Software Quality Assurance

Specification-Based Test Case Design Techniques

SQA – IT4100

Learning Outcomes

Specification based testing techniques

- Equivalence partitioning
- Boundary value analysis
- Decision table

Specification Based Testing Technique

- Discovering what claims are made in the specifications and testing the product against them.
- Only concentrates on what the software does, not how it does it.
- Do not require the knowledge of how the system or component is structured.
- Also known as Behavior Based Testing and Black Box Testing.

Specification Based Testing Design Technique

- Uses the specification of the program as the point of reference for test data selection and adequacy.
- A specification can be anything like a written document, collection of use cases, a set of models, or a prototype.



Types of Specification Based Testing Techniques

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Tables
- State Transitioning

Equivalence Partitioning

- A technique to ensure the maximum test coverage using minimal test cases .
- Divides the input data of a software unit into partitions of equivalent data from which test cases can be derived.
- The idea behind the technique is to divide a set of test conditions into groups or sets that can be considered as same.

Equivalence Partitioning (Cont.)

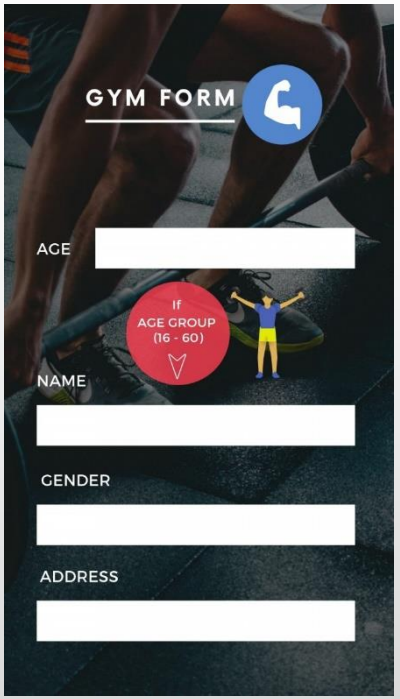
- Equivalence partitioning works on certain assumptions.

    1. The system will handle all the test input variations within a partition in the same way.
    2. If one of the input condition passes, then all other input conditions within the partition will pass as well.
    3. If one of the input conditions fails, then all other input conditions within the partition will fail as well.

- The success and effectiveness of Equivalence partitioning lie in how reasonable the above assumptions hold.

Example

Fitness1st gym provides membership for the clients between the age of 16 to 60. It has the following online application form for getting the gym membership.



In this form;

- User has to fill the age first.
- User can go further only if the age is between 16 to 60.
- Otherwise, there will be a message saying that you cannot get a membership.

Testing the form

- Form has to be tested with values;
  - Less than 16
  - Between 16-60
  - More than 60

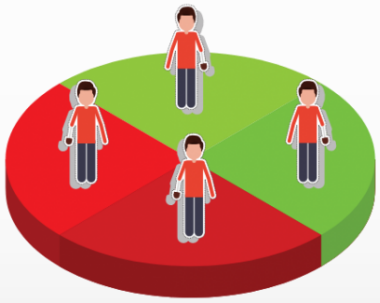
- How many combinations are required to say that the functionality works safely?
  - <16 has 15 combinations from 0-15. More conditions can be added by considering the negative values.
  - 16-60 has 45 combinations
  - >60 has 40 combinations (if only consider till 100)

- Is it possible to test all these 100+ combinations?

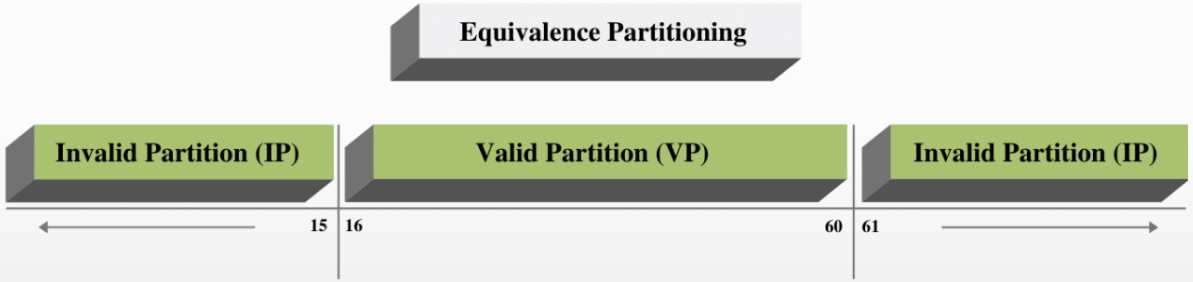


How to do Equivalence Partitioning? (Cont.)

Divide (partition) the input values into sets of valid and invalid partitions.



Equivalence Partitioning



- Valid partitions are values that should be accepted by the component or system under test. This partition is called "**Valid Equivalence Partition**."
- Invalid partitions are values that should be rejected by the component or system under test. This partition is called "**Invalid Equivalence Partition**."

How to do Equivalence Partitioning? (Cont.)

- Test conditions required to test the system:

  - ✓ Enter Age = 5
  - ✓ Enter Age = 20
  - ✓ Enter Age = 65

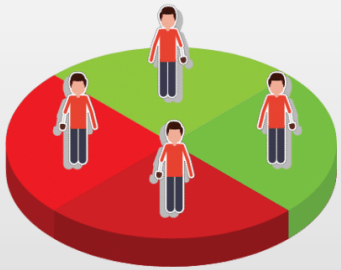


How to do Equivalence Partitioning? (Cont.)

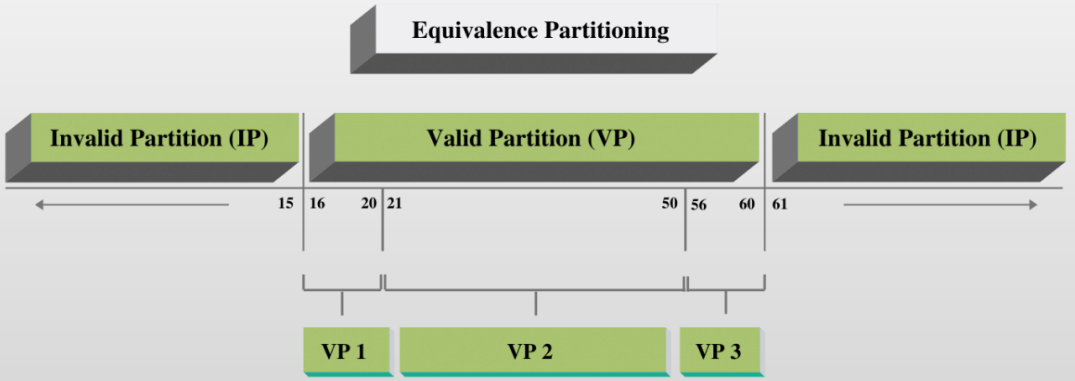
- These partitions can be further divided into sub-partitions if required.

Assume that if the age is 16-20 years old or 55-60 years old, there is an additional requirement to attach a proof of the age, while submitting the membership form.

In this case, partition will look like below;



Equivalence Partitioning



How to do Equivalence Partitioning? (Cont.)

- VP1, VP2, and VP3 are all valid sub partitions based on the additional requirements.

- Therefore, test conditions could be:
  - ✓ Enter Age = 5
  - ✓ Enter Age = 18
  - ✓ Enter Age = 30
  - ✓ Enter Age = 58
  - ✓ Enter Age = 65

- These five conditions will cover all the requirements for the age field.
- It is possible to use any value from each partition.

Equivalence Partitioning (Cont.)

- Ensure that the partitioning is unique and not overlapping. Each value taken should belong to only one partition.

- Total test coverage can be achieved when the test cases cover all identified partitions.

- Equivalence partitioning test coverage can be measured as the number of partitions tested divided by the total number of recognized partitions.

Drawbacks of Equivalence Partitioning

- Success of equivalence partitioning is dependent on the ability to create correct partitions.

- Ability to create partition limits based on what is mentioned in requirements. Therefore, testers who decides the test conditions will not have an understanding about the approach used by the developer.

Equivalence Partitioning - Exercise

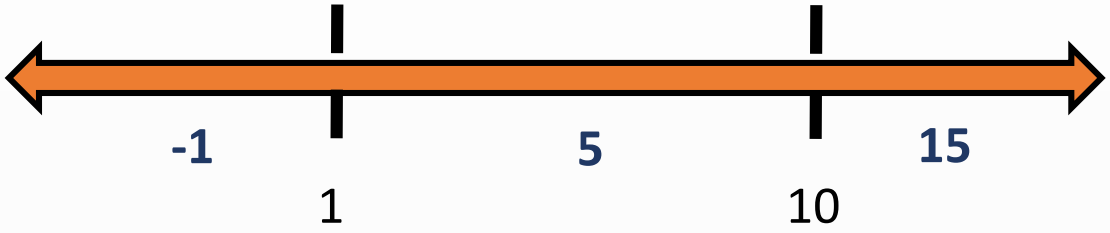
Consider the behavior of “Order Pizza” text box given below.



- Requirement of the above function.
  - Pizza values 1 to 10 are considered valid. A success message is shown.
  - Pizza values less than 1 are considered invalid and an error message will appear, “Please enter a valid count“
  - Pizza values grater than 10 are considered invalid and an error message will appear, "Only 10 Pizzas can be ordered at a time“

Perform equivalence partitioning and identify the test conditions with sample values for the above function.

Answer



Test conditions:

- ✓ Any number less than 1 that is 0 or below is considered invalid.

  Enter Pizza Value = -1

- ✓ Numbers 1 to 10 are considered valid.

  Enter Pizza Value = 5

- ✓ Any Number greater than 10 is considered invalid.

  Enter Pizza Value = 15

Boundary Value Analysis

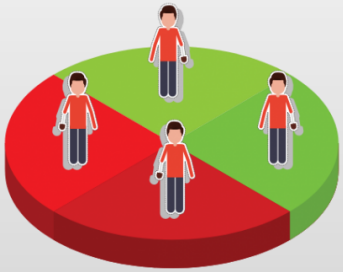
- A software testing technique in which tests are designed to include representatives of boundary values in a range.
- An extension of equivalence partitioning.
- Testing the boundaries of partitions.
- Useable only when the partition is ordered, consisting of numeric or sequential data.
- The minimum and maximum values of a partition are its boundary values.

Boundary Value Analysis (Cont.)

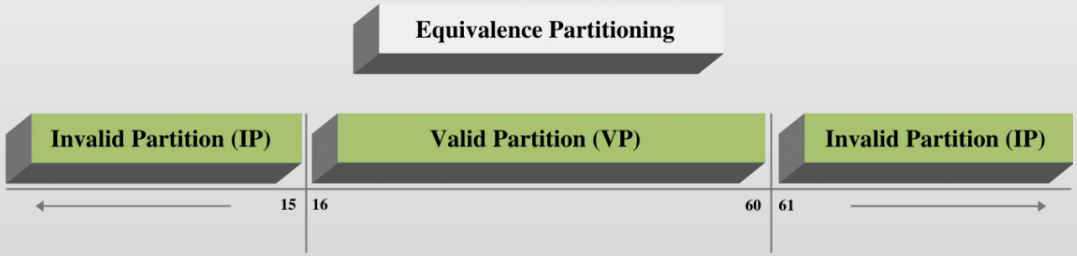
- There are high chances of finding the defects at the boundaries of a partition.
- Equivalence partitioning alone is not sufficient to catch such defects.
- Boundary Value Analysis was designed to detect anomalies at the boundaries of a partition.

How to do Boundary Value Analysis?

- Refer to the same example of Fitness1st gym form (on equivalence partitioning) where the user is required to enter the age.
- The first step of boundary value analysis is to create the equal partitions.



Equivalence Partitioning



How to do Boundary Value Analysis? (Cont.)

- Concentrate on the valid partition, which ranges from 16-60.
- There is a three-step approach to identify boundaries:
  1. Identify the exact boundary value of this partition Class → 16 and 60.
  2. Get the boundary value which is one less than the exact boundary → 15 and 59.
  3. Get the Boundary Value which is one more than the precise Boundary → 17 and 61.



How to do Boundary Value Analysis? (Cont.)

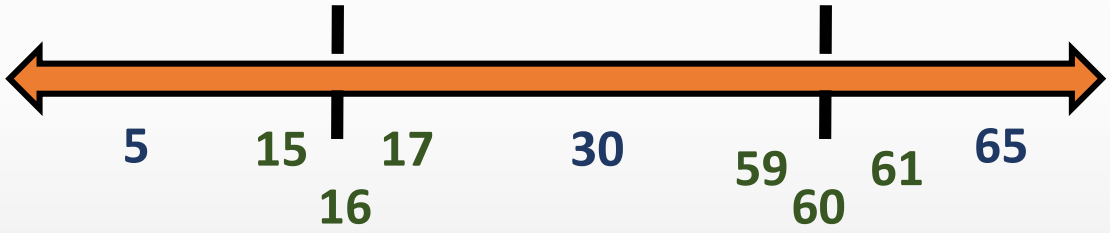
- Following are the combinations for boundary value for the age criteria.
  - ✓Valid Boundary Conditions : Age = 16, 17, 59, 60
  - ✓Invalid Boundary Conditions : Age = 15, 61

- Valid boundary conditions fall under valid partition class, and invalid boundary conditions fall under invalid partition class.

Boundary Value Analysis with Equivalence Partitioning

- Boundary Value Analysis is combined with equivalence partitioning to get a full set of test conditions.

  Refer to the previous example;



- The range is from 16 – 60;
  - **Equivalence partition analysis gives test conditions as 5, 30, 65**
  - **Boundary value analysis gives test conditions as 15, 16, 17, 59, 60, 61.**

- **All test conditions: 5, 15, 16, 17, 30, 59, 60, 61, 65**

Drawbacks of Boundary Value Analysis

- Boundary value and equivalence partitioning assume that the application will not allow you to enter any other characters or values. This assumption is, however, not valid for all applications.

- Boundary value analysis cannot handle situations where the decision depends on more than one input values.

  Example: If the Gym form has another field Male and Female, and the age limit will vary according to that selection.

Boundary Value Analysis- Exercise

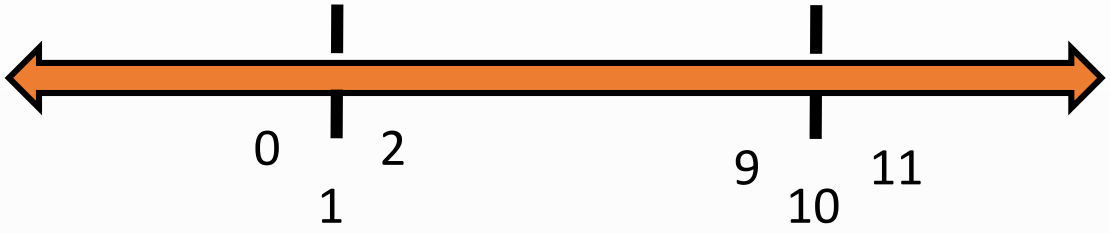
Consider the behavior of “Order Pizza” text box given below.



