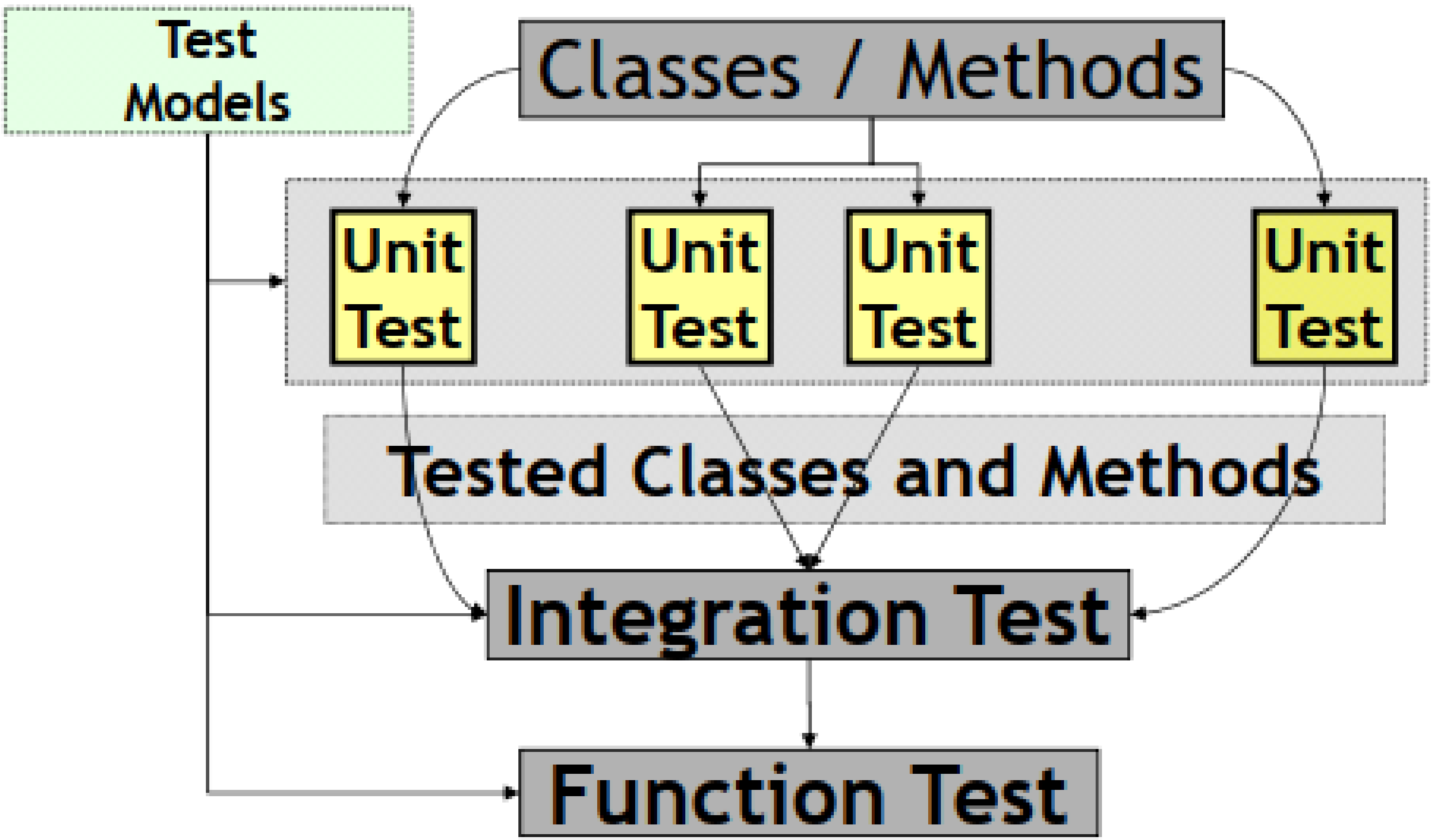
Software Engineering
(CSE 355)

