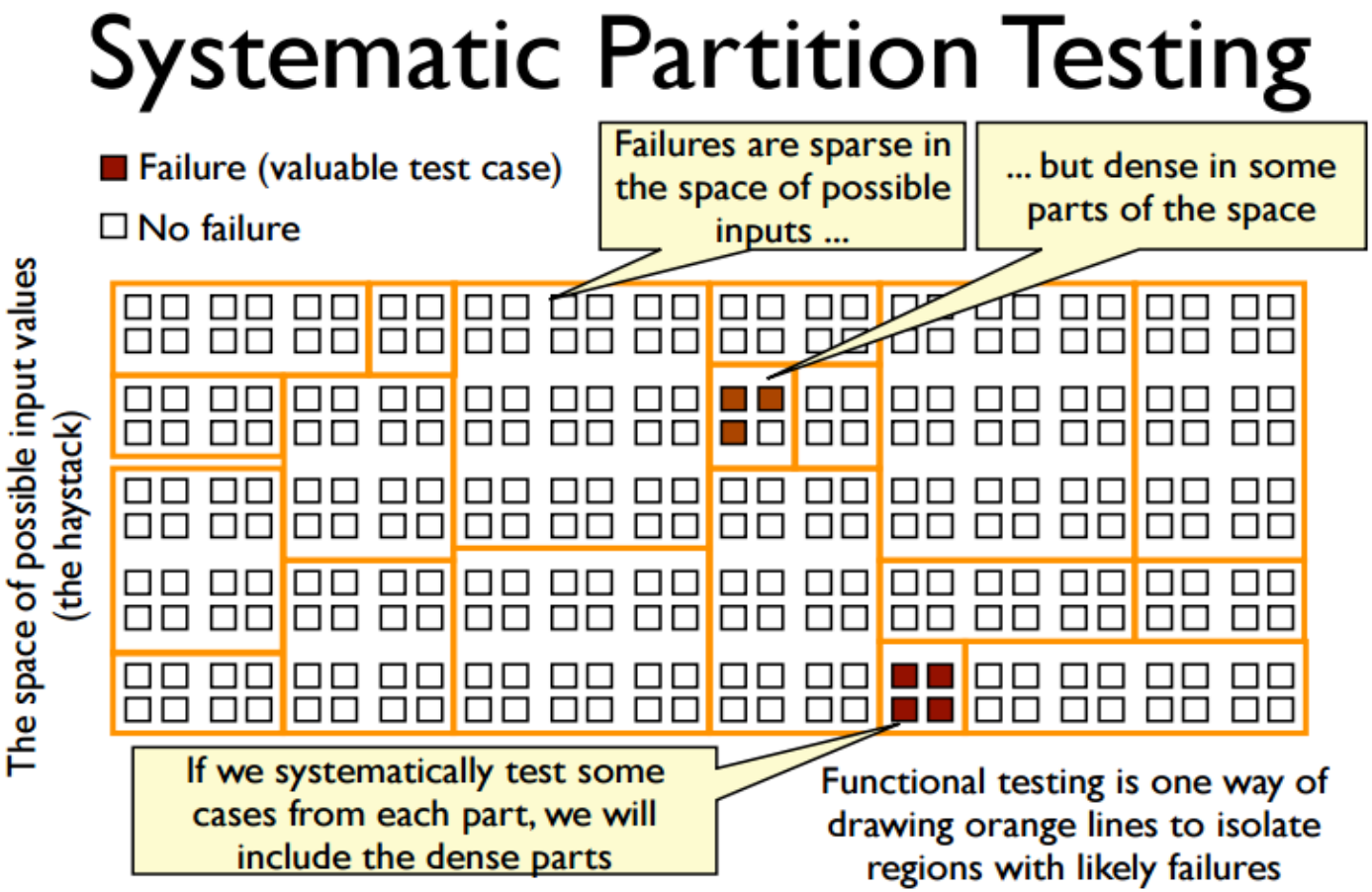
- Which values for *a, b, c* should we test?

  assuming a, b, c, were 32-bit integers, we'd have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

Equivalence Partitioning

- Typically the universe of all possible test cases is so large that you cannot try them all
- You have to select a relatively small number of test cases to actually run
- Which test cases should you choose?
- Equivalence partitioning helps answer this question