- Requirement of the above function.
  - Pizza values 1 to 10 are considered valid. A success message is shown.
  - Pizza values less than 1 are considered invalid and an error message will appear, “Please enter a valid count“
  - Pizza values grater than 10 are considered invalid and an error message will appear, "Only 10 Pizzas can be ordered at a time“

Perform boundary value analysis and identify boundary value test conditions for the above function.

Answer



Boundary value analysis test conditions:

- ✓ Value = 0 → Invalid
- ✓ Value = 1 → Valid
- ✓ Value = 2 → Valid
- ✓ Value = 9 → Valid
- ✓ Value = 10 → Valid
- ✓ Value = 11 → Invalid

Decision Tables

- Software Testing technique in which tests are more focused on business logic or business rules.
- A decision table is a good way to deal with combinations of inputs.
- Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers.
- Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.

How to use Decision Tables for test designing?

1. Identify a suitable function or subsystem which reacts according to a combination of inputs or events. The system should not contain too many inputs otherwise the number of combinations will become unmanageable.
   - *It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.*

How to use Decision Tables for test designing? (Contd.)

2. Put all identified functions into a table, listing all the combinations of True and False for each of the aspects.

Example 1: *A loan application can enter the amount of the monthly repayment or the number of years a customer wants to take to pay it back (the term of the loan). If the monthly repayment is entered by the user, then the application will process the loan amount. If the number of years is entered by the user, then the application will process the term.*

The two conditions are the loan amount and the term. Therefore, put them in a table.

Conditions
Repayment amount has been entered:
Term of loan has been Entered:

How to use Decision Tables for test designing? (Contd.)

3. Identify all of the combinations (rules) of True and False.

With two conditions, each of which can be True or False, there will be four combinations.

Number of Combinations = Two to the power of the number of conditions

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:				
Term of loan has been Entered:				

How to use Decision Tables for test designing? (Contd.)

Decision table with input combinations

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:	T	T	F	F
Term of loan has been Entered:	T	F	T	F

4. Identify the correct outcome for each combination. In this example, user can enter one or both of the two fields. Each combination is also referred to as a rule.

How to use Decision Tables for test designing? (Contd.)

Decision table with input combinations with correct outcome for each combination.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:	T	T	F	F
Term of loan has been Entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:	Y	Y		
Process term:	Y		Y	

Decision Tables for test designing

- **What happens if the customer doesn't enter anything in either of the two fields?**
- The table has depicted a combination that was not mentioned in the specification.
- Can assume that this combination should result in an error message. Therefore, add another action to the decision table.

Decision Tables for test designing (Contd.)

Decision table with additional outcomes

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:	T	T	F	F
Term of loan has been Entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:	Y	Y		
Process term:	Y		Y	
Error message:				Y

Decision Tables for test designing (Contd.)

- This highlights the strength of this technique to discover omissions and ambiguities in specifications.

- It is not unusual for some combinations to be omitted from specifications. Therefore, this is also a valuable technique to use when reviewing the test basis.

Decision Tables for test designing – Example 2

- Change in this example 1, so that the customer is not allowed to enter both repayment and term.

- Now the outcome of our table will change, because there should also be an error message if both are entered.

Example 2

Decision table for Example 2

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:	T	T	F	F
Term of loan has been Entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:		Y		
Process term:			Y	
Error message:	Y			Y

Example 2

- Notice that there is only one ‘Yes’ for the actions in each column.
- This is called as mutually exclusive actions – only one action occurs for each combination of conditions.

Example 2

It is possible to represent the decision table in a different way by listing the actions in the cell of one row.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered:	T	T	F	F
Term of loan has been Entered:	T	F	T	F
Actions/Outcomes				
Result:	Error Message	Process loan amount	Process term	Error Message

Note that if more than one action results from any of the combinations, then it is better to show them as separate rows rather than combining them into one row.

How to use decision tables for test designing? (Contd.)

- The final step of this technique is to write test cases to exercise each of the four rules in our table.



Decision Tables - Exercise

A customer can use a credit card with three conditions as follows;

- New customer will get a 15% discount on all your purchases today.
- If the customer is an existing customer and holds a loyalty card, then he or she will get a 10% discount.
- If the customer has a coupon, then he or she can get 20% off today. (but it can't be used with the 'new customer' discount).

Create the decision table for above scenario.

Answer

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New Customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions/Outcomes								

Answer

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New Customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions/Outcomes								
Discount (%)	X	X	20	15	30	10	20	0

SLIIT
Discover Your Future

Software Quality Assurance

Code Coverage Analysis

SQA – IT4100

Learning Outcomes

Code Coverage Methods

- Statement Coverage
- Decision Coverage
- Path Coverage
- Condition Coverage
- Multiple Condition Coverage

Code Coverage

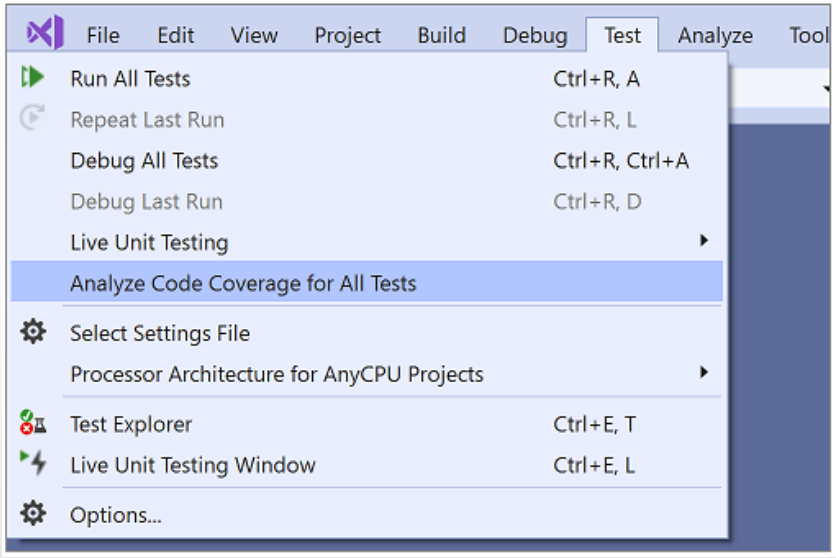
- Code coverage is a term used in software testing to describe how much program source code is covered by a testing plan.
- Developers look at the number of program subroutines and lines of code that are covered by a set of testing resources and techniques.
- Code coverage is also known as test coverage.

Code Coverage Analysis

- Code coverage analysis can provide reassurance that programs are broadly tested for bugs and relatively error-free.
- Code coverage analysis is done mostly to find the precise areas that are not covered by testing strategies.
- Code coverage analysis and other testing aspects in beta or other development rounds exposes bugs to a small test audience, rather than to millions of users when products finally go live.

Code Coverage Analysis (Cont.)

- Microsoft Visual Studio have specific menu tools for doing code coverage analysis.



Reference: https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2019

Code Coverage Analysis (Cont.)

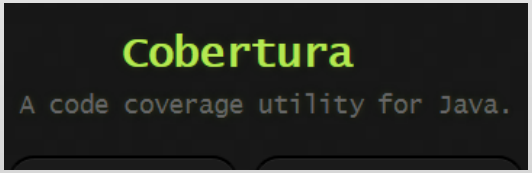
- Developers may use relatively manual methods that involve mapping out the software source code and determining where testing applies.
- Third-party vendors also provide specific code coverage tools for different programming languages.











Uses of Code Coverage Analysis

- Helps to measure the efficiency of test implementation
- Offers a quantitative measurement.
- Defines the degree to which the source code has been tested.

Code Coverage Methods

- Statement Coverage
- Decision Coverage
- Path Coverage
- Condition Coverage
- Multiple Condition Coverage

Statement Coverage

- A metric which ensures that each statement of the code is executed at least once.
- Measures the number of lines executed.
- Verifies what the written code is expected to do and not to do.
- This method can be considered as white box testing, as it intends to evaluate the internal structure of the code.
- A programmer is the one who can perform this task efficiently.

Statement Coverage Calculation

$$\textbf{Statement Coverage} = \frac{\textit{Number of executed statements}}{\textit{Total number of statements}} \times 100$$

- *Note: All statements including the statement with a function name and statements with only braces ("{" "}") are counted in statement coverage.*

Statement Coverage - Example

```
read a;
read b;
if (a>b)
        print "A is greater than B";
else
        print "B is greater than A";
```

Test Conditions:

1. a = 5, b = 1
2. a = 1, b = 5

Statement Coverage – Example (Cont.)

Total number of Statements = 6

Condition 1 :

If a=5 and b=1, then the first print statement is executed.

Number of distinct statements executed = 4

Condition 2 :

If a=1 and b=5, then the second print statement is executed.

Number of distinct statements executed = 2

Statement Coverage – Example (Cont.)

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of Statements}} \times 100\%$$

Statement Coverage = [(4+2) / 6] x 100% = (6 / 6) x 100%

= 100%

Statement Coverage – Exercise

Calculate the statement coverage for the given code in following conditions.

When;

1. a = 5 and b = 7
2. a = 4 and b = 4

```
Read a;
read b;
i=0
if a>b
  while(i<a)
    print(i)
    i++;
  end while
else
  while(i<b)
    print(i)
    i++;
  end while
end if;
```

Decision Coverage

- Decision coverage reports the true or false outcomes of each Boolean expression.
- There are many different methods of reporting this metric. All these methods focus on covering the most important combinations.

Decision Coverage Calculation

$$\textit{Decision Coverage} = \frac{\textit{Number of decision outcomes exercised}}{\textit{Total number of decision outcomes}} \times 100\%$$

Decision Coverage - Example

```
Read a;
If (a> 5)
        a=a*3
print (a)
```

Test Conditions:

1. a = 2
2. a = 7

Decision Coverage – Example (Cont.)

Total number of decision outcomes = 2

Condition 1 :

If a=2, then the statement inside is not executed.

Number of distinct decision outcomes exercised = 1

Here the “FALSE" outcome of the decision If (a>5) is checked.

Condition 2 :

If a=7, then the statement inside is executed.

Number of distinct decision outcomes exercised = 1

Here the “TRUE" outcome of the decision If (a>5) is checked.

Decision Coverage – Example (Cont.)

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

Statement Coverage = [(1+1) / 2] x 100% = (2 / 2) x 100%

= 100%

Decision Coverage – Exercise

Calculate the decision coverage for the given code in following conditions.

When;

1. a = 5 and b = 7
2. a = 3 and b = 1

```
read a;
read b;
if a>b
   while (a>b)
      print("B is still small")
      b++
      if(b>a)
         print("B is now greater than A")
else
   if b>7
     print("B is greater than A")
```

Path Coverage

- A path is a unique sequence of branches from the function entry to the exit.
- Path coverage covers a function from its entry till its exit point.
- Path coverage refers to designing test cases such that all linearly independent paths in the program (method) are executed at least once.
- A control flow graph describes how the control flows through the application.
- A linearly independent path can be defined in terms of what's called a control flow graph of an application.
- A linearly independent path is a path with at least one new edge in the control flow graph.
- Cyclomatic complexity metric can be used to identify the number of independent paths in a program.

Path Coverage Calculation

$$\textit{Path Coverage} = \frac{\textit{Number of linearly independent paths executed}}{\textit{Total number of linearly independent paths}} \times 100\%$$

Path Coverage

- **Steps:**
  1. Draw the control flow graph
  2. Identify linearly independent paths and total number of linearly independent paths in the program.
  3. Identify linearly independent paths executed by each test condition.
  4. Identify the number of linearly independent paths covered by all test conditions.
  5. Calculate the path coverage.

Path Coverage - Example

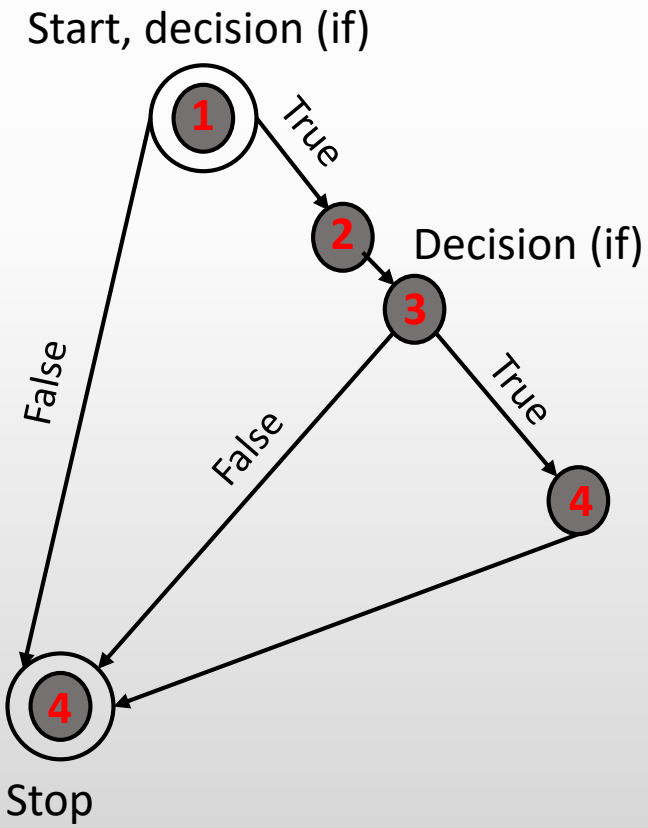
```
Demo(int a)
If (a> 5)
        a=a*3
        if (b > 8)
                b = b – 5
```

Test Conditions:

1. a = 4, b = 6
2. a = 6, b = 6

Path Coverage – Example (Cont.)

- Steps 1: Draw the Control Flow graph.



Path Coverage – Example (Cont.)

- Steps 2: Identify linearly independent paths and total number of linearly independent paths in the program.

  Cyclomatic Complexity:

  V(G) = e – n + 2 = 6 - 5 + 2 = **3**

  Total number of linearly independent paths = 3

  Path 1: 1→5

  Path 2: 1→2→3→5

  Path 3: 1→2→3→4 → 5

Path Coverage – Example (Cont.)

Steps 3: Identify linearly independent paths executed by the each test condition.

Condition 1:

If a=4, and b = 6

Paths executed = Path 1: 1→5

Condition 2:

If a=6, and b = 6

Paths executed = Path 2: 1→2→3 →5

Path Coverage – Example (Cont.)

