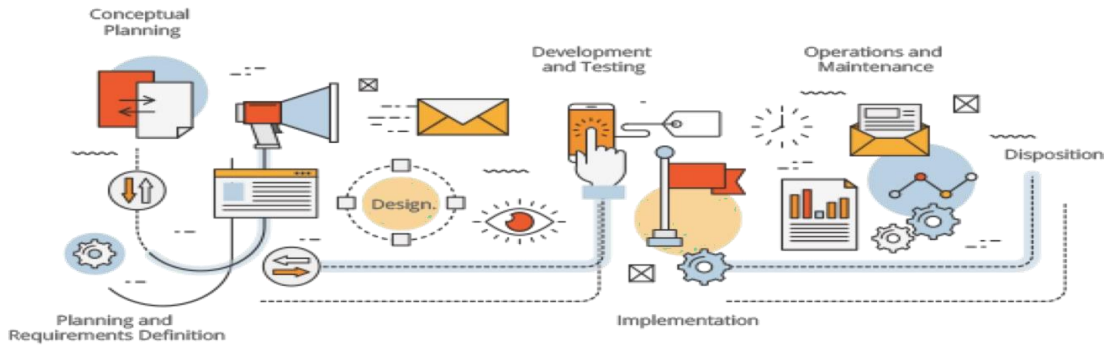
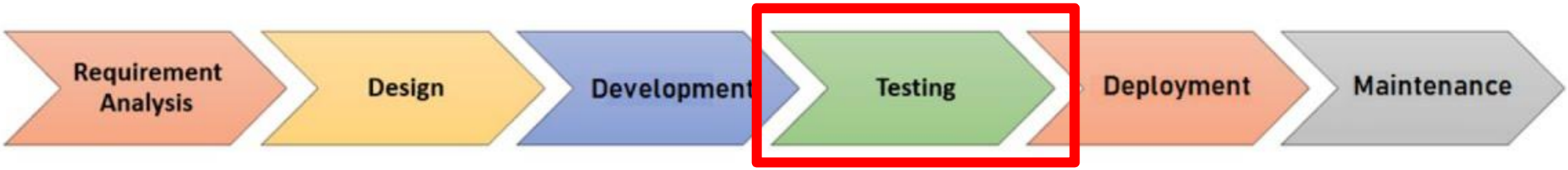
Software Engineering

Requirement Engineering



Software Development Life Cycle (SDLC)



Testing Objectives

What is being tested and why?

Testing is a method used to determine whether each of the project's software deliverables meet the defined quality standards established for the software development project.

Goal is production-ready software.

Testing

- There is a massive misunderstanding about testing: that it improves software. It doesn't!
  - Weighing yourself doesn't reduce your weight
  - Going to the do
- Those things he choose to resolve. Testing does toc
- The testing does not make the product better, even though it's part of a process that does make a product better.

Similarly, testing software doesn't magically make it better. It only tells you what's wrong so you can fix it. Just like stepping on a scale won't make you lose weight—it only shows whether you need to!

Before Writing Test Cases, Consider These Factors

1 Understand the Language Fundamentals

- Know the **sizes and limits of variables** (e.g., int, float, char).
- Be aware of **platform-specific differences** (e.g., 32-bit vs. 64-bit, endianness).
- Consider **how data types affect computation** (e.g., precision loss in floating points, integer overflow).

2 Understand the Domain

- Read and analyze the **requirements** carefully.
- **Think like a user**: What do they want to achieve? What might they accidentally do?
- Identify **edge cases**, such as:
  - Invalid input (wrong formats, missing values).
  - Impossible conditions (e.g., negative age, exceeding system limits).

cont..

3 Define the Purpose of Testing

- **Correct Operation**: Ensures correct output for valid input.
- **Robustness**: Handles incorrect input gracefully (without crashes).
- **User Acceptance**: Reflects real-world user behavior and expectations.

4 Write Down the Test Cases

- Cover **positive** scenarios (expected behavior).
- Include **negative** cases (invalid input, boundary values).
- Test **edge cases** and **performance limits**.
- Ensure all **functional and non-functional** requirements are tested.

Writing Good Test Cases

- Test Cases need to be simple and transparent
- Create Test Case with end user in mind
- Avoid test case repetition
- Do not Assume
  - Stick to the Specification Documents.
- Ensure 100% Coverage
- Test Cases must be identifiable.
- Implement Testing Techniques
  - It's not possible to check every possible condition in your software application
  - Testing techniques help you select a few test cases with the maximum possibility of finding a defect

Writing Good Test Cases

- Boundary Value Analysis (BVA)
  - testing of boundaries for specified range of values.