Equivalence Partitioning



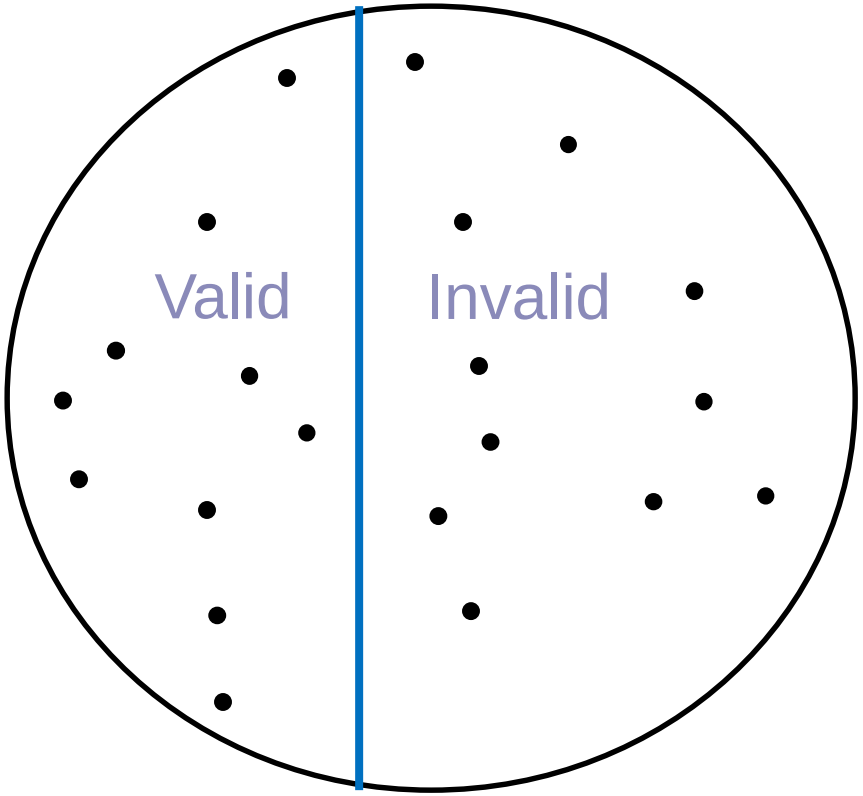
Source: “Software Testing and Analysis”, (Pezzè and Young), and Saarland University course on Software Testing (Fraser and Zeller)

Equivalence Partitioning

- Partition the test cases into "equivalence classes"
- Each equivalence class contains a set of "equivalent" test cases
- Two test cases are considered to be equivalent if we expect the program to process them both in the same way (i.e., follow the same path through the code)
- If you expect the program to process two test cases in the same way, only test one of them, thus reducing the number of test cases you have to run

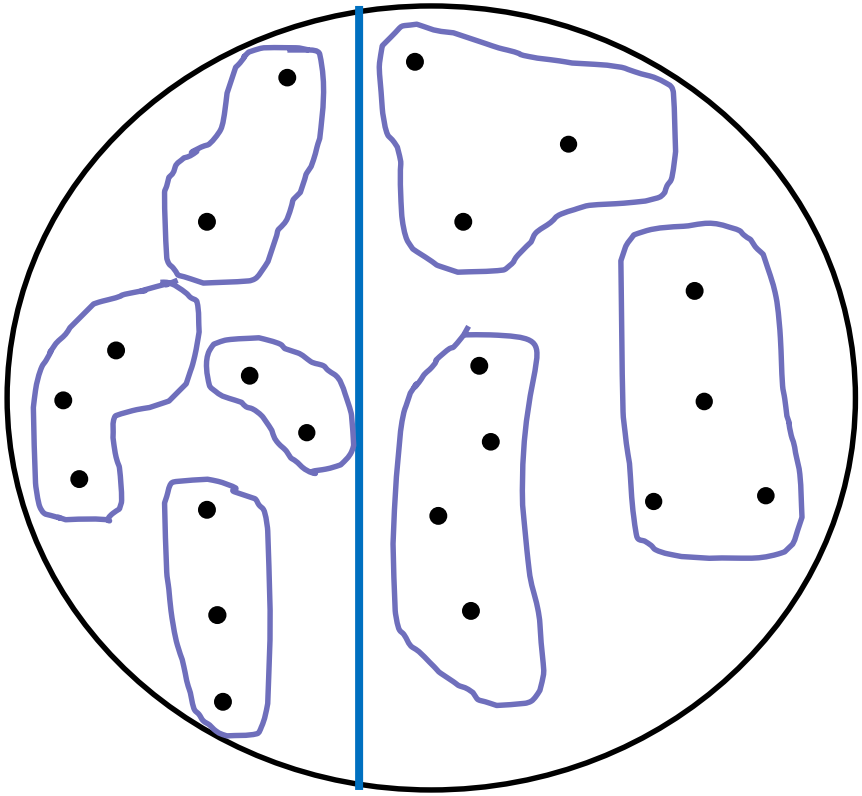
Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid test cases