Lecture 8

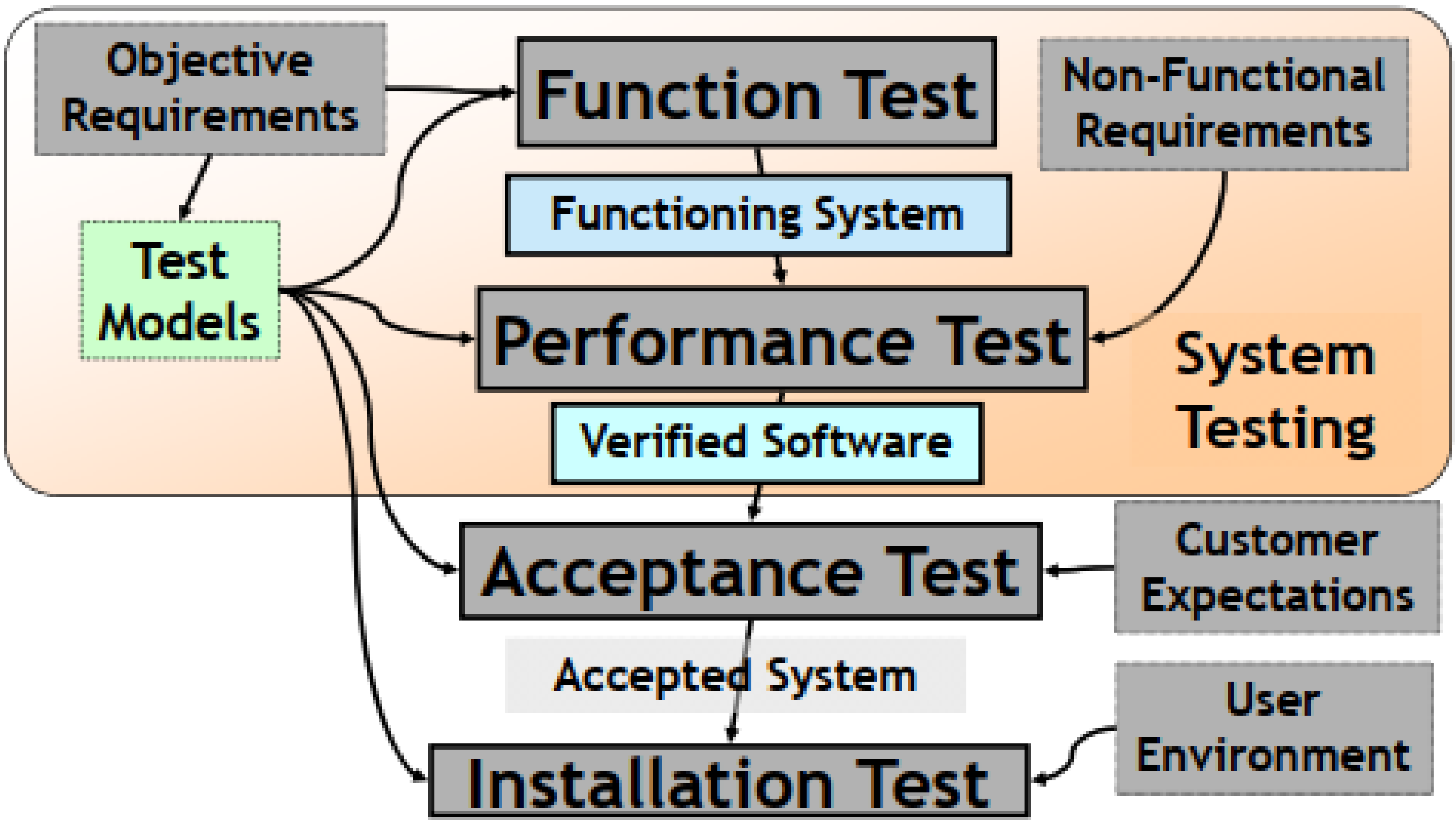
(Black Box Testing)

Unit Testing

Levels of Testing ...



Levels of Testing



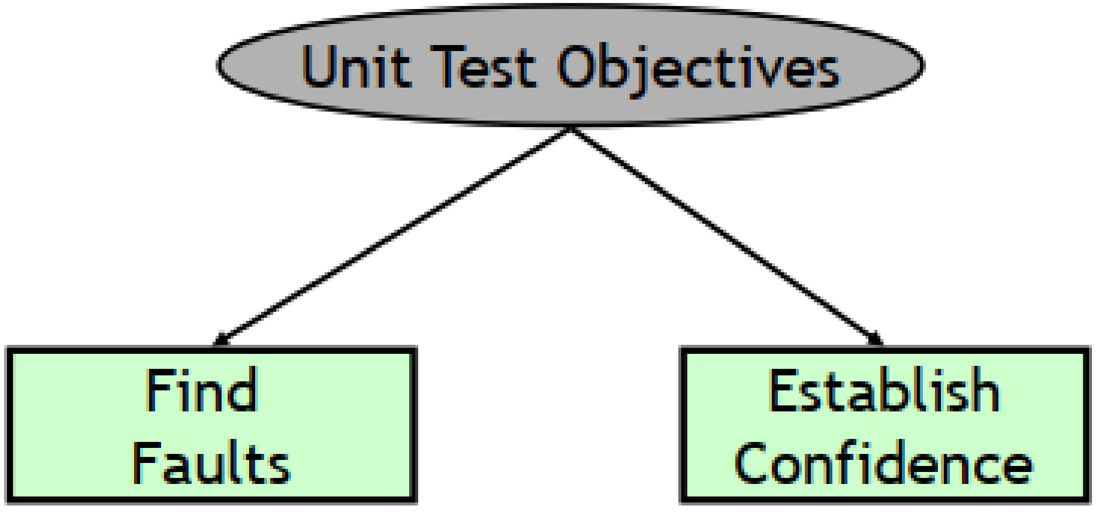
What is Unit Testing?