- Repeatable and self-standing
  - The test case should generate the same results every time no matter who tests it

Writing a test case

While drafting a test case do include the following information:

- The description of what requirement is being tested
- Inputs and outputs or actions and expected results
  - Test case must have an expected result.

- Verify the results are correct
  - ü Testing Normal Conditions (in case of calculator Test Case: Enter 5 + 3, expect 8.)
  - ü Testing Unexpected Conditions (Upload an empty file, expect error message: "File cannot be empty".)
  - ü Bad (Illegal) Input Values (Enter "123456" as a password when it requires letters, expect error: "Password must contain letters and numbers".)
  - ü Boundary Conditions (Example: A bank system should not allow withdrawals below $10 or above $10,000.)
    1. ◈ Test Case: Withdraw $10, expect success.
    2. ◈ Test Case: Withdraw $9, expect error: "Minimum withdrawal is $10".
    3. ◈ Test Case: Withdraw $10,001, expect error: "Maximum withdrawal is $10,000".

Writing test cases

- Beware of problems with comparisons
  - How to compare two floating numbers
    - Never do the following:
      float a, b;
      . .
      if (a == b)
    - Is it 4.0000000 or 3.9999999 or 4.0000001 ?
    - What is your limit of accuracy?

Why Can't We Use a == b?
Computers store floating-point numbers with limited precision, so exact values may not be stored correctly.
4.0 may actually be stored as 3.9999999 or 4.0000001 due to rounding errors in floating-point arithmetic.
Direct comparison (a == b) may fail even if the numbers are "visually" the same.

- In object oriented languages make sure whether you are comparing the contents of an object or the reference to an object

String a = "Hello world!\n"
String b = "Hello world!\n"
if (a == b)
vs.
if (a.equals(b))

a == b (Reference Comparison)
This checks if both variables point to the same memory location.
If a and b refer to the exact same object in memory, == returns true.
In this example, a == b is true because Java stores string literals in a "string pool", and identical literals point to the same memory location.
However, if the strings were created dynamically (e.g., using new String("Hello world!\n")), == would return false.

Verification vs validation

- **Verification**:
  - "Are we building the product right".
  - The software should conform to its specification.
  - Defect detection and correction
  - Comparison between implementation and the corresponding specification

- **Validation**:
  - "Are we building the right product".
  - The software should do what the user really requires.
  - Defect prevention
  - SW is traceable to customer requirements

Example

Example of verification in the calculator application:

- **Requirement**: The calculator application should correctly add two numbers.
- **Verification**: You review the code for the addition operation and ensure that it correctly adds two numbers by performing unit tests. You verify that the addition function returns the correct sum for different input values, such as 2 + 2 = 4, 5 + 3 = 8, etc. This process confirms that the implemented code behaves as expected according to the specified requirement.

Example of validation in the calculator application:

- **User Need**: Users expect the calculator application to accurately perform mathematical calculations and provide correct results.
- **Validation**: You conduct user acceptance testing (UAT) with a group of users who perform various calculations using the application. Users input different sets of numbers and verify that the calculator produces accurate results for addition, subtraction, multiplication, and division operations. The feedback from users confirms that the calculator application meets their needs and performs as expected in real-world usage scenarios.

Types of correctness

The terms probable correctness, possible correctness, and absolute correctness relate to the confidence level in the validity of a statement, argument, or solution, particularly in logic, mathematics, and computer science.

Three Levels of Correctness

- **Possible correctness**

- Possible correctness means that *a program can be correct under certain conditions*, but this does not guarantee it will work correctly in all cases.

- It implies that the software has the potential to be correct if the right conditions hold, such as correct inputs, expected environments, and no unexpected interactions.

- Example: "It is possible that it will rain tomorrow." (There is some chance, but no certainty.)

Three Levels of Correctness

Probable correctness

- Probable correctness means confidence, not certainty.
- More testing increases confidence but never proves absolute correctness.
- This involves statistical or probabilistic reasoning, suggesting that correctness holds in most cases.
- Example: "Based on historical data, it is probable that it will rain tomorrow."

Three Levels of Correctness

Absolute correctness

- Absolute correctness means that a program is proven to be 100% correct for all possible inputs and conditions. It guarantees that the software will always produce the expected output without failures, given the specifications.

- However, in real-world software development, achieving absolute correctness is nearly impossible due to complexity, external dependencies, and unpredictable conditions.