Steps 4: Identify the number of linearly independent paths covered by all test conditions.

- Paths covered by all test conditions:

  Path 1: 1→5

  Path 2: 1→2→3 →5

Number of linearly independent paths covered by all test conditions = 2

Path Coverage – Example (Cont.)

Steps 5: Calculate the path coverage.

Number of linearly independent paths executed = 2

Total number of linearly independent paths = 3

Path Coverage = (2/3) * 100 = 67%

Path Coverage – Exercise 1

Calculate the path coverage for the given code in following conditions.

When;

1. a = 5 and b = 7
2. a = 3 and b = 1

```
read a;
read b;
if a>b
   while (a>b)
      print("B is still small")
      b++
      if(b>a)
         print("B is now greater than A")
else
   if b>7
   print("B is greater than A")
```

Multiple Condition Coverage

- Multiple Condition Coverage checks the coverage of all combinations of conditions in a program.
- Total test cases for a program with n number of conditions will be 2 to the power n.
  - If there are two conditions, then n=2.
  - Therefore, 4 (2^2) test cases are needed to get the full multiple condition coverage.

Code Coverage – Exercise

Calculate the statement coverage, decision coverage, and path coverage for the given code in following conditions.

When;

1. input = 5
2. input = 20

```
read input;
if input>10
   while (input!=0)
      print("Valid")
      if(input%2==0)
         print("Even")
      else
         print("Odd")
      input = input - 1
else
   if input>0
      print("Please enter valid value!!")
   else
      print("Enter a value greater than 10")
```

SLIIT

Discover Your Future

Software Metrics

SLIIT
FACULTY OF COMPUTING

Software Quality

- How can software quality be measured?
- Software quality metrics:
  - Code Quality
  - Reliability
  - Performance
  - Usability
  - Correctness
  - Maintainability
  - Integrity
  - Security

Cyclomatic Complexity (CC)

- Measures the number of linearly **independent paths** in a program.

V(G) = d + 1

V(G) = e – n + 2

Vg = No of decision statements in each method + No of methods in a class

Approaches to Measure Cyclomatic Complexity

- Explain some approaches that can be used to measure the Cyclomatic Complexity of a program.

Output of RSM

```
ark:bvk>rsm -Tf Results.java

Function: Results.OutResults
Complexity      Param 1      Return 4      Cyclo Vg 4      Total      9
LOC 10          eLOC 9       lLOC 4        Comment 0       Lines      10
Function Points              FP(LOC) 0.2   FP(eLOC) 0.2    FP(lLOC)   0.1

                    ~~ Project Functional Analysis ~~

Total Functions .....:    1     Total Physical Lines .....:    10
Total LOC .....:          10    Total Function Pts LOC.....:   0.2
Total eLOC .....:         9     Total Function Pts eLOC .....: 0.2
Total lLOC.....:          4     Total Function Pts lLOC.....:  0.1
Total Cyclomatic Comp.....:   4     Total Interface Comp .....:    5
Total Parameters .....:       1     Total Return Points .....:     4
Total Comment Lines .....:    0     Total Blank Lines .....:       0
```

Calculating Cyclomatic Complexity of a Byte Code

- Calculate the cyclomatic complexity of the following byte code?

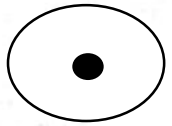
```
Method void D0(boolean, java.lang.String)
   0 iload_0
   1 ifeq 11
   4 getstatic #10 <Field java.io.PrintStream out>
   7 aload_1
   8 invokevirtual #16 <Method void println(java.lang.String)>
  11 return
```

Question??

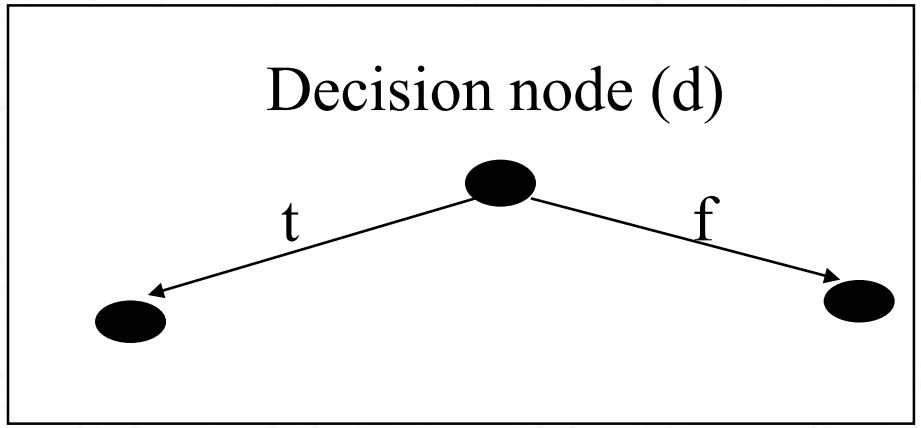
- Calculate the cyclomatic complexity of the following byte code?

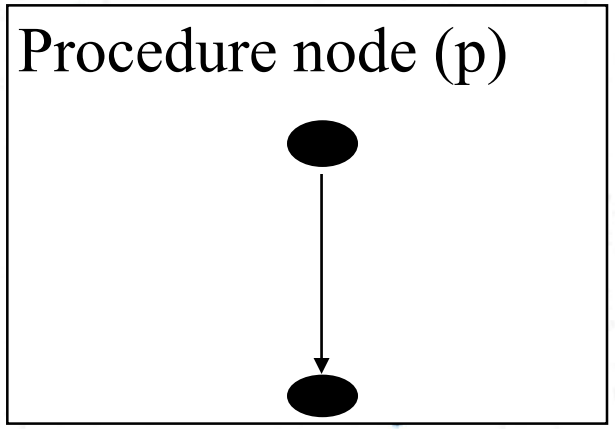
```
Method void D1(boolean, java.lang.String, java.lang.String)
 0  iload_0
 1  ifeq 14
 4  getstatic #2 <Field java.io.PrintStream out>
 7  aload_1
 8  invokevirtual #3 <Method void println(java.lang.String)>
11 goto 21
14 getstatic #2 <Field java.io.PrintStream out>
17 aload_2
18 invokevirtual #3 <Method void println(java.lang.String)>
21 return
```

Control Flow Graph Rules

1. To represent a start or a stop node use the notation 
2. To represent an intermediary node use the notation 
3. Start node, stop node, decision nodes and true/false paths have to be labeled.
4. Edges should always indicate the directions.
5. Along with a start node, procedure nodes and decision can also be represented.
6. A procedure node represent one or more non-decisional statements.

Intermediary Nodes in a Control Flow graph





Control Flow Graph Rules

3. Start node, stop node, decision nodes and true/false paths have to be labeled.

4. Edges should always indicate the directions.

Control Flow Graph Rules – Rule Five

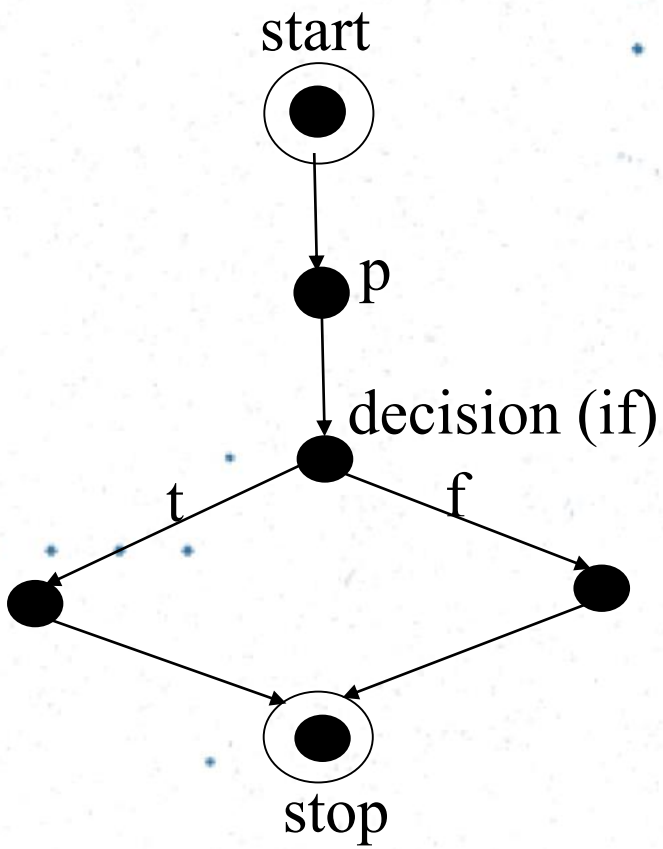
5. Along with a start node, procedure nodes and decision can also be represented.

```
int p;
if(p <10)
    System.out.println("Value of p is less than 10");
else
    System.out.println("Value of p is a grater than or equal to 10");
```

Control Flow Graph Rules – Rule Five

5. Along with a start node, procedure nodes and decision can also be represented.

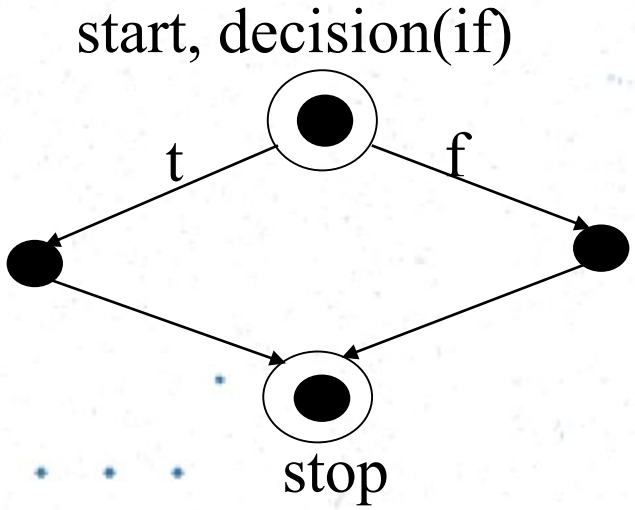
```
int p;
if(p <10)
    System.out.println("Value of p is less than 10");
else
    System.out.println("Value of p is a grater than or equal to 10");
```



Control Flow Graph Rules – Rule Five

5. Along with a start node, procedure nodes and decision can also be represented.

```
int p;
if(p <10)
    System.out.println("Value of p is less than 10");
else
    System.out.println("Value of p is a grater than or equal to 10");
```



Control Flow Graph Rules – Rule Six

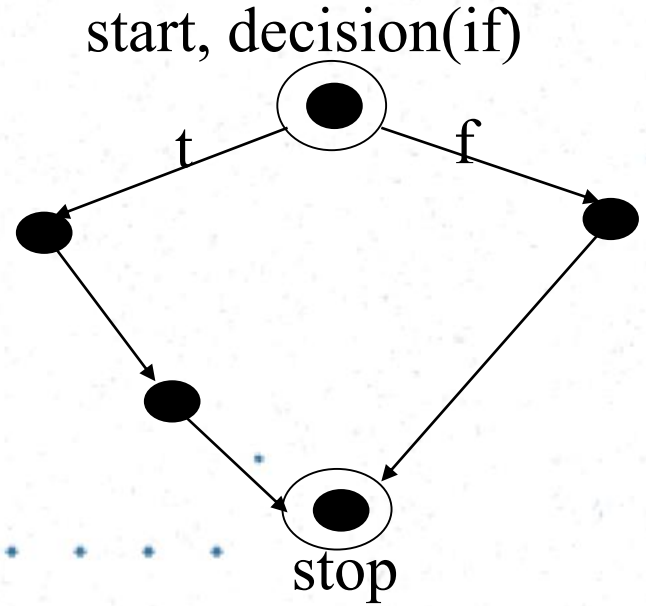
6. A procedure node represent one or more non-decisional statements.

```
int p;
if(p <10)
    System.out.println("Current value of p is " + p);
     p = p + 1;
else
    System.out.println("Value of p is a grater than or equal to 10");
```

Control Flow Graph Rules – Rule Six

6. A procedure node represent one or more non-decisional statements.

```
int p;
if(p <10)
    System.out.println("Current value of p is " + p);
    p = p + 1;
else
    System.out.println("Value of p is a grater than or equal to 10");
```



V(G) = e – n + 2

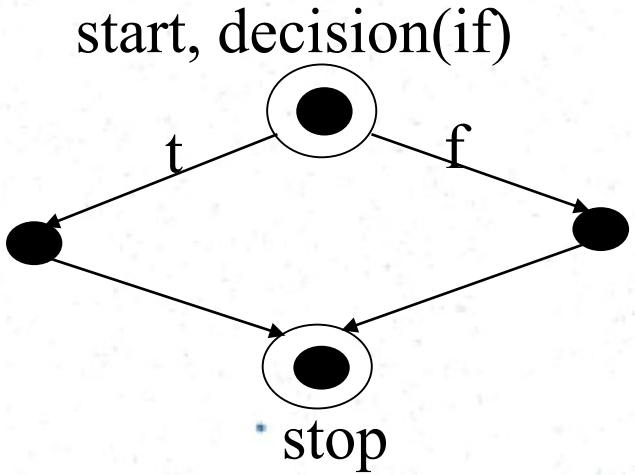
= 5 – 5 + 2

= 2

Control Flow Graph Rules – Rule Six

6. A procedure node represent one or more non-decisional statements.

```
int p;
if(p <10)
    System.out.println("Current value of p is " + p);
    p = p + 1;
else
    System.out.println("Value of p is a grater than or equal to 10");
```

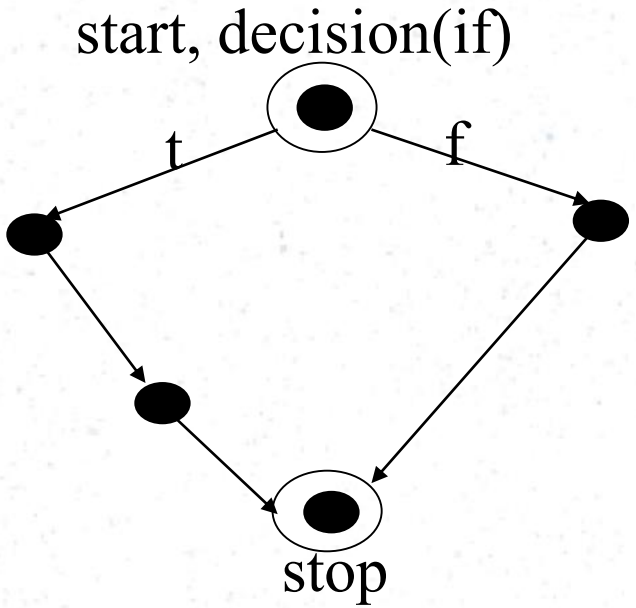


$V(G) = e - n + 2$

$= 4 - 4 + 2$

$= 2$

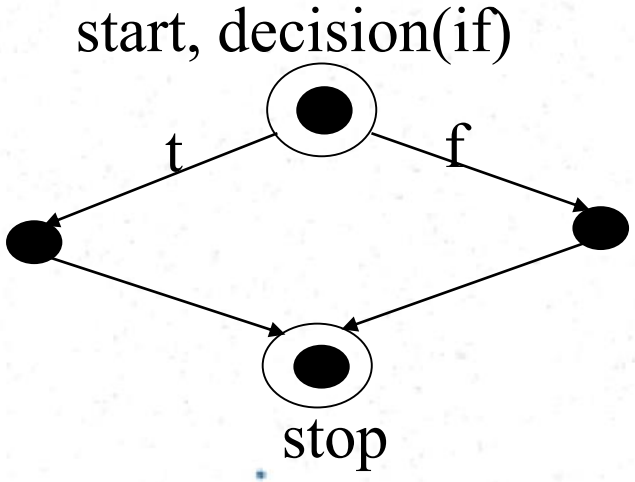
Do procedure nodes have an impact on the CC value??