- Test effort is focused on the building blocks:
  - Methods
  - Classes
  - Components

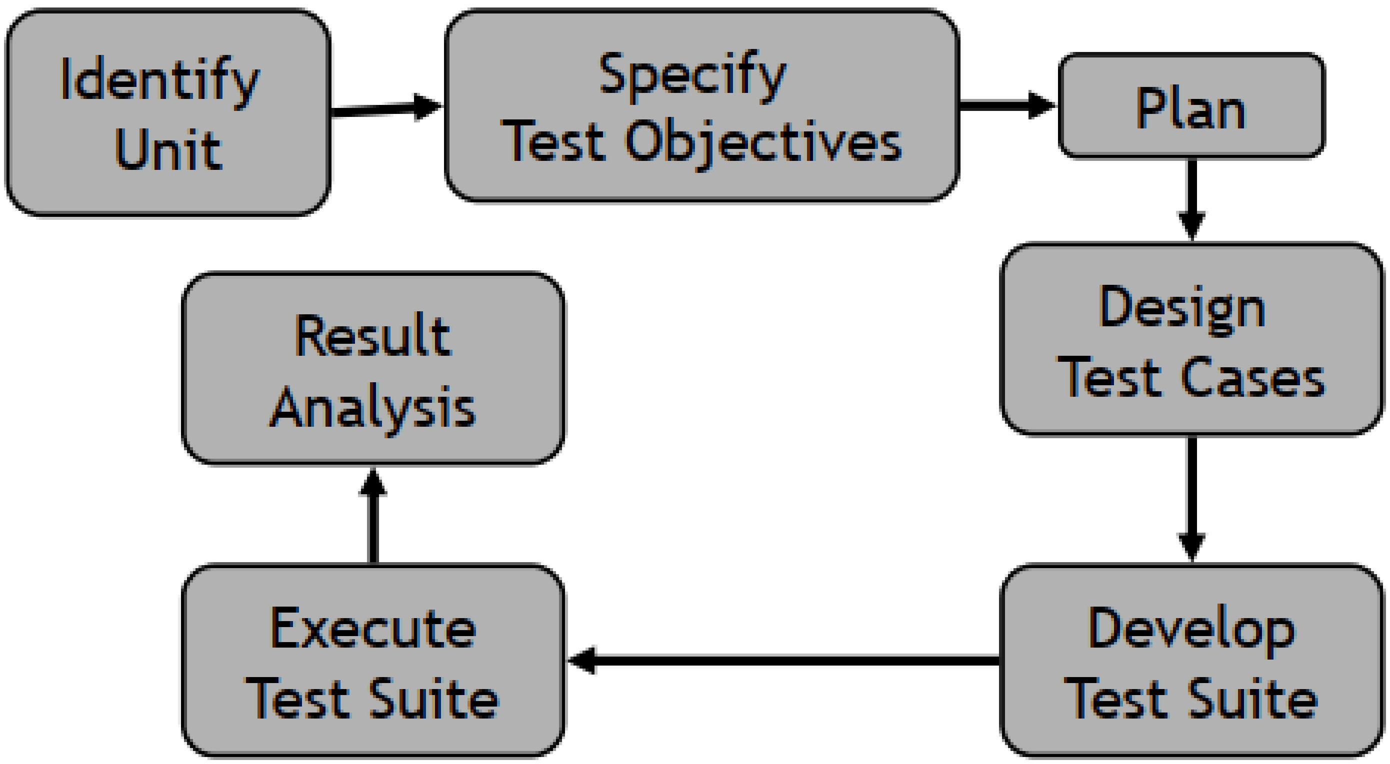
Why Unit Test?

- Reduces complexity
- Makes it easier to pinpoint and correct faults
- Allows parallelism in the testing activities

Unit Test Objectives



Unit Testing Process



Who is involved in Unit Testing?

- Ideal scenario is to have a separate testing team.
- Reality:
  - **Software programmers** often end up performing unit testing.
  - Only high-risk units are re-tested by an independent **Testing Team**.

Unit Identification ...

- We can use the following artefacts to help identify the unit for testing:
  - Functional Specifications
  - Use Case Diagrams
  - Class Diagrams
  - Sequence Diagrams
  - State Charts
  - Source Code
  - Code Metrics

Unit Identification ...

- **Specifications**: Can be used as a starting point to identify classes
- **Class Diagrams**: Help to identify a group of related classes fast
- **Component Diagrams**: Help by listing all the components in the system directly
- **Use Cases**: Can be used to identify classes implementing user interface
- **Sequence Diagrams**: Help by providing a list of clear collaborators for classes

Unit Identification

- **State Charts**: Help in designing test cases for an individual class
- **Source Code**: Source code can be reverse engineered and a detailed design can be generated. This can help refine the boundary of a unit better
- **Code Metrics**: Metrics can show complexity. This helps as we can ensure that highly complex classes are tested as units on their own

Examples of a Unit ...