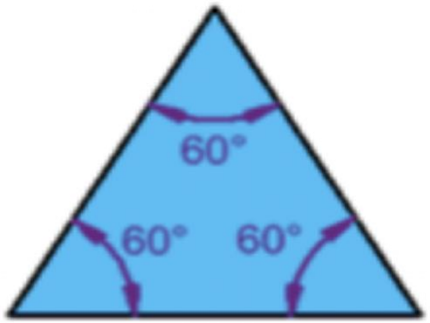
A Small Testing Example

- You are to develop a small program that reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle.
- The program displays a message that states whether the triangle is scalene, isosceles, or equilateral. Write a set of test cases—specific sets of data—to properly test the program
  - The set of test data must be handled by the program correctly to be considered a successful program.

Remember that

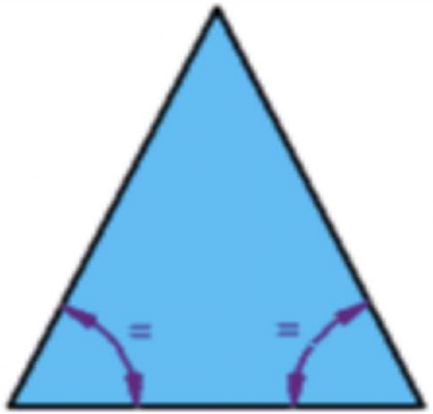
- a scalene triangle is one where no two sides are equal,
- whereas an isosceles triangle has two equal sides, and
- an equilateral triangle has three sides of equal length.

Definitions



Equilateral Triangle

Three equal sides
Three equal angles, always 60°



Isosceles Triangle

Two equal sides
Two equal angles

Scalene Triangle

No equal sides
No equal angles

```
int main()
{
    float a,b,c;
    cin>>a >> b >>c;
    if(a<(b+c) && b < (a+c) && c< (a+b))
    {
        cout<<"\nIt is a Triangle.";
        if (a==b && b==c)
            cout<<"\nIt is a Equilateral Triangle.";
        if(a==b || a==c || b==c)
            cout<<"\nIt is a Isosceles Triangle.";
        else
            cout<<"\nIt is a Scalene Triangle.";
    }
    else
        cout<<"This Triangle is not possible.";

        return 0;
     }
```

WHAT TEST CASES DO YOU THINK SHOULD BE THERE?

For the input values to represent a triangle, they must be integers greater than 0 where the sum of any two is greater than the third.

Questions to Answer ??

1. Do you have a test case that represents a **valid scalene triangle**
2. Do you have a test case that represents a **valid equilateral triangle?**
3. Do you have a test case that represents a **valid isosceles triangle?**
4. Do you have **at least three test cases that represent valid isosceles triangles** such that you have tried all three permutations of two equal sides (such as, 3, 3, 4; 3, 4, 3; and 4, 3, 3)?
5. Do you have a test case in which **one side has a zero value**?
6. Do you have a test case in which **one side has a negative value**?
7. Do you have a test case with **three integers greater than zero such that the sum of two of the numbers is equal to the third**? (That is, if the program said that 1, 2, 3 represents a scalene triangle, it would contain a bug.)

Questions to Answer ?? (2)

8 Do you have **at least three test cases in category 7** such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (for example, 1, 2, 3; 1, 3, 2; and 3, 1, 2)?

9 Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (such as 1, 2, 4 or 12,15,30)?

10 Do you have at least three test cases in category 9 such that you have tried all three permutations (for example, 1, 2,4; 1, 4, 2; and 4, 1, 2)?

11 Do you have a test case **in which all sides are zero** (0, 0, 0)?

12 Do you have at least **one test case specifying noninteger values** (such as 2.5, 3.5, 5.5)?

13 Do you have at least one test case **specifying the wrong number of values** (two rather than three integers, for example)?

14 For each test case did you specify the expected output from the program in addition to the input values?

Testing the triangle program – Some thoughts!

A very simple program

To do exhaustive testing

- Test with all possible values of a, b, & c (the three sides)
  - And their combinations
  - 32-bit integer, 2^{32} possible values for one integer
  - Assuming only integers from 1 to 10, there are 10^{12} possible values for a triangle.
    - Testing 1000 cases per second, you would need 317 years!
- Test with all the invalid inputs
  - e.g., All strings ;)

A 32-bit integer can represent 2^{32} different values. In decimal terms, that's:

$$2^{32} = 4,294,967,296$$

So, there are $4,294,967,296$ possible values for a 32-bit integer. These values range from 0 to $2^{32} - 1$, inclusive, because integers are typically represented using binary notation where each bit represents a power of 2, starting from $2^0 = 1$ up to 2^{31}. The maximum value that can be represented by a 32-bit integer is $2^{32} - 1$, which is $4,294,967,295$.

Testing the triangle program – Some thoughts!

- If testing a trivial program is so complex, what about testing large programs of thousands of lines of code, e.g., air traffic control software
- Solution?

Testing the triangle program – Some thoughts!