$V(G) = e - n + 2$

$= 5 - 5 + 2$

$= 2$



$V(G) = e - n + 2$

$= 4 - 4 + 2$

$= 2$

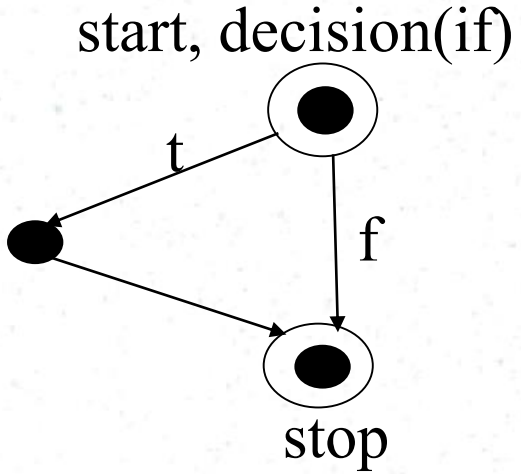
Questions??

- Draw the control flow graphs for the following code segments and calculate the cyclomatic complexity.

1)	public static void D0 (boolean a, String x){ if(a) System.out.println("x"); }
2)	public static void D1 (boolean a, String x, String y){ if(a) System.out.println("x"); else System.out.println("y"); }

Answer

1)	public static void D0 (boolean a, String x){ if(a) System.out.println("x"); }



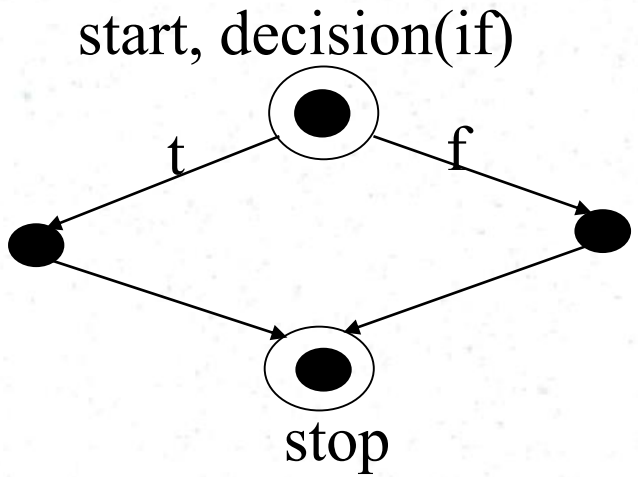
V(G) = e – n + 2

= 3 – 3 + 2

= 2

Answer

2)	public static void D1 (boolean a, String x, String y) { if(a) System.out.println("x"); else System.out.println("y"); }



V(G) = e – n + 2

= 4 – 4 + 2

= 2

Questions??

- Draw the control flow graphs for the following code segments and calculate the cyclomatic complexity.

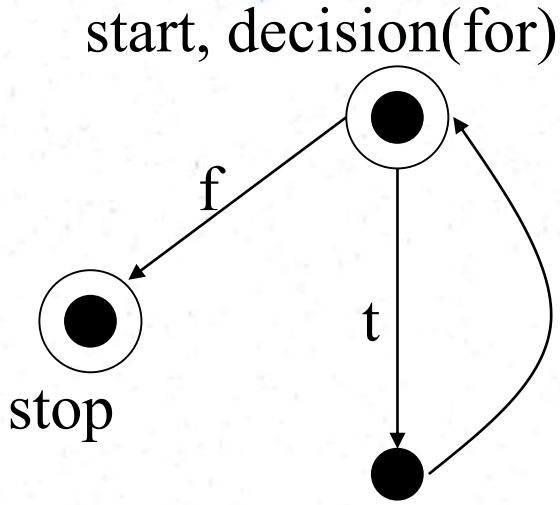
3)	public static void D3(int m, String x) { for(int i=0; i<m; i++) System.out.println("x"); }

4)	public static void D3(int a, String x) { do { System.out.println("x"); a++; } while (a <10) }

Answer

3)

```
public static void D3(int m, String x) {
for(int i=0; i<m; i++)
    System.out.println("x");
}
```



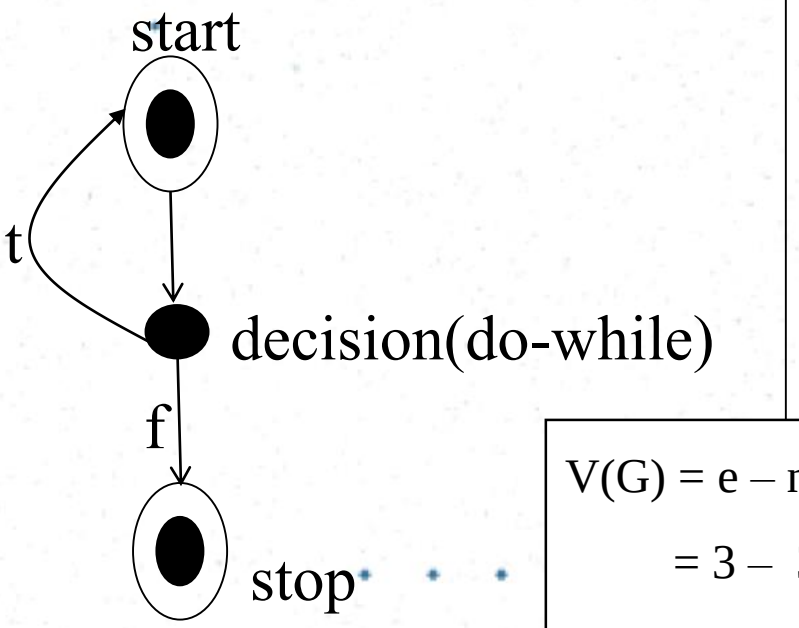
V(G) = e – n + 2

= 3 – 3 + 2

= 2

Answer

4)	public static void D3(int a, String x) { do { System.out.println("x"); a++; } while (a <10) }



V(G) = e – n + 2

= 3 – 3 + 2

= 2

Question??

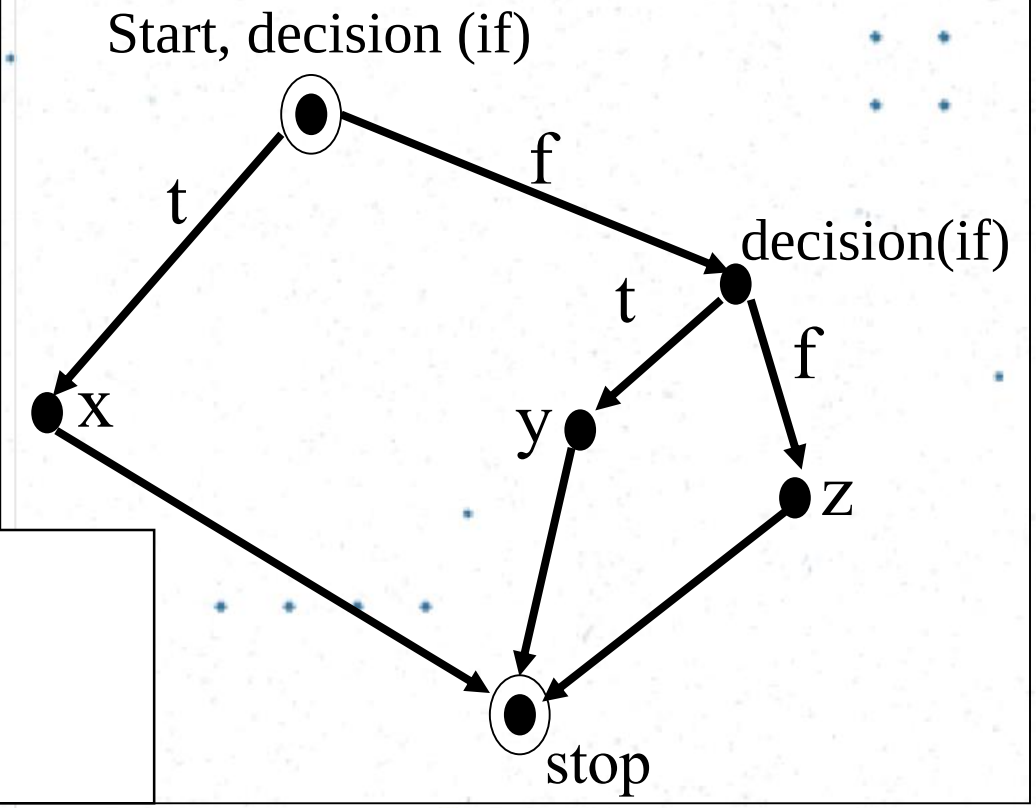
- Draw the control flow graph for the following code segment and calculate the cyclomatic complexity.

5)	void composite (boolean a, boolean b, String x, String y, String z) { if (a) System.out.println(x); else { if (b) System.out.println(y); else System.out.println(z); } }

Answer

5)

```
void composite (boolean a, boolean b, String x, String y, String z)
{
    if (a)
      System.out.println(x);
    else {
      if (b)
        System.out.println(y);
      else
        System.out.println(z);
      }
}
```



V(G) = e – n + 2

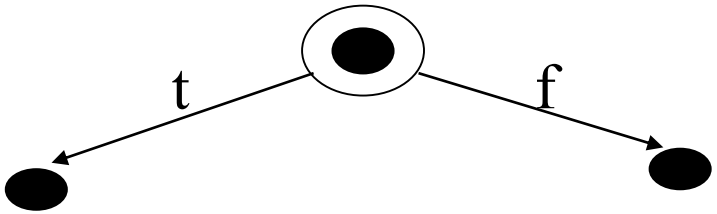
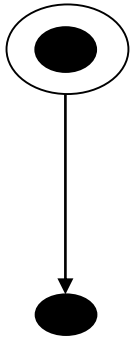
= 7 – 6 + 2

= 3

Question??

- From the V(G) = e – n + 2 equation, drive that V(G) = d + 1.

• Nodes in a control flow graph: ▪ Decision nodes (d) ▪ Procedure nodes (p) ▪ Start node ▪ Stop node	Total number of nodes in a control flow graph = d + p + 1

Decision node (d)  	Procedure node (p) 	Total edges in a control flow graph = 2d + 1p

Derivation

e = 2d + p

Where

e = number of edges

d = decision nodes

p = procedure nodes

n = d + p + 1

Where

n = number of nodes

$$V(G) = e - n + 2$$

$$= (2d + p) - (d + p + 1) + 2$$

$$= d + 1$$

Cyclomatic Complexity of a Class

Total Cyclomatic Complexity for a class (V_g)	=	Sum of the cyclomatic complexity of each method

$$V_g = \sum_{i=1}^{n} V(G_i)$$

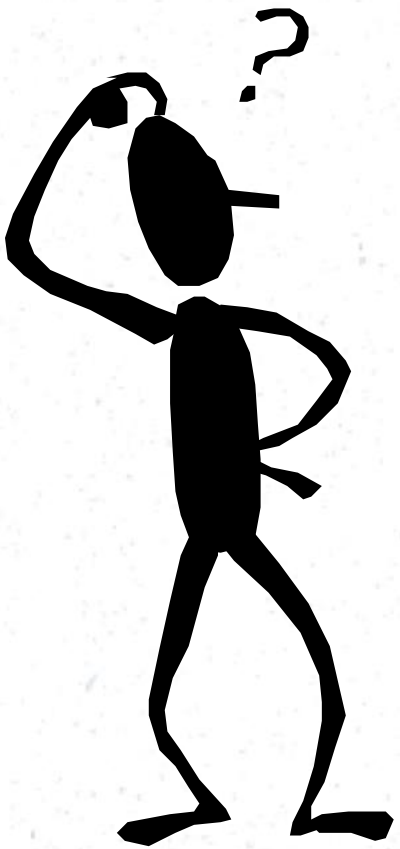
$$V_g = \sum_{i=1}^{n} (d_i + 1)$$

$$\mathbf{V_g = n + d_i}$$

Where:

- n = Number of methods in the class
- G_i = Flow graph for method **i**
- d_i = Number of decisions in method **i**

Programs with Compound Statements



- You can't expect the same cyclomatic complexity from all the approaches.
- The CC value obtained from the class file can be higher than CC obtained from the source file.

Question??

- Draw the control flow graph for the following code segment and calculate the cyclomatic complexity.