- Utility Method
  - A method that calculates the Standard Deviation on an array of double's
- Method implementing an Algorithm
  - Quick Sort, Binary Search
- Data Structure Class
  - Stack, Linked List
- Component or Java Bean
  - Chart Bean, Transaction Class

Overview of This Lecture

- Test Case Design
  - White Box
    - Control Flow Graph
    - Cyclomatic Complexity
    - Basic Path Testing
  - Black Box
    - Equivalence Classes
    - Boundary Value Analysis

Test Case Design: Revision

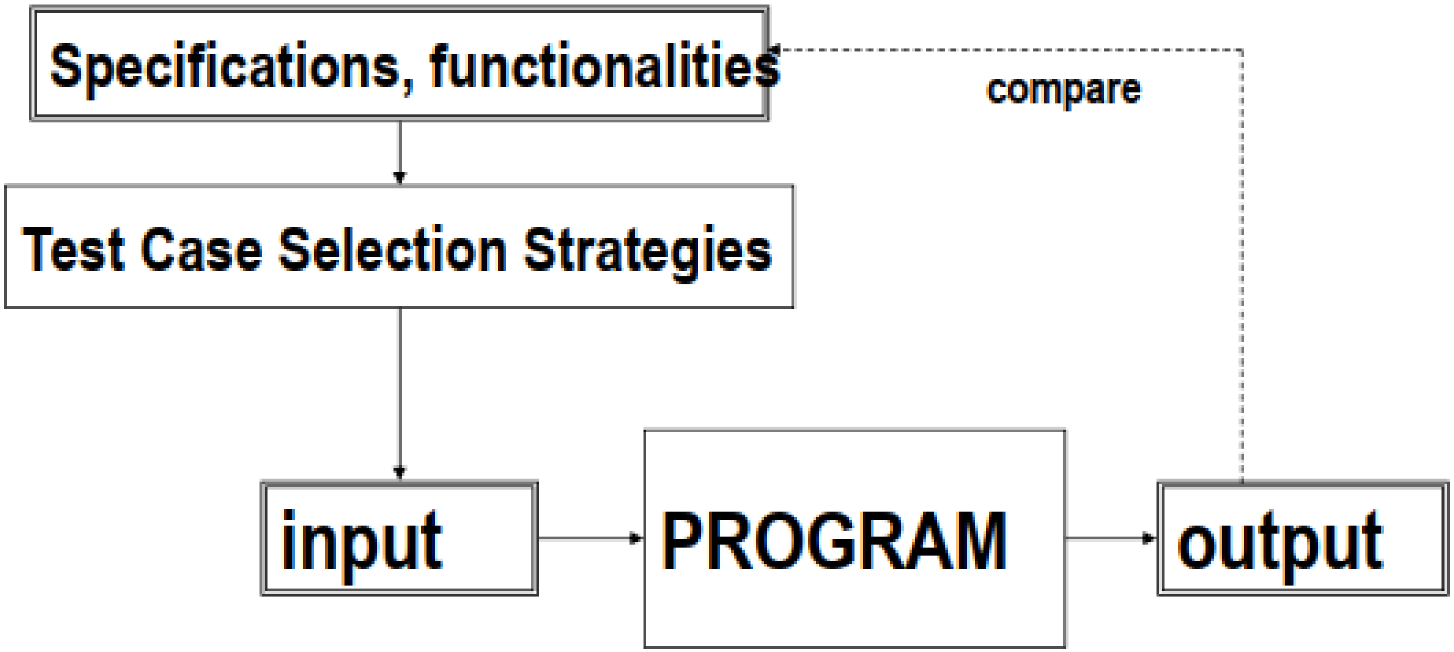
- White box testing:
  - Knowing the internal workings of a component;
  - Test cases exercise specific sets of condition, loops, etc.
- Black box testing:
  - Knowing the specified function a component has been designed for;
  - Tests conducted at the interface of the component.

Black Box Testing

Black Box Testing: Introduction

- Test Engineers have no access to the source code or documentation of internal working.
- The "Black Box" can be:
  - A single unit.
  - A subsystem.
  - The whole system.
- Tests are based on:
  - Specification of the "Black Box".
  - Providing **inputs** to the "Black Box" and inspect the **outputs.**

Black Box Testing – A common view



Test Case Design

- Two techniques will be covered for the black box testing in this course:
  - Equivalence Partition;
  - Boundary Value Analysis.

Equivalence Partition Testing

- Black-box testing technique
- It aims to minimise the number test cases
- All the possible inputs are partitioned into equivalence partition sets
- <u>One</u> test case is selected for each partition
- Assumption is that the program usually behaves in similar ways for <u>all</u> members of a partition set
- Consists of two (2) steps:
  - Identification of the equivalence partitions
  - Selection of the test inputs

Equivalence Partition ...