- If testing a trivial program is so complex, what about testing large programs of thousands of lines of code, e.g., air traffic control software

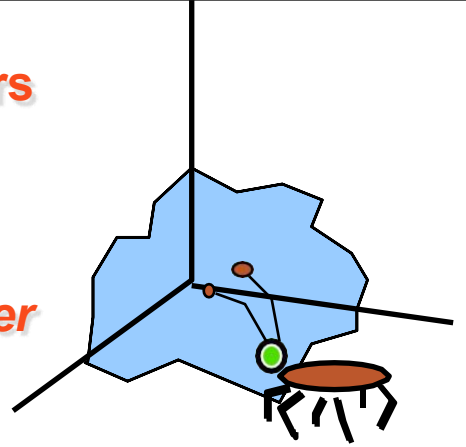
Solution?

- Use of sophisticated testing techniques
  - Select the test cases that have most chances of triggering a failure
  - e.g, boundary values

Test case design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



OBJECTIVE **---- to uncover errors**

CRITERIA **----- In a complete manner**

CONSTRAINT **------ with a minimum of effort and time**

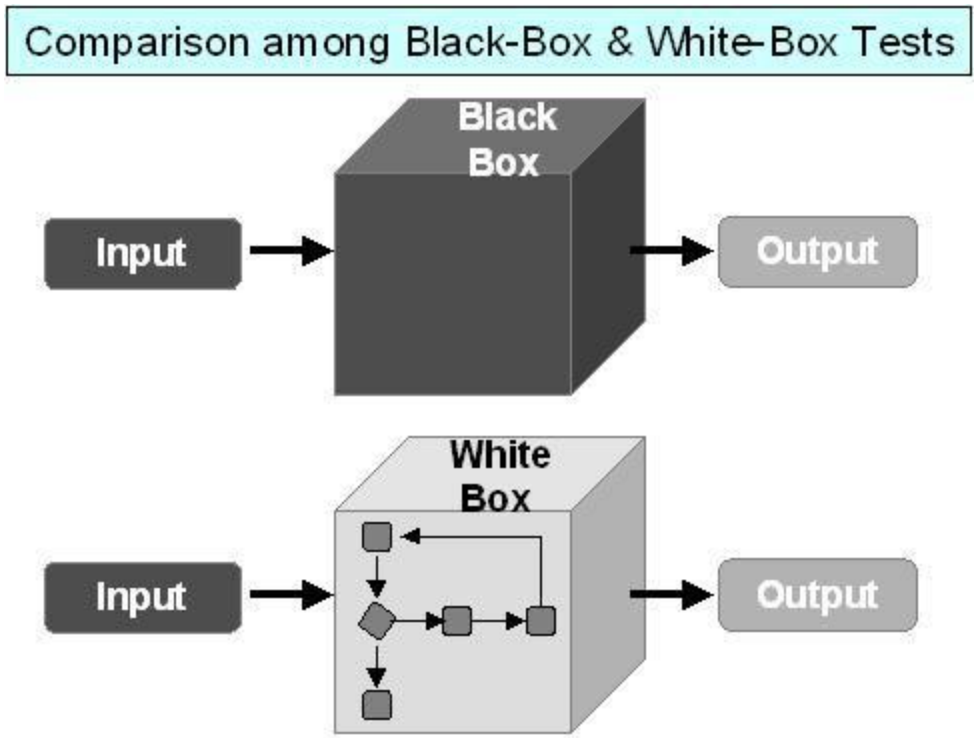
- A successful test -- -- that uncovers an as-yet undiscovered error.
- Minimum number of required tests with 100% functional coverage and 0% redundancy.
- Rich variety of test case design methods
  - - Cause-effect graphing, Equivalence Class Partitioning, Boundary Analysis, and vendor specific: client/server, OO test case design.

Possible approaches:

Black Box Testing: Tests the software based on its external behavior and functionality without considering its internal structure or code. It focuses on validating against requirements and user expectations.

White Box Testing: Examines the internal structure, logic, and code of the software. Testers use their understanding of the implementation to design test cases that exercise different paths within the code.

Both approaches are essential for comprehensive testing, with black box testing ensuring that the software meets user needs and requirements, while white box testing verifies the correctness of the internal logic and implementation.



Black Box Testing

- Types of errors regarding functional requirements of software:
  - -- Incorrect or missing functions
  - -- Interface errors
  - -- Error in data structure & external data base access
  - -- Performance errors
  - -- Initialization & termination errors
- No functional requirements NO Black Box Testing.
- Demonstrates that each function is fully operational.
- Uncovers different kind of errors than white box testing.
- Performed later in the testing process.