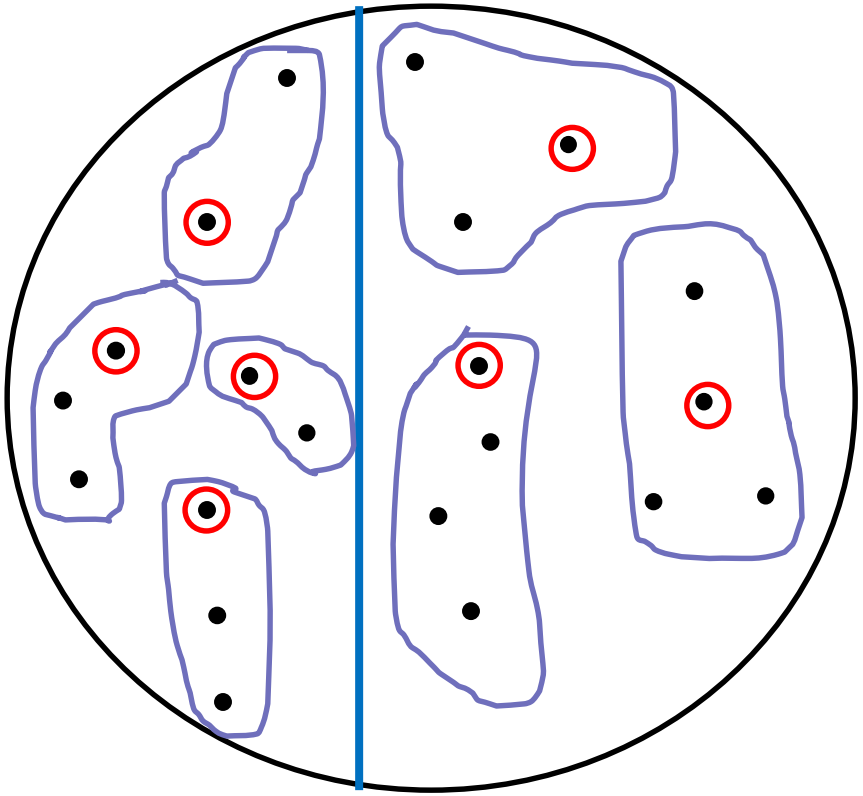
Equivalence Partitioning

- Partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



Equivalence Partitioning

- When designing test cases, you may use different definitions of “equivalence”, each of which will partition the test case space differently
  - Example: `int Add(n1, n2, n3, …)`
    - Equivalence Definition 1: partition test cases by the number of inputs (1, 2, 3, etc.)
    - Equivalence Definition 2: partition test cases by the number signs they contain (positive, negative, both)
    - Equivalence Definition 3: partition test cases by the magnitude of operands (large numbers, small numbers, both)
    - Etc.

Equivalence Partitioning

- If an oracle is available, the test values in each equivalence class can be randomly generated. This is more useful than always testing the same static values.
  - Oracle: something that can tell you whether a test passed or failed

- Test multiple values in each equivalence class. Often you're not sure if you have defined the equivalence classes correctly or completely, and testing multiple values in each class is more thorough than relying on a single value.