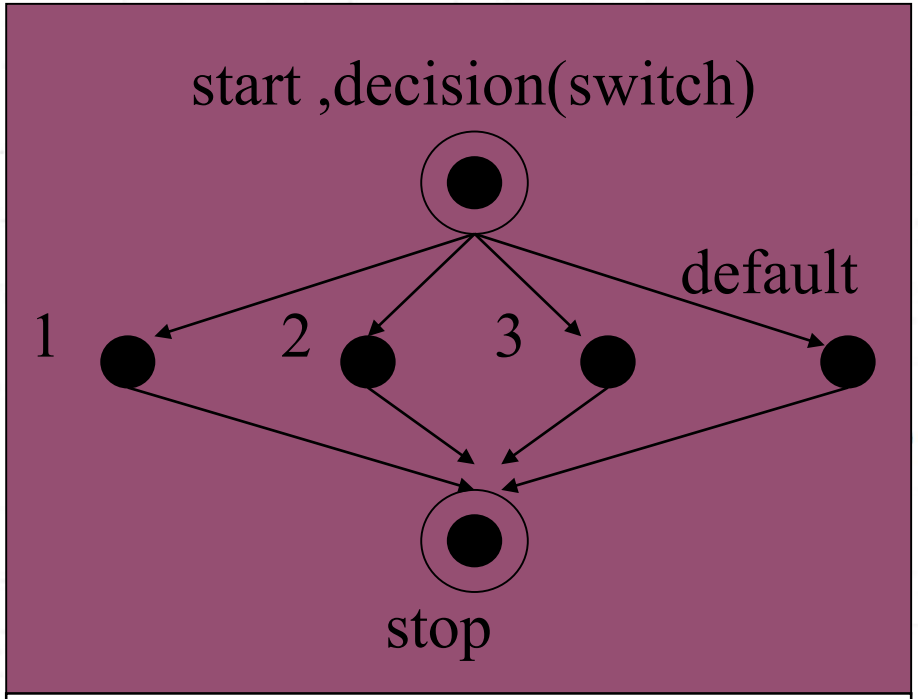
6)

```
public static void main (String[ ] args) {
    int i = 0;
    switch (i) {
            case 1: System.out.println("its 1");
                    break;
            case 2: System.out.println("its 2");
                    break;
            case 3: System.out.println("its 3");
                    break;
            default: System.out.println("its none");
                    break;
      }
  }
```

Answer

6)

```
public static void main (String[ ] args) {
    int i = 0;
    switch (i) {
            case 1: System.out.println("its 1");
                    break;
            case 2: System.out.println("its 2");
                    break;
            case 3: System.out.println("its 3");
                    break;
            default: System.out.println("its none");
                    break;
      }
  }
```



V(G) = e – n + 2

= 8 – 6 + 2

= 4

SLIIT

Discover Your Future

Object-Oriented Metrics

Weighted Composite Complexity (WCC) Measure

- Considers complexity as a multidimensional attribute.

- Based on four key factors.
  - Size
  - Type of control structures
  - Nesting level of control structures
  - Inheritance level of statements

Computing the WCC value

- WCCM value of a program = $\sum_{j=1}^{n} S_j * (W_t)_j$
  - Sj = Size of j^{th} executable statement in terms of token count
  - n = Total number of executable statements in a program
  - (Wt)j = Total weight of the j^{th} executable statement in the program

- Wt = Wc + Wn + Wi
  - Wc = Weight due to type of control structures
  - Wn = Weight due to nesting level of control structures
  - Wi = Weight due to inheritance level of statements

Question

- Identify the tokens and the size value of each statement of the following program:

Line No	Program Statements
1	public class Result{
2	public void outresult(int marks) {
3	if (marks > -1 && marks < 50)
4	System.out.println("Fail");
5	else
6	System.out.println("Pass");
	}
7	public static void main(String args[ ]){
8	Result r = new Result( );
9	r.outresult(50);
	}
	}

Answer

Line No	Program Statements	Tokens	Size (S)
1	public class Result{		
2	public void outresult (int marks) {	void, outresult( )	2
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8
4	System.out.println("Fail");	System, ., out, ., println( ), "Fail"	6
5	else		
6	System.out.println("Pass");	System, ., out, ., println( ), "Pass"	6
	}		
7	public static void main(String args[ ]){	void, main( )	2
8	Result r = new Result( );	Result, r, =, new, Result( )	5
9	r.outresult(50);	r, ., outresult( )	3
	}		
	}		

Weight Due to Type of Control Structures (Wc)

Type of control structure	Weight
Sequential	0
Branch	1
Iterative	2
Switch statement with **n** cases	n

Weight Due to Nesting Level of Control Structures (Wn)

Nesting Level of Statements	Weight
Sequential statements	0
Statements inside the outer most level/first level of control structures	1
Statements inside the second level control structures	2
Statements inside the third level control structures	3
Statements inside the n^{th} level control structures	n

Weight Due to Inheritance Level of Statements (Wi)

Inheritance Level of Statements	Weight
Statements inside the base class/root class	0
Statements inside the first derived class	1
Statements inside the second derived class	2
Statements inside the n^{th} derived class	n

Computing the WCC Value – Question

- Compute the WCC value of the following program:

Line No	Program Statements
1	public class Result{
2	public void outresult (int marks) {
3	if (marks > -1 && marks < 50)
4	System.out.println("Fail");
5	else
6	System.out.println("Pass");
	}
7	public static void main(String args[ ]){
8	Result r = new Result( );
9	r.outresult(50);
	}
	}

Computing the WCC Value – Answer

Line No	Program Statements	Tokens	S	Wc	Wn	Wi	Wt	WC
1	public class Result{							
2	public void outresult (int marks) {	void, outresult( )	2	0	0	1	1	2
3	if (marks > -1 && marks < 50)	if-else( ), marks, >, -1, &&, marks, <, 50	8	1	1	1	3	24
4	System.out.println("Fail");	System, ·, out, ·, println( ), "Fail"	6	0	1	1	2	12
5	else							
6	System.out.println("Pass");	System, ·, out, ·, println( ), "Pass"	6	0	1	1	2	12
	}							
7	public static void main(String args[ ]){	void, main( )	2	0	0	1	1	2
8	Result r = new Result( );	Result, r, =, new, Result( )	5	0	0	1	1	5
9	r. outresult(50);	r, ·, outresult( )	3	0	0	1	1	3
	}							
	}							
		WCC Value						**60**

SLIIT

Discover Your Future

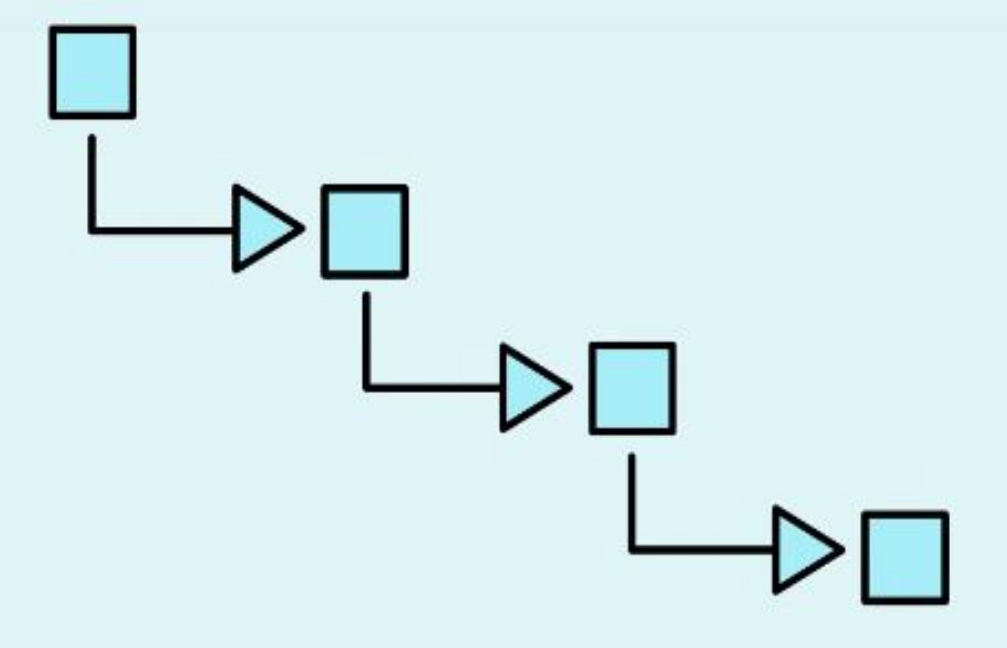
Paradigm Independent Metrics

Cognitive Functional Size (CFS) Measure

- Is a function of three fundamental factors:
  - Cognitive weights of Basic Control Structures (BCSs)
  - Number of inputs (Ni)
  - Number of outputs (No)

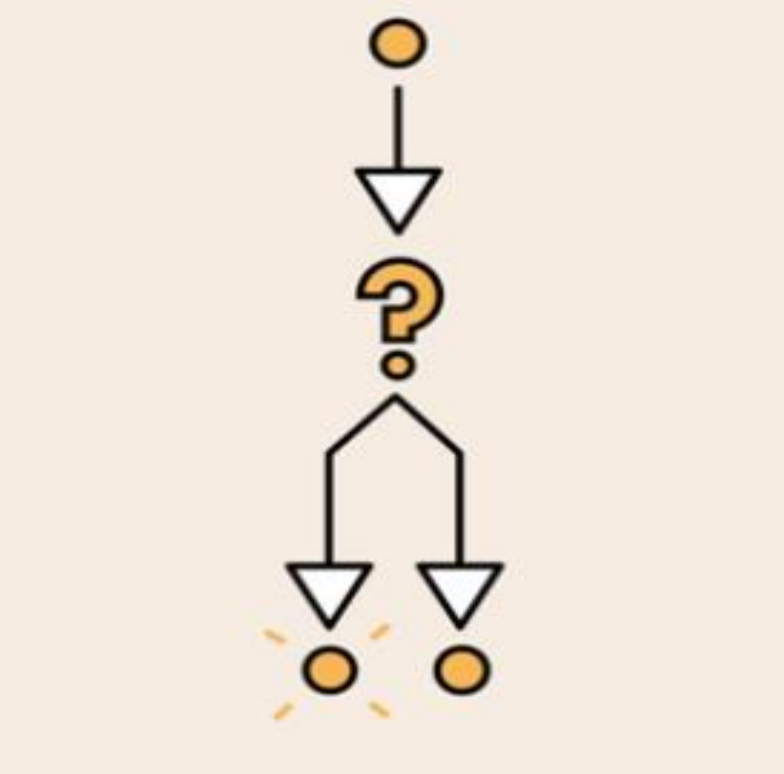
- The **cognitive weight** of software is the degree of difficulty or relative time and effort required for comprehending a given piece of software modelled by a number of BCSs.

Sequence Structures



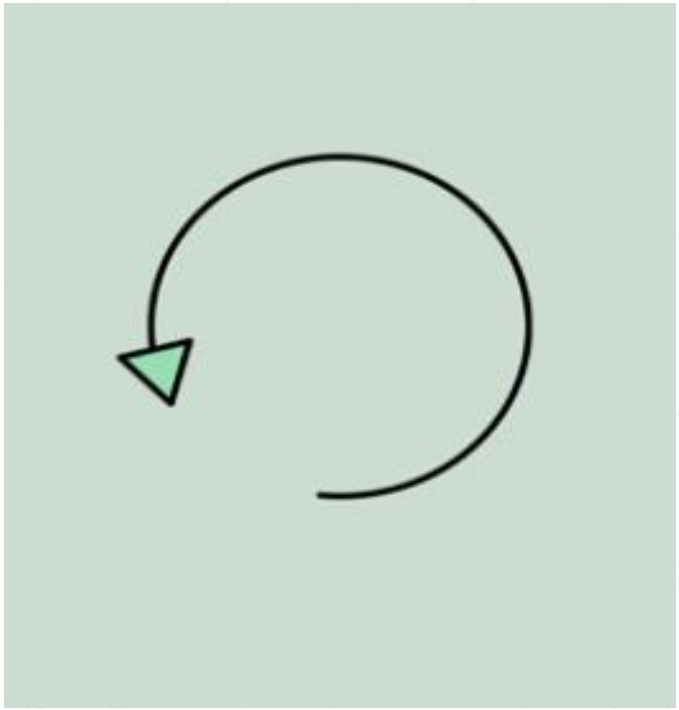
A sequence structure consists of a series of actions that is completed in a specific order. Action 1 is performed, then Action 2, then Action 3, etc., until all the actions in the sequence have been carried out.

Branch Structures



A branch or selection structure executes a certain piece of code only when a certain condition is met.

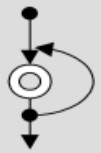
Iterative Structures



An iterative structure keeps on executing a certain piece of code until a certain condition is satisfied.

Basic Control Structures (BCSs)

Category	BCS	Structure	W_i
Sequence	Sequence (SEQ)		1
Branch	If-then-[else] (ITE)		2
	Case (CASE)		3

Category	BCS	Structure	W_i
Embedded component	Function call (FC)		2
	Recursion (REC)		3

Category	BCS	Structure	W_i
Iteration	For - do (R_i)		3
	Do - while (R_1)		3
	While-do (R_0)		3

Total Cognitive Weight

- The **total cognitive weight** of a software component, Wc, is defined as the sum of the cognitive weights of its q linear blocks composed of individual BCSs.
- Since each block may consist of *m* layers of nesting of BCSs, and each layer of *n* linear BCSs, Wc is calculated as follows:

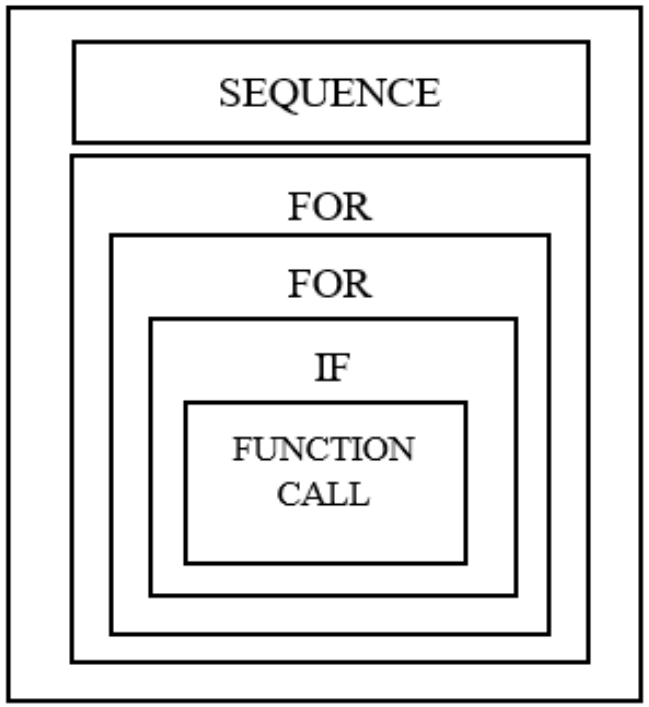
$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c\,(j,k,i) \right] \qquad (1)$$

Total Cognitive Weight

- If there is no embedded BCS in any of the q blocks, i.e., m=1, then (1) can be simplified as follows:

$$W_c = \sum_{j=1}^{q} \sum_{i=1}^{n} W_c(j,i) \qquad (2)$$

Calculating Total Cognitive Weight

Source Code	Structure of BCSs	Cognitive Weight
public void bubbleSort(){ int out, in; for(out=nElems-1; out>1; out--) for(in=0; in<out; in++) if(a[in] > a[in+1]) swap(in, in+1); }		$W_C = \sum_{j=1}^{q} [\prod_{k=1}^{m} \sum_{i=1}^{n} W_c\,(j,k,i)]$ $W_C = 1 + 3\,(\,3\,(\,2\,(\,2\,)\,)\,)$ $W_C = 1 + 36$ $W_C = 37$