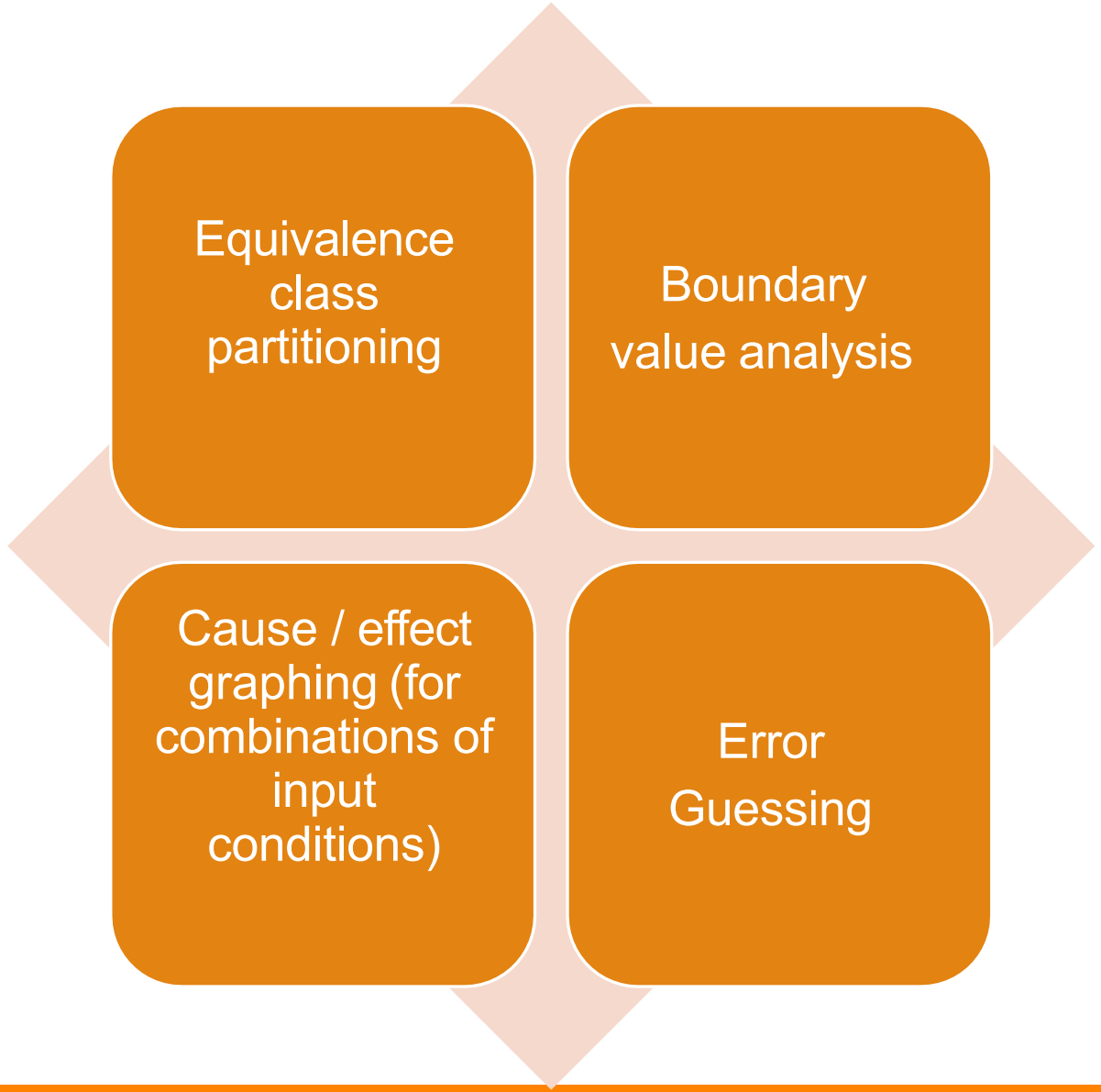
- Black box techniques derive a set of test cases that satisfy the following criteria:

    1. Test cases reduce the number of additional test cases that must be designed to achieve reasonable testing

    2. Test cases that tell us something about the presence or absence of classes of errors, rather than errors associated only with the specific test at hand.

- Black box techniques can supplement the test cases generated by white box.

How to Design Test Cases?



Equivalence Partitioning

- Type of Black Box Testing.
- Equivalence Class Partitioning (ECP) is a used to divide input data into groups (equivalence classes) that are expected to exhibit similar behavior. The goal is to reduce the number of test cases while ensuring adequate test coverage.
- By identifying this as an equivalence class, we are stating that if no error is found by a test of one element of the set, it is unlikely that an error would be found by a test of another element of the set.

Example

Travel service offers discounts to travelers based on their age.