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	Invalid format 5555555, (555)(555)5555, etc. Area code < 200 or > 999 Area code with non-numeric characters *Similar for Prefix and Suffix*

Boundary Value Analysis

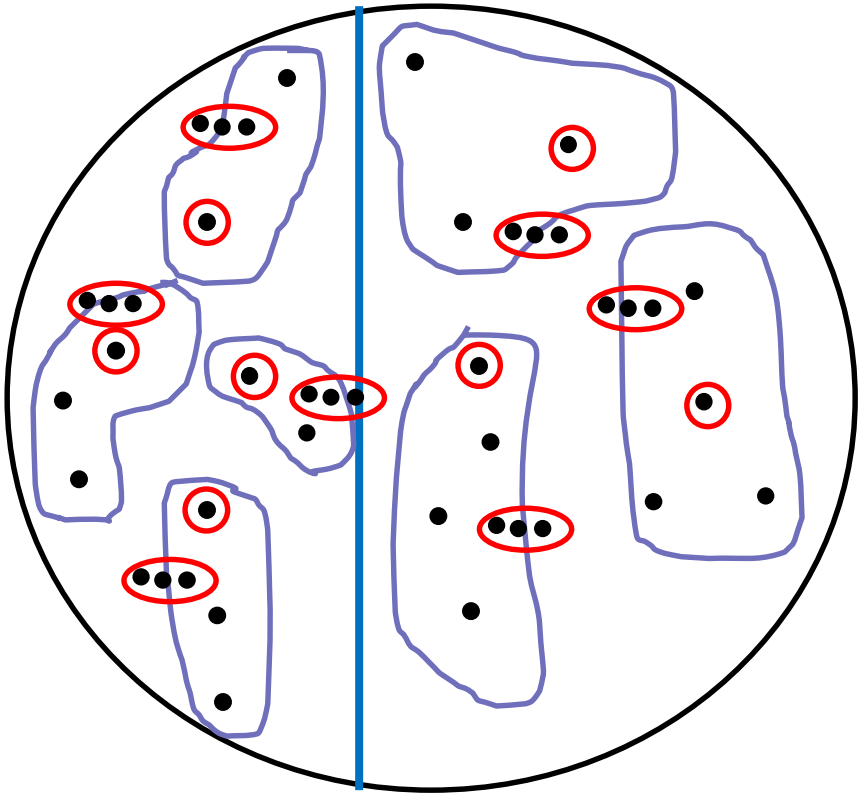
- When choosing values from an equivalence class to test, use the values that are most likely to cause the program to fail

- Errors tend to occur at the boundaries of equivalence classes rather than at the "center"
  - If (200 < areaCode && areaCode < 999) { // valid area code }
  - Wrong!
  - If (200 <= areaCode && areaCode <= 999) { // valid area code }
  - Testing area codes 200 and 999 would catch this error, but a center value like 770 would not

- In addition to testing center values, we should also test boundary values
  - Right on a boundary
  - Very close to a boundary on either side

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	Area code: 199, 200, 201 Area code: 998, 999, 1000 Prefix: 200, 199, 198 Prefix: 998, 999, 1000 Suffix: 3 digits, 5 digits

Mainstream usage testing

- Don't get so wrapped up in testing boundary cases that you neglect to test "normal" input values
  - Values that users would typically enter during mainstream usage

Ad Hoc Exploratory Testing (Error Guessing)

- Based on intuition, guess what kinds of inputs might cause the program to fail
- Create some test cases based on your guesses
- Intuition will often lead you toward boundary cases, but not always
- Some special cases aren't boundary values, but are mishandled by many programs
  - Try exiting the program while it's still starting up
  - Try loading a corrupted file
  - Try strange but legal URLs: hTtP://Www.CSus.EDU/

Comparison Testing

- Also called Back-to-Back testing
- If you have multiple implementations of the same functionality, you can run test inputs through both implementations, and compare the results for equality
- Why would you have access to multiple implementations?
  - Safety-critical systems sometimes use multiple, independent implementations of critical modules to ensure the accuracy of results
  - You might use a competitor's product, or an earlier version of your own, as the second implementation
  - You might write a software simulation of a new chip that serves as the specification to the hardware designers. After building the chip, you could compare the results computed by the chip hardware with the results computed by the software simulator
- Inputs may be randomly generated or designed manually