- If two tests give the same result – then we consider them as 'equivalent'
  - Example: Assume a method "isEven(int n)" returns true for all even numbers between 0 and 1000.
  - It should return true for all Even Inputs: 2,4,6..1000
  - It should return false for all Odd Inputs: 1,3,5..999
- For this example consider the following test:
  - Test 1: Input – 2, Expected Result - true
  - Test 2: Input – 7, Expected Result - false
  - Test 3: Input – 8, Expected Result - true
- Test 1 and 3 are considered 'Equivalent'

Equivalence Partition

- A set of tests form an Equivalence Partition if they meet the following criteria:
  - Coverage
  - Disjointedness
  - Representation
- Each of these is explored in detail now…

Equivalence Partitions - Coverage

- Every possible input belongs to one (and only one) of the partitions.
- The partitions cover the entire set of input data.
- If the Input Set is Split (for the isEven() Example) like:
  - Partition 1: Even Numbers between 0 – 100
  - Partition 2: Odd Numbers between 0 – 1000
  - Partition 3: Even Numbers between 101 – 1000
- Partition 1, 2 and 3 should cover all possible inputs.

Equivalence Partitions - Disjointedness

- No input belongs to more than one partition
- If the Input Set is split (for the isEven() example) like:
  - Partition 1: Even Numbers between 0 – 100
  - Partition 2: Odd Numbers between 0 – 1000
  - Partition 3: Even Numbers between 101 – 1000
- Partitions here are disjointed – there is no overlap

Equivalence Partitions - Representation

- If a data element belongs to a partition and reveals a fault, then the probability that every other element in that partition will reveal the same fault should be very high.
- The idea is that if one input from Partition 1 reveals an error, then a large number of inputs from this partition will also reveal this fault.
- Example: If the function is not working for number between 0 – 100, but works for any number greater than 100.

Equivalence Partition: Introduction

- To ensure the correct behavior of a “black box”, both valid and invalid cases need to be tested.
- Example:
  - Given the method below:

```
boolean isValidMonth(int m)

Functionality: check m is [1..12]
Output:
       - true if m is 1 to 12
       - false otherwise
```

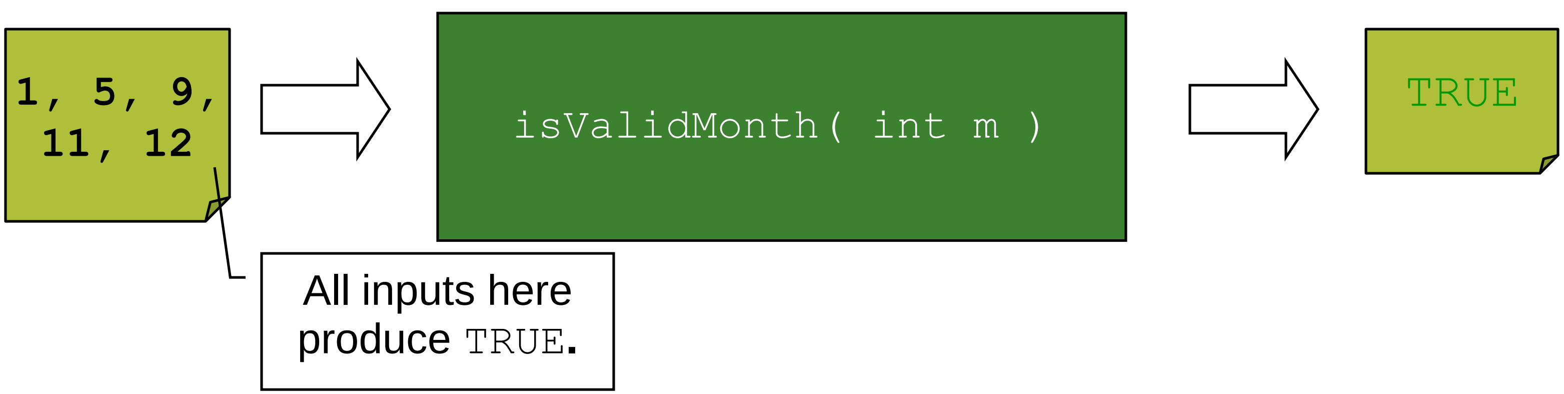
- Is there a better way to test other than testing **all** integer values $[-2^{31}, \ldots, 2^{31}-1]$?

Equivalence Partition

- Experience shows that exhaustive testing is not feasible or necessary:
  - Impractical for most methods.
  - An error in the code would have caused the same failure for many input values:
    - There is no reason why a value will be treated differently from others.
    - E.g., if value `240` fails the testing, it is *likely* that `241` is likely to fail the test too.
- A better way of choosing test cases is needed.