- 0-4 years 100%
- 5-15 years 50%
- 16-64 years 0%
- 64 years and older 25%

Equivalence classes for age

- 0, 1, 2, 3, 4
- 5, 6, 7, …15
- 16, 17, 18, …64
- 65, 66, 67, …120

Similarly for destinations

- **e.g., destination can be grouped based on regions (that have same fair)**

Nothing special for name

Identifying Equivalence classes

A key concept in the identification of classes is negation, i.e. If a characteristic is identified as an equivalence class, then one should immediately negate the characteristic in order to find examples of classes which should cause the module to do something different such as “generate an error message”.

The idea is that for every valid input class, there should be at least one corresponding invalid input class that leads to different system behavior, such as error handling.

Concept of Negation

Let's say a system accepts **ages between 18 and 65** as valid input.

- **Valid equivalence class**: Ages **18 to 65** (system should process normally)
- **Negated (invalid) equivalence classes:**
  - Ages **less than 18** (e.g., 17, 0, -5 → should cause an error)
  - Ages **greater than 65** (e.g., 66, 100, 150 → should cause an error)

By systematically negating characteristics, we ensure that the software correctly differentiates between acceptable and unacceptable inputs, improving its robustness and reliability.

Key Concepts

- **Equivalence Class**: A set of input values that should produce the same outcome.
- **Valid Equivalence Class**: A group of inputs that the system should accept and process correctly.
- **Invalid Equivalence Class**: A group of inputs that the system should reject or handle with an error

Steps to Apply ECP

1. **Identify Input Variables**: Determine all input fields and their possible values.
2. **Define Equivalence Classes**: Group values into valid and invalid categories.
3. **Select Representative Test Cases**: Pick one value from each class for testing.
4. **Execute & Validate**: Run tests and check if the system behaves as expected.

Equivalence Partitioning Guidelines

- Range of values condition
  - Input spec: itemCount can be from 1 to 999, inclusive.
    - identify one valid (1<itemCount<999)
    - identify two invalid (itemCount < 1, itemCount>999)

- Countable set of values condition
  - Input spec: one through six owners can be listed…
    - identify one valid owner
    - identify two invalid (zero owners, more the six owners)

Identifying the Equivalence Classes

- Finite but uncountable set of values condition
  - Input spec: Values (BUS, TRUCK, TAXICAB, PASSENGER, MOTORCYCLE)
    - identify one valid for each
    - and one invalid (not in set, e.g. TRAILER)
- Binary (must be) condition
  - Input spec: First character of the identifier must be a alpha
    - identify one valid (an alpha, a..z)
    - identify one invalid (not an alpha)

Equivalence Class Test Cases

Consider a numerical input variable, i, whose values may range from -200 through +200. Then a possible partitioning of testing input variable by 4 people may be:

-200 to -100
-101 to 0
1 to 100
101 to 200

Define "same sign" as the equivalence relation, R, defined over the input variable's value set, i = {-200 - -,0, - -, +200}. Then one partitioning will be:

-200 to -1 (negative sign)
0 (no sign)
1 to 200 (positive sign)

A sample equivalence test case "set" from the above "same sign" relation would be { -5; 0; 8 }

Equivalence Partitioning Guidelines

- If the input condition is a ***range***, identify one valid equivalence class member and two invalid equivalence class members.
- If input condition is a ***number* of values**, identify one valid and two invalid.
- If input condition is a ***set of input values*** (each to be handled differently), identify one valid for each and one invalid.
- If input condition specifies **a *must be situation***, identify one valid and one invalid.
- If elements in an equivalence class are not handled in the same manner, split the equivalence class.

Guidelines for Defining Equivalence Classes

If an input condition specifies a range, one valid and two invalid equivalence classes are defined

Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}

If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}

If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined

Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}

If an input condition is a Boolean value, one valid and one invalid class are define

Input: {true condition} Eq classes: {true condition}, {false condition}

Equivalence Class Testing

- Use the mathematical concept of partitioning into equivalence classes to generate test cases for Functional (Black-box) testing

- The key goals for equivalence class testing are similar to partitioning:
  - ü completeness of test coverage
  - ü lessen duplication of test coverage

Test Case and the Equivalence Classes

- Formally, **one test case per equivalence class should be enough**.
- A black box method aimed at increasing the **efficiency of testing** and, at the same time, **improving coverage of potential error conditions**

Test Case and the Equivalence Classes

According to the equivalence class partitioning method:

- Each valid EC and each invalid EC are included in at least one test case.

- Note: equivalence CLASS!
  - Ø For an acceptable range of input integers is 1-10, we need at least one test with a value in the range 1-10.
  - Ø Boundaries: 1 and 10
  - Ø Invalid EC classes might be EC with values < 1, another class > 10, another one - negative numbers, ...