Calculating Cognitive Weight – Question

- Calculate the cognitive weight of following code segment:

```
public static void main(String[] args) {

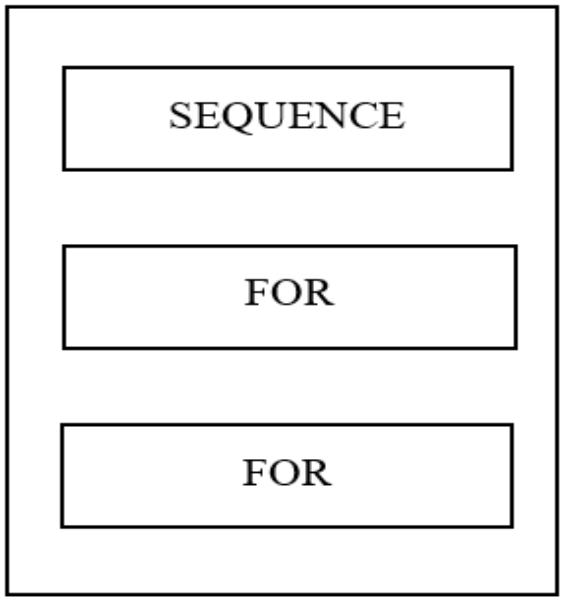
  String[] modules = {"SEPQM", "DS", "ESD", "AF", "SA"};

  for (int i = 0; i < modules.length; i++) {
        System.out.println(modules[i]);
  }

  System.out.println("In reverse order:");

  for (int i = modules.length - 1; i >= 0; i--) {
        System.out.println(modules[i]);
  }
}
```

Calculating Cognitive Weight – Answer

Source Code	Structure of BCSs	Cognitive Weight
public static void main(String[] args) { String[] modules = {"SEPQM", "DS", "ESD", "AF", "SA"}; for (int i = 0; i < modules.length; i++) { System.out.println(modules[i]); } System.out.println("In reverse order:"); for (int i = modules.length - 1; i >= 0; i--) { System.out.println(modules[i]); } }		$W_C = \sum_{j=1}^{q} \sum_{i=1}^{n} W_c(j,i)$ $W_C = 1 + 3 + 3$ $W_C = 7$

Cognitive Complexity of a Basic Component

- The cognitive functional size of a basic software component that only consist of **one method**, S_f, is defined as a product of the sum of inputs and outputs ($N_{i/o}$) and the total cognitive weight (Wc). More formally, it can be defined as follows:

$$S_f = (N_i + N_o) \times W_c \qquad (3)$$

Cognitive Complexity of a Complex Component

- Based on (3), the cognitive functional size of a complex software component with ***n*** **methods**, $S_f(c)$, is defined as follows:

$$S_f(c) = \sum_{c=1}^{n} S_f(c) \qquad (4)$$

Cognitive Complexity of a Component-Based Software System

- The cognitive functional size of a component-based software system, $\hat{S}$, with ***p* components**, $\hat{S}_f$, is defined as follows:

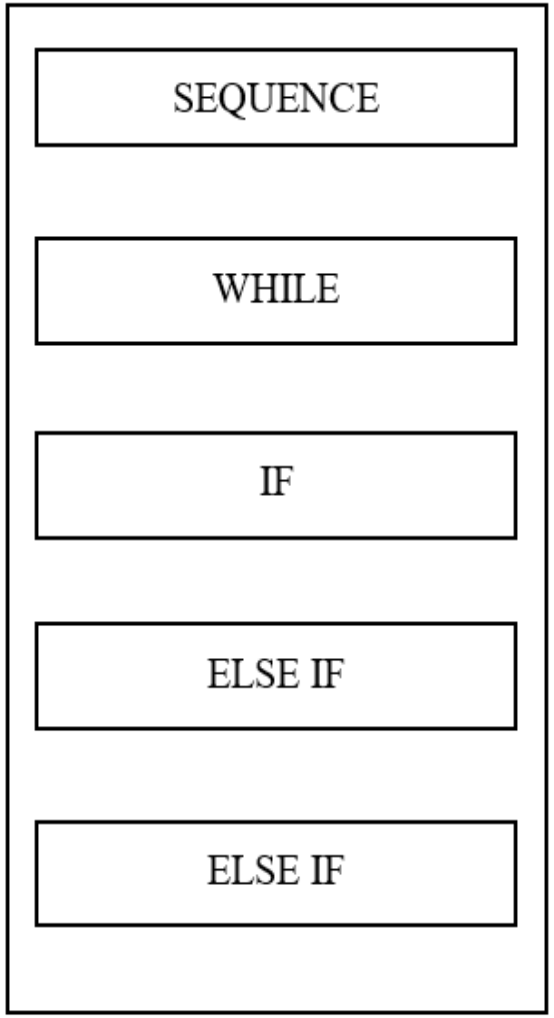
$$\hat{S}_f = \sum_{k=1}^{p} S_f(k) \qquad (5)$$

Calculating Cognitive Functional Size Value – Question

- Calculate the cognitive functional size of the following code segment:

```
import java.util.Scanner;
public class Results{
  public static void main(String[] args) {
    System.out.print("Enter your marks: ");
    Scanner sc = new Scanner(System.in);
    int marks = sc.nextInt();
    while(marks < 0 || marks > 100){
      System.out.print("Enter a valid mark: ");
      marks = sc.nextInt();
    }
    if (marks>75)
      System.out.println("A Pass");
    else if (marks<=75 && marks>65)
      System.out.println("B Pass");
    else if (marks<=65 && marks>45)
      System.out.println("C Pass");
    else
      System.out.println("Fail");
  }
}
```

Calculating Cognitive Functional Size Value – Answer

Source Code	Structure of BCSs	Cognitive Functional Size
import java.util.Scanner; public class Results{ public static void main(String[] args) { System.out.print("Enter your marks: "); Scanner sc = new Scanner(System.in); int marks = sc.nextInt(); while(marks < 0 || marks > 100){ System.out.print("Enter a valid mark: "); marks = sc.nextInt(); } if (marks>75) System.out.println("A Pass"); else if (marks<=75 && marks>65) System.out.println("B Pass"); else if (marks<=65 && marks>45) System.out.println("C Pass"); else System.out.println("Fail"); } }		$W_C = \sum_{j=1}^{q} [\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i)]$ $W_C = 1 + 3 + 2 + 2 + 2$ $W_C = 10$ $N_i = 2$ $N_o = 1$ (Only one S.O.P statement is excuted at a given time) $S_f = (N_i + N_o) \times W_c$ $S_f = (2 + 1) \times 10$ $S_f = 30$ [CWU]

WCC Measure – Computation Guidelines

1. Identification of the tokens begins **after** the **class declaration**.
2. In general, all the operators, keywords (except access flags such as public, static, etc.), strings, identifiers, and numerical values (including zero) are identified as separate tokens. However, they are exceptional cases.
3. All the characters inside a pair of single or double quotes are identified as a single token.
4. Along with the array name, the square brackets of an array are considered as one token.
5. Each comma operator that separates two program components from one another, is identified as a separate token.
6. Brackets are not identified as separate tokens.
7. In a program statement that contains a **variable declaration**, the variable name is not identified as a token. Hence, only the data type of the variable and comma operator that separates two variables are identified as tokens. On the other hand, if a program statement contains a **variable definition**, in addition to the data type of the variable and comma operator, the variable name is also identified as a token.

```
int x;
```

```
int x = 5;
```

8. In a method declaration or invocation, the round brackets and the method name are identified as one token. However, the components inside the round brackets of **user-defined methods** are not identified as tokens. Similarly, the components inside the round brackets of the **constructors of user-defined classes** are also not identified as tokens.
9. In a decisional statement, along with the keyword that defines the decisional type, the round brackets are identified as one token. Thus, **if( )** , **if-else( )**, **else-if( )**, **for( )**, **while( )**, **do-while( )**, and **switch( )** are identified as one token. However, the words **'else'** and **'do'** are not considered for the complexity calculation. In addition, the default :
10. The **'case :'** and **'default :'** in a **switch** statement are identified as separate tokens.
11. The keyword **'catch'** and the round brackets are identified as one operator. However, the word **'try'** is not considered for the complexity calculation.
12. The **'.' operator** that is used to connect classes, fields and/or methods is identified as a separate token. The **class**, **method**, or **field names** which are connected by '.' operators are also identified as separate tokens.
13. The statement terminator (;) is not identified as a token.
14. Manipulators such as **endl**, **"\n"** are considered as tokens.
15. The "∗" sign used in the declaration of pointer is not a dereference operator. It is just a notation that creates a pointer. Thus, it is not considered as a token.

16. The keyword **‘return’** is not considered as a token.

17. For a program which does not have a **built-in root class**, the weight allocation of the Wi attribute begins at **1**.

access flags (public, private, static, etc), do else, return, not tokens



Software Quality Management and Testing

Contents

- Quality Management Fundamentals
- Testing Fundamentals
- Test Planning and Documentation
- Test Execution and Reporting
- Challenges and Best Practices
- Software Testing as a Career Path



Quality Management Fundamentals

Quality Attributes

- Functionality: Functionality refers to the degree to which the software meets specified requirements and performs the intended tasks accurately and efficiently.

- Reliability: Reliability refers to the ability of the software to perform consistently and predictably under various conditions without failure or errors.

- Usability: Usability relates to how easy and intuitive the software is to use, including factors such as user interface design, navigation, and user experience.

Quality Attributes

- Performance: Performance refers to the responsiveness, speed, scalability, and efficiency of the software in executing its functions and handling user interactions.

- Security: Security involves protecting the software and its data from unauthorized access, manipulation, or breaches, ensuring confidentiality, integrity, and availability.

Quality Standards and Certification

- **ISO Standards**: ISO (International Organization for Standardization) develops and publishes international standards for various industries. ISO standards provide a framework for organizations to implement quality management systems and ensure consistency, efficiency, and safety in products and services. Some of the key ISO standards related to quality management include:
- ISO 9001: This standard specifies requirements for a quality management system (QMS) that helps organizations demonstrate their ability to consistently provide products and services that meet customer and regulatory requirements.
- ISO 14001: Focuses on environmental management systems, helping organizations develop and implement policies and procedures to minimize their environmental impact.
- ISO 27001: Deals with information security management systems (ISMS), ensuring organizations implement appropriate security controls to protect sensitive information.
- ISO 45001: Addresses occupational health and safety management systems, providing guidelines for organizations to ensure a safe and healthy work environment.

Quality Standards and Certification

- **CMMI (Capability Maturity Model Integration)**: CMMI is a process improvement approach that provides organizations with essential elements of effective processes. Developed by the Software Engineering Institute (SEI) at Carnegie Mellon University, CMMI helps organizations improve their processes and performance across various domains, including software development, product manufacturing, and service delivery. CMMI defines maturity levels ranging from Initial to Optimizing, with each level representing a higher degree of process maturity and capability.
- CMMI for Development: Focuses on improving processes related to product and service development.
- CMMI for Services: Tailored for organizations providing services rather than products.
- CMMI for Acquisition: Aimed at improving acquisition processes in organizations.

Quality Standards and Certification

- **Six Sigma**: Six Sigma is a data-driven methodology aimed at improving processes by identifying and eliminating defects or variations. Originally developed by Motorola in the 1980s and popularized by companies like General Electric, Six Sigma focuses on reducing variability and improving process efficiency. The term "Six Sigma" refers to a statistical concept representing a process performance goal of no more than 3.4 defects per million opportunities. Six Sigma methodology typically follows the DMAIC (Define, Measure, Analyze, Improve, Control) or DMADV (Define, Measure, Analyze, Design, Verify) framework for process improvement.
- Green Belt, Black Belt, and Master Black Belt certifications are common in Six Sigma, indicating different levels of expertise in applying the methodology to process improvement projects.

Quality Assurance vs. Quality Control

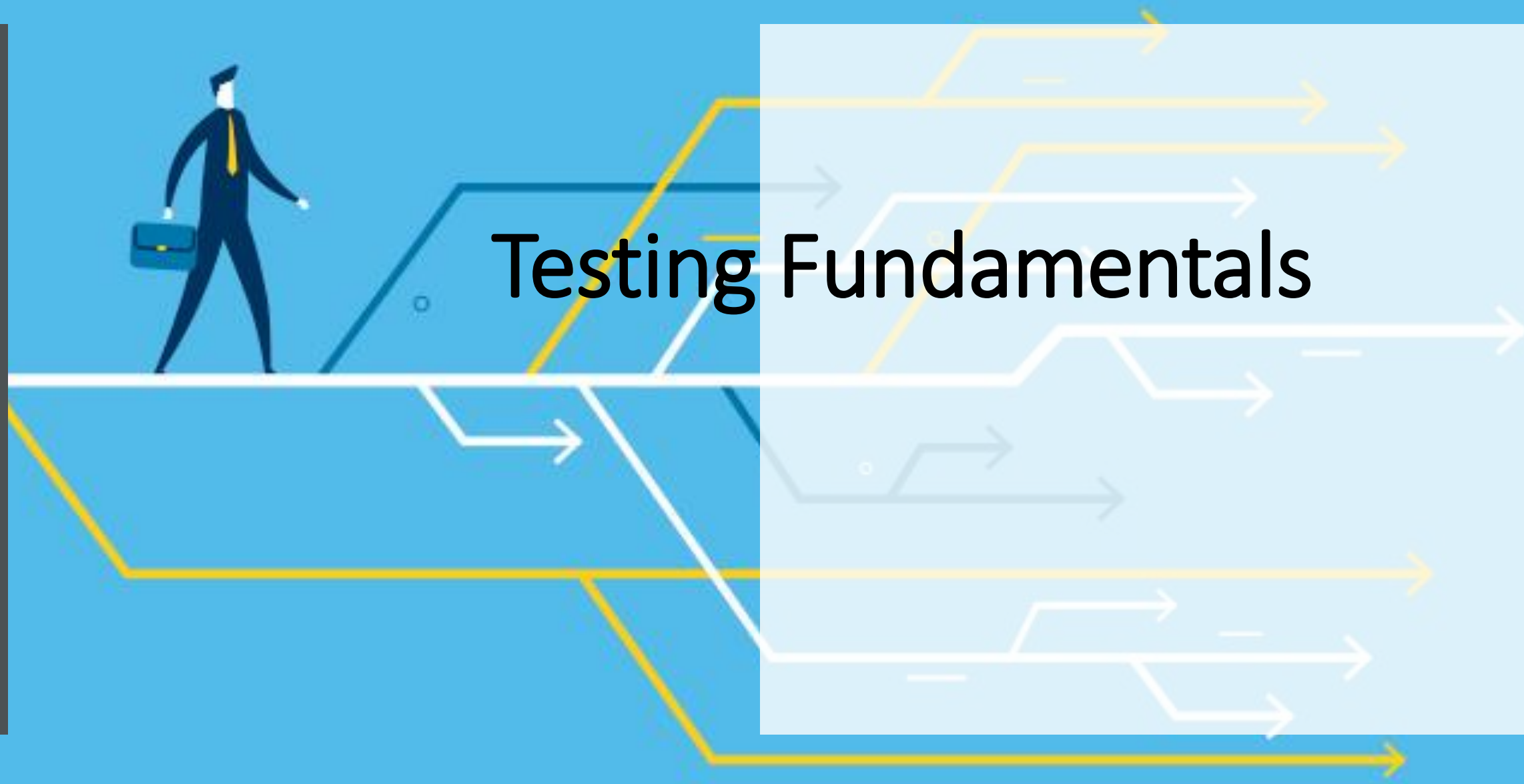
In simple terms:

- Quality Assurance (QA):

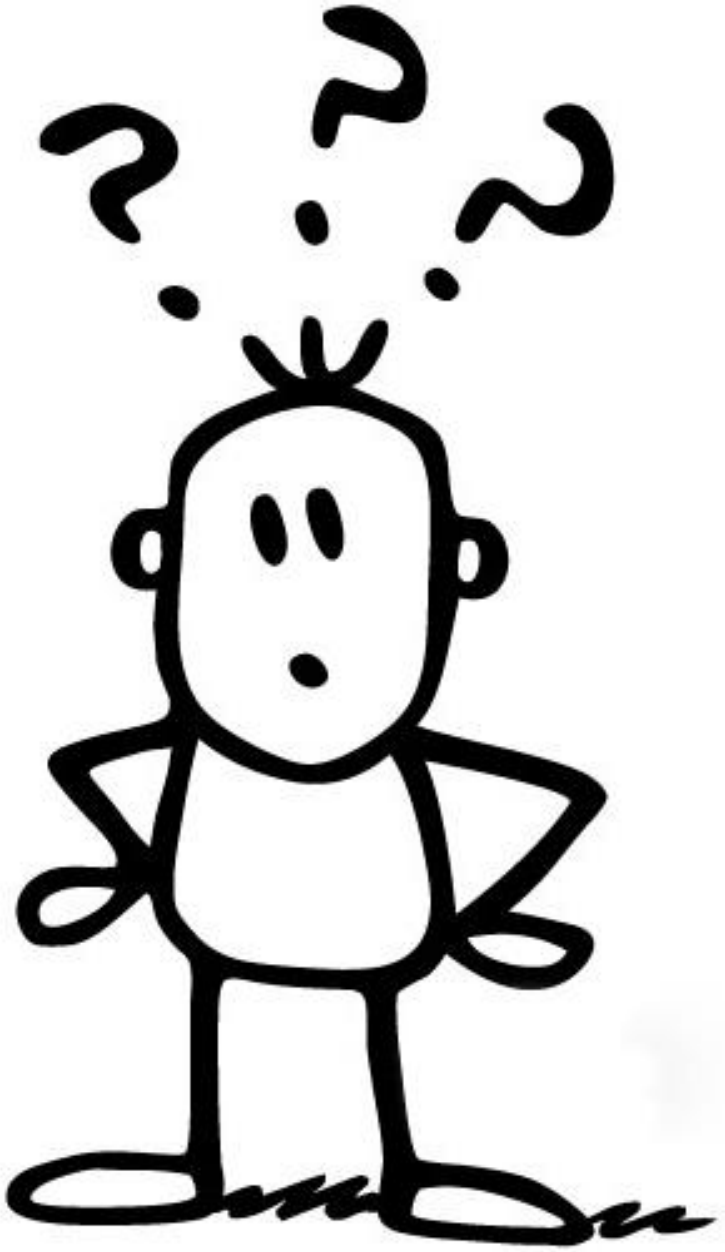
  QA ensures that processes are in place to **prevent defects from occurring in the first place**. It's about setting up systems and standards to ensure quality at every stage of production or service delivery.

- Quality Control (QC):

  QC involves checking the end product or service to ensure that it meets the predefined standards and specifications. It's the inspection and testing process that **identifies and corrects defects after they occur.**



Testing Fundamentals

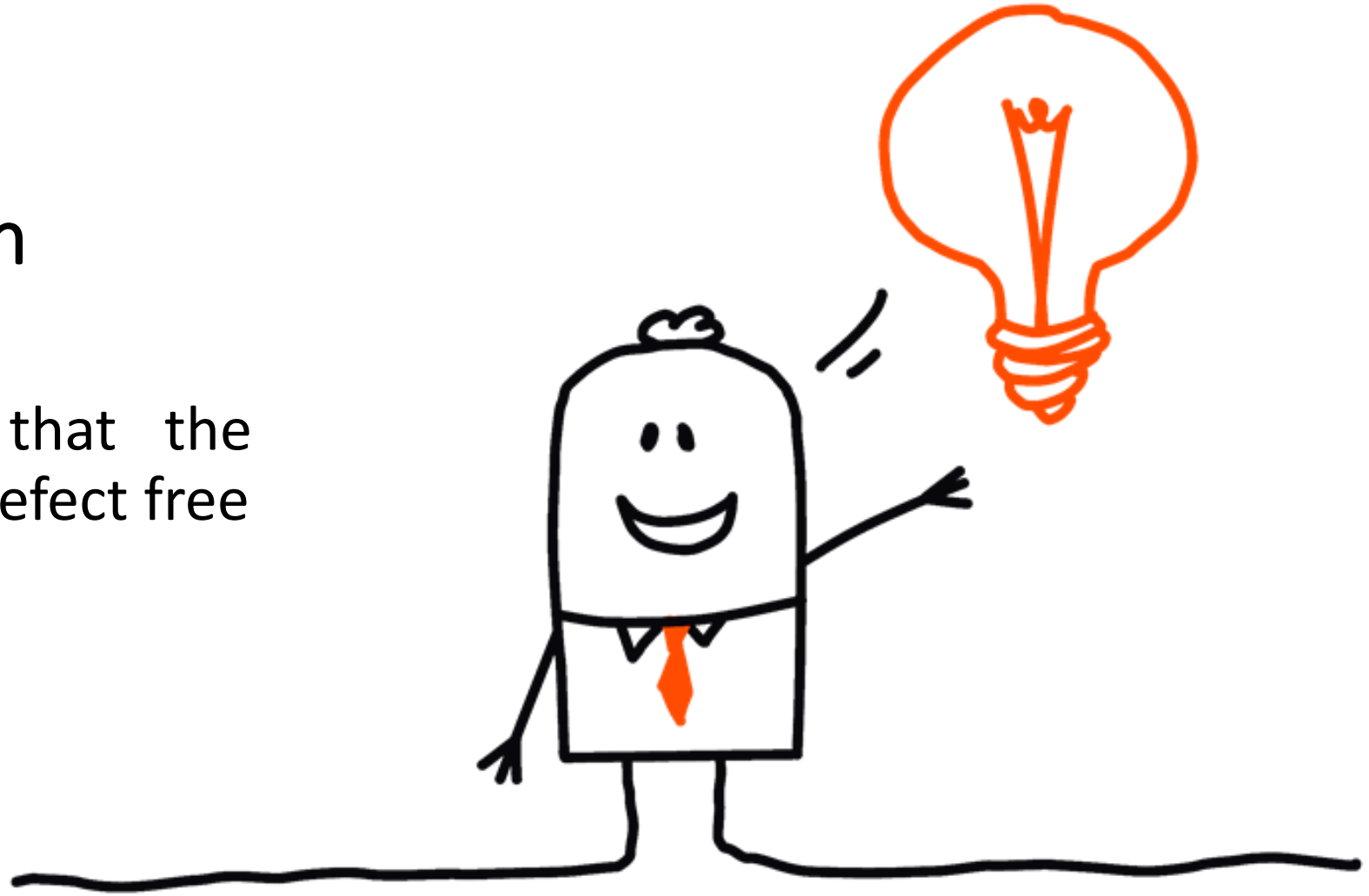


What is Software Testing?

- Software testing is a process used to identify the correctness, completeness and quality of a developed computer software.
- It is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is defect free.
- Software testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements.
- It includes a set of activities conducted with the intent of finding errors in software, so that it could be corrected before the product is released to the end users.

Simple Definition

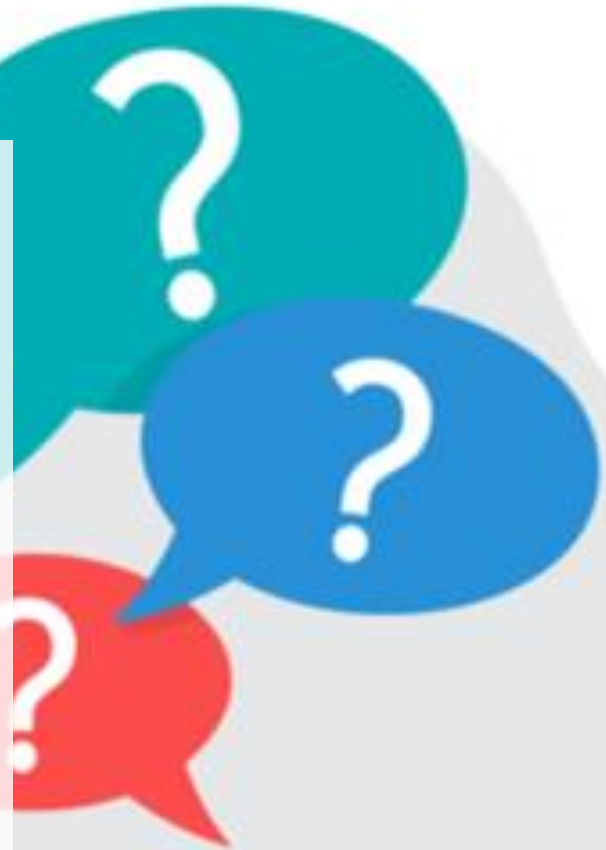
Activity to check that the software system is defect free



Why is Software Testing Important?

- Testing is important because software bugs could be expensive or even dangerous.

- Software bugs can potentially cause monetary and human loss.



Space:Ariane5





On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off.

It turned out that the cause of the failure was a software error in the inertial reference system.

Radiation Therapy Machine Malfunction



In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.

Banking Software Error

In May 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.



China Airlines Flight Crash



China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocent lives.

Static Testing Vs Dynamic Testing

Static Testing

- Detects defects directly by reviewing software artifacts like requirements and code without executing them.
- Applicable to non-executable work products such as requirements documents or design specifications.
- Can measure quality characteristics like maintainability without needing to execute code.
- Example; Code reviews, Requirement analysis, Static Code Analysis

Dynamic Testing

- Uncovers defects through executing the software and analyzing failures.
- Essential for detecting defects that manifest during runtime.
- Limited to executable work products like compiled code or running applications.
- Measures quality characteristics like performance efficiency that require code execution.
- Example; Unit Testing, Integration Testing, System Testing

Types of test activities and their significance

Functional Testing

- Unit Testing
- Smoke Testing
- Regression Testing
- Sanity Testing
- System Testing
- User Acceptance Testing

Types of test activities and their significance

Non-Functional Testing

- Performance Testing
- Compatibility Testing
- Usability Testing
- Security Testing
- Localization Testing

Software Testing Principles

- It is important that you achieve optimum test results while conducting software testing without deviating from the goal.
- To get the advantageous result we need to follow some basic testing principles.
- There are ten common testing principles that are widely practiced in the software industry.

Testing Principles

1. A necessary part of a test case is a definition of the expected output or result.
2. A programmer should avoid attempting to test his or her own program.
3. A programming organization should not test its own programs.
4. Any testing process should include a thorough inspection of the results of each test.
5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.



Testing Principles

6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7. Avoid throwaway test cases unless the program is truly a throwaway program.
8. Do not plan a testing effort under the tacit assumption that no errors will be found.
9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10. Testing is an extremely creative and intellectually challenging task.



A necessary part of a test case is the definition of the expected output or result

- If the expected result of a test case has not been predefined, chances are that a plausible, but erroneous, result will be interpreted as a correct result because of the phenomenon of ‘‘the eye seeing what it wants to see.’’
- Therefore, a test case must consist of two components:
  - A description of the input data to the program.
  - A precise description of the correct output of the program for that set of input data.

A programmer should avoid attempting to test his or her own program

- After a programmer has constructively designed and coded a program, it is extremely difficult to suddenly change his/her perspective to look at the program with a destructive eye.
- A programmer may subconsciously avoid finding errors for the fear of retribution from peers or a supervisor, a client, or the owner of the program or system being developed.
- A program may contain errors due to the programmer's misunderstanding of the problem statement or specification. If this is the case, it is likely that the programmer will carry the same misunderstanding into tests of his or her own program.

A programming organization should not test its own programs

- It is easy to measure time and cost objectives, whereas it is extremely difficult to quantify the reliability of a program.

- Therefore, it is difficult for a programming organization to be objective in testing its own programs, because the testing process, if approached with the proper definition, may be viewed as decreasing the probability of meeting the schedule and the cost objectives.

Any testing process should include a thorough inspection of the results of each test

- From numerous experiments it was seen that many subjects failed to detect certain errors, even when symptoms of those errors were clearly observable on the output listings.
- Errors that are found in later tests were often missed in the results from earlier tests.

Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected

- When testing a program there is a natural tendency to concentrate only on the valid and expected input conditions and to neglect the invalid and unexpected conditions.
- Many errors that are suddenly discovered in production software turn up when it is used in some new or unexpected way.
- It is hard, if not impossible, to define all the use cases for software testing. Therefore, test cases representing unexpected and invalid input conditions seem to have a higher error detection than the test cases for valid input conditions.

Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do

- This is a corollary to the previous principle. Programs must be examined for unwanted side effects.

- Example:

  A payroll program that produces the correct paychecks is still an erroneous program if it also produces extra checks for nonexistent employees, or if it overwrites the first record of the personnel file.

Avoid throwaway test cases unless the program is truly a throwaway program

- The major issue is that test cases represent a valuable investment.
- Some test cases disappears after the testing has been completed once. Whenever the program has to be tested again the test cases must be reinvented.
- Since this reinvention requires a considerable amount of work, people tend to avoid it.
- Therefore, the retest of the program is rarely as rigorous as the original test, meaning that if the modification causes a previously functional part of the program to fail, this error often goes undetected.