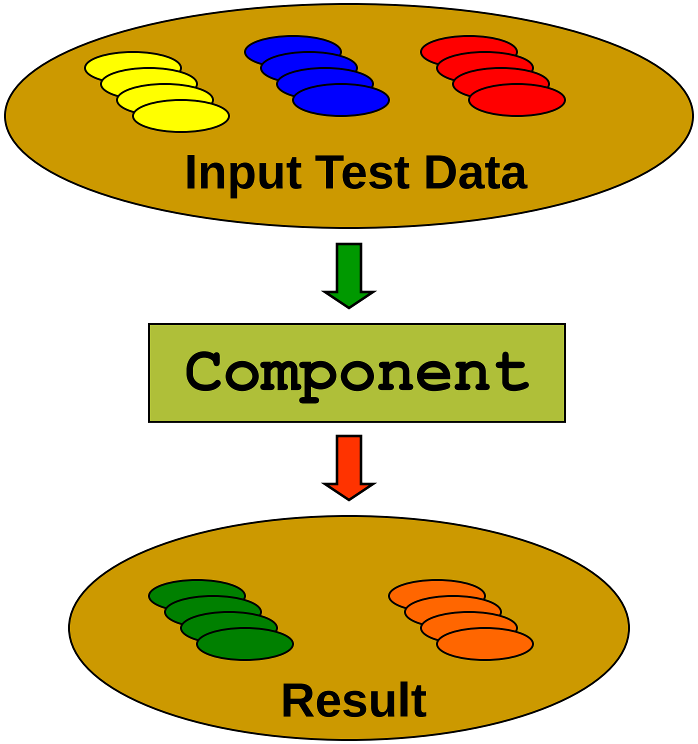
Equivalence Partition

- Observations:
  - For a method, it is common to have a number of inputs that produce similar outcomes.
  - Testing one of the inputs *should be* as good as exhaustively testing all of them.
  - So, pick only a few test cases from each “category” of input that produce the same output.



Equivalence Partition: Definition

- Partition input data into *equivalence classes.*
- Data in each equivalence class:
  - Likely to be treated equally by a reasonable algorithm.
  - Produce same output state, i.e., valid/invalid.
- Derive test data for each class.



Example (*isValidMonth*)

- For the *isValidMonth* example:
  - Input value `[1 … 12]` should get a similar treatment.
  - Input values lesser than `1`, larger than `12` are two other groups.
- Three partitions:
  - `[ -∞ ... 0 ]` should produce an **invalid** result
  - `[ 1 … 12 ]` should produce a **valid** result
  - `[ 13 … ∞ ]` should produce an **invalid** result
- Pick one value from each partition as test case:
  - E.g., `{-12, 5, 15}`
  - Reduce the number of test cases significantly.

Common Partitions

- If the component specifies an input range, [X … Y].
  - Three partitions:

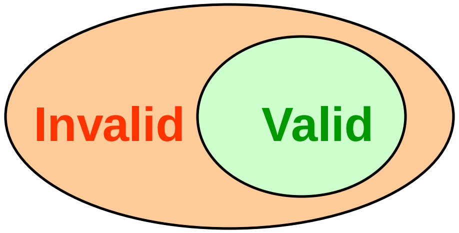
-∞ ... <X	X ... Y	>Y … ∞
Invalid	**Valid**	**Invalid**

- If the component specifies a single value, [X].
  - Three partitions:

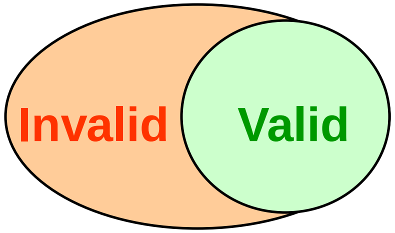
-∞ ... <X	X	>X … ∞
Invalid	**Valid**	**Invalid**

Common Partitions

- If the component specifies member(s) of a set:
  - E.g., The traffic light color = {Green, Yellow, Red}; Vehicles have to stop moving when the traffic light is Red.
  - Two partitions:



- If the component specifies a boolean value.
  - Two partitions:



Combination of Equivalence Classes

- Number of test cases can grow very large when there are multiple parameters.
- Example:
  - Check a phone number with the following format:
    - (XXX) XXX – XXXX

Area Code (optional)	Prefix	Suffix

  - In addition:
    - Area Core Present: Boolean value [`True` or `False`]
    - Area Code: 3-digit number [`200 … 999`] except `911`
    - Prefix: 3-digit number not beginning with `0` or `1`
    - Suffix: 4-digit number

Example (cont)

- Area Core Present: Boolean value [`True` or `False`]
  - Two Classes:
    - [True], [False]
- Area Code: 3-digit number [`200 ... 999`] except `911`
  - Five Classes:
    - [-∞ …199], [200 … 910], [911], [912 … 999], [1000 … ∞]
- Prefix: 3-digit number not beginning with `0` or `1`
  - Three Classes:
    - [-∞ … 199], [200 … 999], [1000 … ∞]
- Suffix: 4-digit number
  - Three Classes:
    - [-∞ … -1], [0000 … 9999], [10000 … ∞]

Example (cont)

- A thorough testing would require us to test all combinations of the equivalence classes for each parameter.
- Hence, the total equivalence classes for the example should be the *multiplication* of equivalence classes for each input:
  - 2 x 5 x 3 x 3 = 90 classes = 90 test cases (!)
- For critical systems, all combinations should be tested to ensure a correct behavior.
- A less stringent testing strategy can be used for normal systems.

Reduction of Test Cases

- A reasonable approach to reduce the test cases is as follows:
  - At least one test for each equivalence class for each parameter:
    - Different equivalence class should be chosen for each parameter in each test to minimize the number of cases.
  - Test all combinations (if possible) where a parameter may affect each other.
  - A few other random combinations.