- Note: Run tests for invalid equivalence classes one at a time.

Weak Normal Equivalence Class Testing

- **Normal Range**: Equivalence classes representing typical or normal input values that the system is expected to handle correctly. These values fall within the expected range of inputs and are likely to be processed without errors.

- **Weak Range**: Equivalence classes representing values that are at the boundaries or just outside the normal range. These values are chosen to test the behavior of the system at its limits or near boundaries, where errors or unexpected behavior may occur.

"weak" in this context may stem from the perspective that these values are not fully representative of the normal or typical range of inputs but rather lie at the edges where behavior might change or errors might occur. Therefore, they are termed "weak" to emphasize their significance in testing, as they help uncover issues related to boundary conditions and edge cases.

Example

Suppose the system has the following requirements:

1. Packages weighing between 0 and 10 kilograms are charged a standard shipping rate.
2. Packages weighing more than 10 kilograms are charged an additional fee for each kilogram over 10.

Based on these requirements, we can identify the following equivalence classes:

1. **Normal Equivalence Classes:**
   - Packages weighing between 0 and 10 kilograms (inclusive).

2. **Weak Equivalence Classes:**
   1. Packages weighing just below the normal range boundary (e.g., 0 kilograms).
   2. Packages weighing just above the normal range boundary (e.g., 10 kilograms).
   3. Packages weighing significantly above the normal range (e.g., 11 kilograms).

Example of Weak Equivalence Testing Class (WETC) *

- Number of WETCs need= Max number of equivalence classes among {A, B, C}
- **Given: |A| = 3, |B| = 4, |C| = 2**
- 4 WETCs are enough

Test Case	A	B	C
WETC1	a1	b1	c1
WETC2	a2	b2	c2
WETC3	a3	b3	c1
WETC4	a1	b4	c2

#test cases = #classes in the partition with the largest numbering of subsets.

Strong Equivalence Classes

- It focuses on testing inputs that are fully representative of the expected behavior, ensuring that all required components are present and fulfilling the system's requirements.

- The Cartesian product guarantees that we have a notion of “completeness” in following two ways:

  a) We cover all equivalence classes.
  b) We have one of each possible combination of inputs.

Example

Suppose the system has the following requirements:

1. Passwords must be at least 8 characters long.
2. Passwords must contain at least one uppercase letter, one lowercase letter, one digit, and one special character.

Example Passwords:

1. "abc" - This password belongs to the Weak Equivalence Class of Short Passwords.
2. "Password1" - This password belongs to the Strong Equivalence Class of Passwords with All Required Components.
3. "12345678" - This password belongs to the Weak Equivalence Class of Long Passwords but is missing the required components.
4. "TestPass" - This password belongs to the Weak Equivalence Class of Long Passwords but is missing a digit and a special character.
5. "!@#test" - This password belongs to the Strong Equivalence Class of Passwords with All Required Components.

Example of Strong Equivalence Class Testing (SECT)

|A| = 3,

|B| = 4,

|C| = 2

# of test cases
- 3x4x2= 24

Test Case	A	B	C
SETC1	a1	b1	c1
SETC2	a1	b1	c2
SETC3	a1	b2	c1
SETC4	a1	b2	c2
SETC5	a1	b3	c1
SETC6	a1	b3	c2
SETC7	a1	b4	c1
SETC8	a1	b4	c2
SETC9	a2	b1	c1
SETC10	a2	b1	c2
SETC11	a2	b2	c1
SETC12	a2	b2	c2
SETC13	a2	b3	c1
SETC14	a2	b3	c2
SETC15	a2	b4	c1
SETC16	a2	b4	c2
SETC17	a3	b1	c1
SETC18	a3	b1	c2
SETC19	a3	b2	c1
SETC20	a3	b2	c2
SETC21	a3	b3	c1
SETC22	a3	b3	c2
SETC23	a3	b4	c1
SETC24	a3	b4	c2

Explanation

- |A| = 3 (3 values in equivalence class A)
- |B| = 4 (4 values in equivalence class B)
- |C| = 2 (2 values in equivalence class C)

We create tuples representing each possible combination:

1. From equivalence class A, we have 3 values: A1, A2, A3.
2. From equivalence class B, we have 4 values: B1, B2, B3, B4.
3. From equivalence class C, we have 2 values: C1, C2.

The Cartesian product is generated by taking all possible combinations of these values:

1. (A1, B1, C1), (A1, B1, C2), (A1, B2, C1), (A1, B2, C2), (A1, B3, C1), (A1, B3, C2), (A1, B4, C1), (A1, B4, C2)
2. (A2, B1, C1), (A2, B1, C2), (A2, B2, C1), (A2, B2, C2), (A2, B3, C1), (A2, B3, C2), (A2, B4, C1), (A2, B4, C2)
3. (A3, B1, C1), (A3, B1, C2), (A3, B2, C1), (A3, B2, C2), (A3, B3, C1), (A3, B3, C2), (A3, B4, C1), (A3, B4, C2)

Testing for Robustness

- If error conditions are a high priority, we should extend strong equivalence class testing to include both valid (E) and invalid inputs (U)
  - For example: age < 0 and age > 120

- Again, robustness can be applied with WECT and SECT

NextDate Example

- NextDate is a function with three variables: month, day, year. It returns the date of the day after the input date.
  - Limitation: 1812-2020

- Treatment Summary:
  - if it is not the last day of the month, the next date function will simply increment the day value.
  - At the end of a month, the next day is 1 and the month is incremented.
  - At the end of the year, both the day and the month are reset to 1, and the year incremented.
  - Finally, the problem of leap year makes determining the last day of a month interesting.

Equivalence Classes

Valid Equivalence Classes

- M1 = {1 <= month <= 12}
- D1 = {1 <= day <= 31}
- Y1 = {1812 <= year <= 2020}

Valid months: January (1) to December (12)

Valid days: 1 to 31 (based on the month, considering leap years and months with fewer days)

Valid years: Up to 2020

Poor Equivalence Classes

Valid Equivalence Classes

- M1 = {1 <= month <= 12}
- D1 = {1 <= day <= 31}
- Y1 = {1812 <= year <= 2020}

- Should take into account special cases
- Leap year, February

Better classes

1. **Valid Equivalence Classes for Month (M)**:
   - M1: January (1), March (3), May (5), July (7), August (8), October (10), December (12)
   - M2: April (4), June (6), September (9), November (11)
   - M3: February (2)
2. **Valid Equivalence Classes for Day (D)**:
   - D1: Days 1 to 28 (valid for all months)
   - D2: Day 29
   - D3: Day 30
   - D4: Day 31 (valid for months in M1: January, March, May, July, August, October, December)
3. **Valid Equivalence Classes for Year (Y)**:
   - Y1: Year 1900
   - Y2: Years 1812 to 2020 (inclusive) that are not 1900 and are divisible by 4 (leap years)
   - Y3: Years 1812 to 2020 (inclusive) that are not 1900 and are not divisible by 4 (non-leap years)

Better classes

Valid Equivalence Classes

- M1 = {1, 3, 5, 7,8, 10, 12}
- M2 = {4, 6, 9, 11}
- M3 = {2}
- D1 = {1 <= day <= 28}
- D2 = {29}
- D3 = {30}
- D4 = {31}
- Y1 = {year = 1900}
- Y2 = {1812 <= year <= 2020 AND (year != 1900) AND (year mod 4 = 0)}
- Y3 = {1812 <= year <= 2020 AND year mod 4 != 0 }

Weak Equivalent Class - Test cases

#test cases=maximum partition size (D)=4

Test Case ID	Month	Day	Year	Output
WETC1	6	14	1900	6/15/1900
WETC2	7	29	1912	7/30/1912
WETC3	2	30	1913	Invalid Input date (not possible)
WETC4	6	31	1900	Invalid Input date

Explanation

1. **Test Case 1: Month 6, Day 14, Year 1900**
   - Expected Output: 6/15/1900
   - Explanation: The given date is June 14, 1900. The next date after June 14, 1900, is June 15, 1900. This date follows the sequential order of days in the month.
2. **Test Case 2: Month 7, Day 29, Year 1912**
   - Expected Output: 7/30/1912
   - Explanation: The given date is July 29, 1912. The next date after July 29, 1912, is July 30, 1912. This date follows the sequential order of days in the month.
3. **Test Case 3: Month 2, Day 30, Year 1913**
   - Expected Output: Invalid Input
   - Explanation: The given date is February 30, 1913. February does not have 30 days, so this date is invalid.
4. **Test Case 4: Month 6, Day 31, Year 1900**
   - Expected Output: Invalid Input
   - Explanation: The given date is June 31, 1900. June has only 30 days, so this date is invalid.

Strong Equivalent Class Testing - Test cases

- #SECT test cases= partition size (D) x partition size (M) x partition size (Y) = 3x4x3=36 test cases

Test Case ID	Month	Day	Year	Expected Output
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERROR
SE11	6	31	1912	ERROR
SE12	6	31	1913	ERROR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERROR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERROR
SE31	2	30	1900	ERROR
SE32	2	30	1912	ERROR
SE33	2	30	1913	ERROR
SE34	2	31	1900	ERROR
SE35	2	31	1912	ERROR
SE36	2	31	1913	ERROR