Example

- As the Area Code has the most classes (i.e., 5), five test cases can be defined which simultaneously try out difference equivalence classes for each parameter:

Test Case	Area Code Present	Area Code	Prefix	Suffix
1	False	[-∞ …199]	[-∞ … 199]	[-∞ … -1]
2	True	[200 … 910]	[200 … 999]	[0000 … 9999]
3	True	[911]	[1000 … ∞]	[10000 … ∞]
4	True	[912 … 999]	[200 … 999]	[0000 … 9999]
5	True	[1000 … ∞]	[1000 … ∞]	[10000 … ∞]

E.g., Actual Test Case Data:
`(True, 934, 222, 4321)`

Example (cont)

- In this example, Area Code Present affects the interpretation of Area Code, so all combinations between these two parameters should be tested.
  - 2 x 5 = 10 test cases.
  - Five combinations have already been tested in the previous test cases, five more would be needed.
- Not counting extra random combinations, this strategy reduces the number of test cases to only 10.

Equivalence Partition Testing

- For each equivalence class, at least two pieces of data are selected:
  - A typical input – This exercises the common case.
  - An invalid input – to exercise the exception handling capabilities of the component.
- Now the input value selection has to be made:
  - If there is a possibility that the inputs do not really represent all possible elements of the equivalence partition – then the partition must be split into smaller sets.

Equivalence Partition Example ...

public static int getNumDaysInMonth(int month, int year).

// Assume month starts at 1 (Jan) and ends with 12 (Dec).

// Assume that the year can range between 0 and maximum integer value (2^{32}).

- Possible partitions for the Month are,
  - Months with 31 days.
  - Months with 30 days.
  - February with 28 days.
  - February with 29 days.
- Possible partitions for the Year are,
  - Leap Years.
  - Non-Leap Years.

Equivalence Partition Example ...

- The Business Rules for Calculating a Leap Year are:
  - All Years that are multiples of 4 (1980,1984)
  - Exception: Years that are multiples of 100 are not leap years, unless they are also multiples of 400.
    - 1900 is not a leap year, but 2000 is

Equivalence Partition Example ...

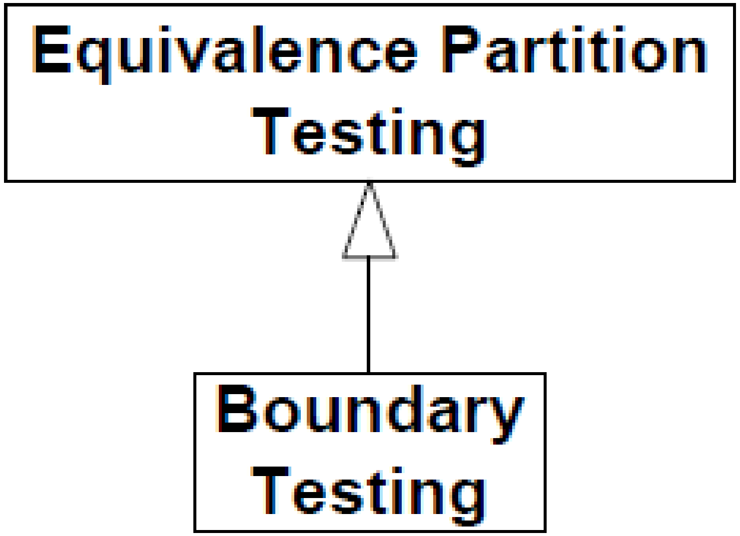
Equivalence Partition	Input Values	
	Month	Year
Months with 31 days, Non-Leap Years	**7 (July)**	**1901**
Months with 31 days, Leap Years	**7 (July)**	**1904**
Months with 30 days, Non-Leap Years	**4 (April)**	**1902**
Months with 30 days, Leap Years	**6 (June)**	**1908**

Equivalence Partition Example

Equivalence Partition	Input Values	
	Month	Year
February (Non-Leap Year)	**2 (Feb)**	**1900**
February (Leap Year)	**2 (Feb)**	**2004**

Boundary Testing ...

- Special case of equivalence testing
- Boundary testing focuses on the edge of each Equivalence Partition



Boundary Testing ...

- Developers often overlook special cases at the boundary of the equivalence partition sets (eg. Zero, -1, Empty Strings, Year 2000)
  - Boundary Testing is designed to catch these errors.
- Also known as “Boundary Value Analysis” in some books

Boundary Value Analysis: Introduction

- It has been found that most errors are caught at the **boundary** of the equivalence classes.
- Not surprising, as the end points of the boundary are usually used in the code for checking:
  - E.g., checking `K` is in range [`X` … `Y`):

```
if (K >= X && K <= Y)
...
```

Easy to make mistake on the comparison.

- Hence, when choosing test data using equivalence classes, boundary values should be used.
- Nicely complement the Equivalence Class Testing.

Using Boundary Value Analysis

- If the component specifies a range, [`X … Y`]
  - Four values should be tested:
    - Valid: `X` and `Y`
    - Invalid: Just below `X` (e.g., `X - 1`)
    - Invalid: Just above `Y` (e.g., `Y + 1`)
  - E.g., [`1 … 12`]
    - Test Data: `{0, 1, 12, 13}`
- Similar for open interval (`X … Y`), i.e., `X` and `Y` not inclusive.
  - Four values should be tested:
    - Invalid: `X` and `Y`
    - Valid: Just **above** `X` (e.g., `X + 1`)
    - Valid: Just **below** `Y` (e.g., `Y - 1`)
  - E.g., (`100 … 200`)
    - Test Data: `{100, 101, 199, 200}`

Using Boundary Value Analysis

- If the component specifies a number of values:
  - Define test data using [`min` value … `max` value]:
    - Valid: `min, max`
    - Invalid: Just below `min` (e.g., `min - 1`)
    - Invalid: Just above `max` (e.g., `max + 1`)
  - E.g., `values = {2, 4, 6, 8} → [2 … 8]`
    - Test Data: `{1, 2, 8, 9}`
- If a data structure has prescribed boundaries:
  - define test data to exercise the data structure at those boundaries.
  - E.g.,
    - String: Empty String, String with 1 character
    - Array : Empty Array, Array with 1 element, Full Array
- Boundary value analysis is not applicable for data with no meaningful boundary, e.g., the set *color* `{Red, Green, Yellow}`.

Boundary Testing Example

Equivalence Partition	Input Values	
	Month	Year
Years divisible by 400	**2 (Feb)**	**2000**
Non-Leap Years divisible by 100, but not by 400	**2 (Feb)**	**1900**
Non-Positive Invalid Months	**0**	**1902**
Positive Invalid Months	**13**	**1315**

Example

- Apply the Black Box Equivalence Partitioning test selection technique to the following question:

(Q1) "Write a small fragment of code that will find the largest number in an array of integers"

- You will need to supply as many tests as there are equivalence partitions.

Sample Answer of Q1

Purpose: Equivalence Partition	Input(s)	Environment	Expected output(s)
Array of size greater than 1: all same number	1, 1, 1, 1, 1	Array of size 5	1
All positive, different and in order	1, 2, 3, 4, 5	Array of size 5	5
All positive, different and descending order	5, 4, 3, 2, 1	Array of size 5	5
All positive, different and in random order	1, 2, 5, 4, 3	Array of size 5	5
All negative and different	-1, -2, -3, -4, -5	Array of size 5	-1
All 0	0, 0, 0	Array of size 3	0
Array of size 1	5	Array of size 1	5
Array of size 0		Array of size 0?	Error?
Non-numbers	1, t, 5, h	Array of size 4	Error

Functionality Testing

- Previous examples have mostly numerical parameters and simplistic functionality, where it is easy to see how Equivalence Class Testing and Boundary Value Analysis can be applied.
- The following example is to illustrate how functionality testing of a method can be accomplished by the black box testing techniques discussed.
- This requires the method to be well specified:
  - The Precondition, Postcondition and Invariant should be available.
  - The Invariant: A property that is preserved by the method, i.e., `true` before and after the execution of method.

Functionality Testing

- For a well specified method (or component).
  - Use the Precondition:
    - Define Equivalence Classes.
    - Apply Boundary Value Analysis if possible to choose test data from the equivalence classes.
  - Use the Postcondition and the Invariant:
    - Derive expected results.

Example (Searching)

```
boolean Search(
     List aList, int key)
```

Precondition:

- `aList` has at least one element

Postcondition:

- `true` if `key` is in the `aList`
- `false` if `key` is not in `aList`

Equivalence Classes

- Sequence with a single value:
  - key found.
  - key not found.
- Sequence of multi values:
  - key found:
    - First element in sequence.
    - Last element in sequence.
    - “Middle” element in sequence.
  - key not found.

Test Data

Test Case	aList	Key	Expected Result
1	[123]	123	True
2	[123]	456	False
3	[1, 6, 3, -4, 5]	1	True
4	[1, 6, 3, -4, 5]	5	True
5	[1, 6, 3, -4, 5]	3	True
6	[1, 6, 3, -4, 5]	123	False

Example (Stack – Push Method)

```
void push (Object obj) throws FullStackException
```

Precondition:

```
        - ! full()
```

Postconditions:

```
        - if !full() on entry  then
                top() == obj && size() == old size() + 1
          else throw FullStackException
```

Invariant:

```
        - size() >= 0 && size() <= capacity()
```

- Common methods in the `Stack` class:
  - `full(), top(), size(), capacity()`

Test Data

- Precondition: stack is not full (i.e., boolean).
  - Two equivalence classes can be defined.
  - Valid Case: `Stack` is not full.
    - Input: a non-full stack, an object `obj`
    - Expected result:

```
top() == obj
size() == old size() + 1
0 <= size() <= capacity()
```

  - Invalid Case: `Stack` is full.
    - Input: a full stack, an object `obj`
    - Expected result:

`FullStackException` is thrown

```
0 <= size() <= capacity()
```