Do not plan a testing effort under the tacit assumption that no errors will be found

- This is a mistake project managers often make and is a sign of the use of the incorrect definition of testing.
- It is impossible to develop a program that is completely error free. Even after extensive testing and error correction, it is safe to assume that errors still exist.

The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section

- Errors tend to come in clusters and in a typical program some sections seem to be much more prone to errors than other sections, although nobody has supplied a good explanation of why this occurs.
- The phenomenon is useful in that it gives us insight or feedback in the testing process.
- If a particular section of a program seems to be much more prone to errors than other sections, then this phenomenon tells that, in terms of yield on our testing investment, additional testing efforts are best focused against this error-prone section.

Testing is an extremely creative and intellectually challenging task

- It is probably true that the creativity required in testing a large program exceeds the creativity required in designing that program.
- It is impossible to test a program sufficiently to guarantee the absence of all errors.

Testing Methods

- The three primary human testing methods are code inspections, walkthroughs and user (or usability) testing.
- These methods can be used at virtually any stage of software development, after an application is deemed to be complete or as each module or unit is complete.
- Inspections and walkthroughs involve a team of people reading or visually inspecting a program. The objective is to find errors but not to find solutions to the errors—that is, to test, not debug.

Walkthrough

- Walkthrough in software testing is used to **review documents with peers, managers, and fellow team members who are guided by the author of the document to gather feedback and reach a consensus.**
- In a walkthrough, a group of developers (with three or four being an optimal number) performs the review. Only one of the participants is the author of the program.
- This review becomes more beneficial for those people who are away from software sphere and through this meeting they get a good insight about the product that is to be developed. The content is being explained by the author step by step to reach a common objective.

Walkthrough

- The audience is selected from different backgrounds in order to have a diverse point of view and thus, provide different dimensions to a common objective. This is not a formal process but is specifically used for high level documents like requirement specifications or functional specifications, etc.

- Activities included in walkthrough.
  - Describe and justify the contents of the document
  - Reach a common consensus on the document
  - Check and discuss the different solutions to a problem and different suggested alternatives

Code Inspections



- A code inspection is a set of procedures and error-detection techniques for group code reading.
- Most discussions of code inspections focus on the procedures, forms to be filled out, and so on.



Inspection Team

- An inspection team usually consists of four people. The first of the four plays the role of moderator, which in this context is tantamount to that of a quality-control engineer.

- The moderator is expected to be a competent programmer, but he or she is not the author of the program and need not be acquainted with the details of the program.

Inspection Team



- Moderator duties include,
  - Distributing materials and scheduling the inspection session.
  - Leading the session.
  - Recording all errors found.
  - Ensuring that the errors are subsequently corrected.
- The second team member is the programmer. The remaining team members usually are the program's designer (if different from the programmer) and a test specialist. The specialist should be well versed in software testing and familiar with the most common programming errors.

Inspection Agenda



- Several days in advance of the inspection session, the moderator distributes the program's listing and design specification to the other participants.
- The participants are expected to familiarize themselves with the material prior to the session.
- Two activities are there during the session.

Inspection Agenda

- The programmer narrates, statement by statement, the logic of the program. During the discourse, other participants should raise questions, which should be pursued to determine whether errors exist. It is likely that the programmer, rather than the other team members, will find many of the errors identified during this narration. In other words, the simple act of reading aloud a program to an audience seems to be a remarkably effective error-detection technique.

- The program is analyzed with respect to checklists of historically common programming errors.

Inspection Agenda

- The moderator is responsible for ensuring that the discussions proceed along productive lines and that the participants focus their attention on finding errors, not correcting them.

- Upon the conclusion of the inspection session, the programmer is given a list of the errors uncovered. If more than a few errors were found, or if any of the errors require a substantial correction, the moderator might make arrangements to reinspect the program after those errors have been corrected. This subsequent list of errors is also analyzed, categorized, and used to refine the error checklist to improve the effectiveness of future inspections.

Benefits of the Inspection Process

- The programmer usually receives valuable feedback concerning programming style, choice of algorithms and programming techniques.
- The other participants gain in a similar way by being exposed to another programmer's errors and programming style.
- Helps reinforce a team approach to the current project and to projects that involve these participants in general.
- The inspection process is a way of identifying early the most error-prone sections of the program, helping to focus attention more directly on these sections during the computer-based testing processes





Test Planning and Documentation

Test Planning process

- Define the purpose and scope of testing, including goals, timelines, resources, and constraints.
- **Risk Assessment**: Identify potential risks to the project, prioritize them based on severity and likelihood, and develop mitigation strategies.
- **Resource Allocation**: Determine the required resources, including personnel, tools, and infrastructure, for testing activities.
- **Test Environment Setup**: Prepare the testing environment, including hardware, software, and configurations required for testing.
- **Test Planning Documentation**: Document the test plan, outlining the overall approach, test objectives, scope, schedule, roles and responsibilities, and exit criteria.



Test Strategy and Approach

- **Test Strategy**: Define the overall approach to testing, including testing levels (e.g., unit, integration, system, acceptance), techniques, and methodologies to be employed.
- **Test Approach**: Specify how testing will be conducted, considering factors such as test design techniques, test automation, test data management, and defect tracking.



Test plan Documentation

- **Test Plan**: Document outlining the objectives, scope, approach, schedule, resources, and deliverables of the testing effort.
- **Components**: Include sections on introduction, test objectives, scope, approach, test deliverables, test environment, test schedules, resource allocation, risk management, and exit criteria.
- **Review and Approval**: Ensure the test plan is reviewed by stakeholders, including project managers, developers, and testers, and obtain approval before execution.



Test Case Design and Documentation

- **Test Case Design**: Develop test cases based on requirements, specifications, and design documents, covering both functional and non-functional aspects.
- **Test Case Attributes**: Each test case should include a unique identifier, description, preconditions, test steps, expected results, actual results, and pass/fail criteria.
- **Test Case Traceability**: Establish traceability between test cases and requirements, ensuring that each requirement is adequately covered by one or more test cases.
- **Test Case Prioritization**: Prioritize test cases based on factors such as criticality, risk, and frequency of use.



Traceability Matrices

- **Requirement Traceability Matrix (RTM)**: Cross-references requirements with test cases, ensuring comprehensive coverage and validation of all requirements.
- **Test Case Traceability Matrix (TCTM)**: Maps test cases to requirements, providing visibility into which requirements are tested by each test case.
- **Defect Traceability Matrix (DTM)**: Links defects to the corresponding test cases, requirements, and other relevant information, facilitating defect management and resolution.





Test Execution and Reporting

Test Execution Process

1. **Test Environment Setup**: This involves setting up the necessary hardware, software, and network configurations required to execute the tests. The test environment should mirror the production environment as closely as possible.
2. **Test Execution**: Testers execute the test cases according to the test plan. They record the actual results and any deviations from expected outcomes.
3. **Defect Reporting**: If any defects are identified during test execution, testers document them in a defect tracking system. Defects are typically categorized by severity and priority.
4. **Regression Testing**: After fixes are made to defects, regression testing is performed to ensure that the changes haven't introduced new defects or caused unintended side effects.
5. **Test Closure**: Once all test cases have been executed, and the exit criteria defined in the test plan have been met, the testing phase is closed. Test summary reports are prepared to document the testing activities, including test coverage, defects found, and any issues encountered.



Defect Management and Tracking

1. **Defect Identification**: Defects are identified during test execution when actual results deviate from expected results.
2. **Defect Logging**: Testers log defects in a defect tracking system, providing detailed information such as steps to reproduce, severity, priority, and any relevant attachments.
3. **Defect Triage**: Defects are triaged by a team consisting of testers, developers, and project managers. They prioritize defects based on severity and impact on the project.
4. **Defect Resolution**: Developers fix the defects identified during testing. Testers verify the fixes to ensure they address the reported issues.
5. **Defect Closure**: Once a defect has been verified as fixed, it is closed in the defect tracking system.



Test Reporting Mechanisms

1. **Test Summary Reports**: Test summary reports provide an overview of the testing activities performed during a specific phase or iteration. They include information such as test coverage, test execution status, defects found, and any outstanding risks or issues.
2. **Defect Reports**: Defect reports provide detailed information about each defect identified during testing. This includes the defect ID, description, steps to reproduce, severity, priority, status, and any related attachments.





Challenges and Best Practices

Challenges

1. **Changing Requirements**: Managing evolving requirements can lead to scope creep, affecting test coverage and stability.
2. **Resource Constraints**: Limited resources, including time, budget, and skilled personnel, can hinder thorough testing.
3. **Complex Systems**: Testing complex systems with intricate interdependencies poses challenges in identifying and addressing defects effectively.
4. **Testing Environment Setup**: Establishing and maintaining the necessary test environments can be time-consuming and error-prone.
5. **Test Data Management**: Generating and managing test data that accurately represents real-world scenarios can be challenging.



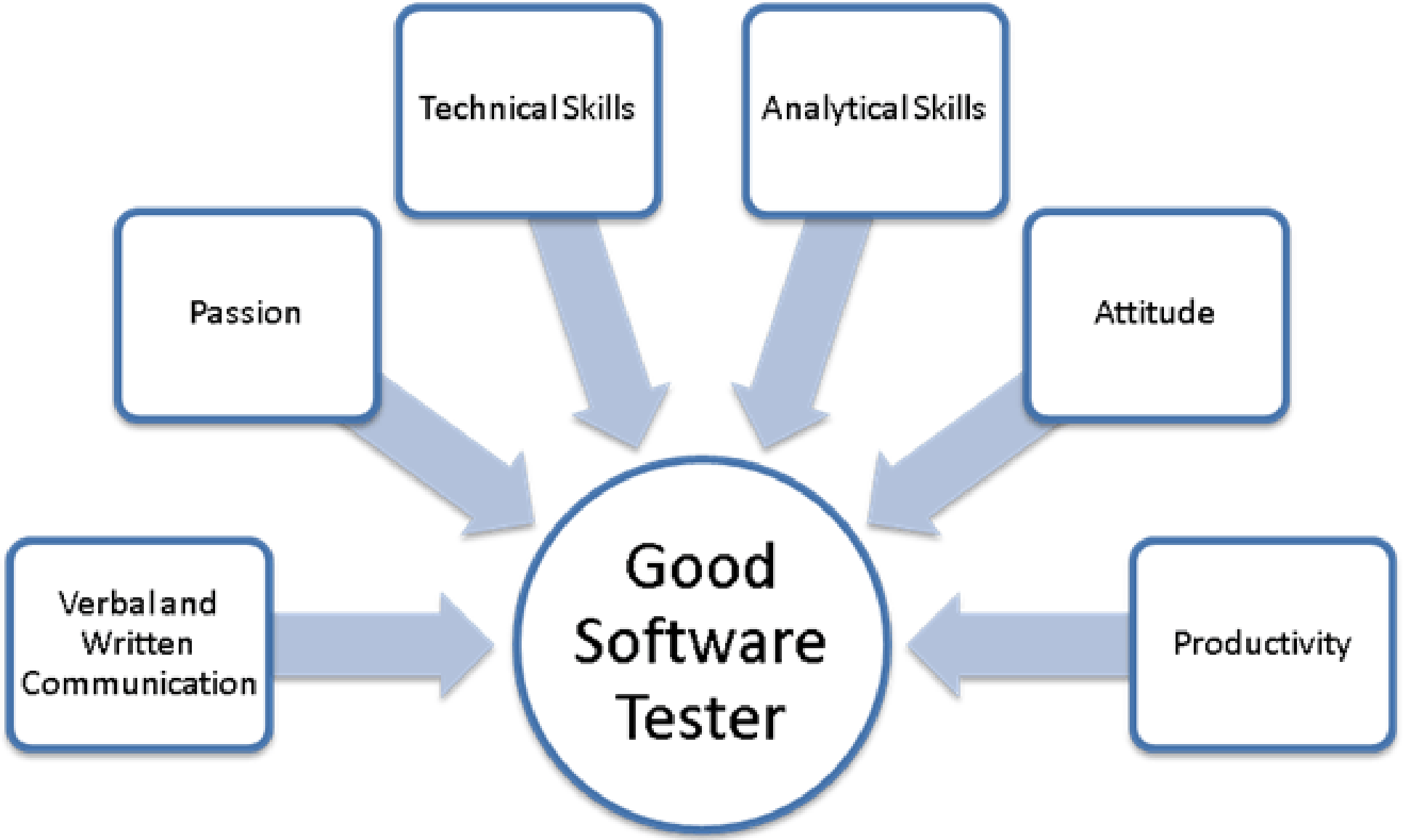
Best Practices to Overcome the Challenges

1. **Requirements Management**: Implement robust requirements management processes, including clear documentation, frequent reviews, and effective change management.
2. **Prioritization**: Prioritize testing efforts based on risk assessment, criticality, and business impact to optimize resource utilization.
3. **Test Automation**: Invest in test automation tools and frameworks to streamline repetitive tasks, increase test coverage, and accelerate testing cycles.
4. **Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines to automate the deployment and testing of software updates, ensuring rapid feedback and quality assurance.
5. **Collaboration and Communication**: Foster a culture of collaboration and open communication among stakeholders, including developers, testers, and business analysts, to facilitate knowledge sharing and alignment of objectives.





Software Testing as a Career Path


Technical Skills
Analytical Skills
Passion
Attitude
Verbal and Written Communication
Good Software Tester
Productivity

Skills required to become a Software Tester

Non-Technical Skills

- **Analytical skills**: A good software tester should have sharp analytical skills. Analytical skills will help break up a complex software system into smaller units to gain a better understanding and create test cases.

- **Communication skill**: A good software tester must have good verbal and written communication skills. Testing artifacts (like test cases/plans, test strategies, bug reports, etc.) created by the software tester should be easy to read and comprehend.



Skills required to become a Software Tester

Non-Technical Skills

- **Time Management & Organization Skills:** Testing at times could be a demanding job especially during the release of code. A software tester must efficiently manage workload, have high productivity, exhibit optimal time management, and organization skills.
- **Great Attitude:** To be a good software tester you must have a GREAT attitude. An attitude to 'test to break', detail orientation, willingness to learn and suggest process improvements.



Skills required to become a Software Tester

Technical Skills

- **Basic knowledge of Database/ SQL**

  Software systems have a large amount of data in the background. This data is stored in different types of databases like Oracle, MySQL, etc. in the backend. So, there will be situations when this data needs to be validated.

- **Basic knowledge of Linux commands:** Most of the software applications like web-services, databases, application servers are deployed on Linux machines. So it is crucial for testers to have knowledge about Linux commands.

Skills required to become a Software Tester

Technical Skills

- **Test Management -** Without proper test management techniques, software testing process will fail. Test management is nothing but managing your testing related artifacts.
- **Defect Tracking tool -** Defect tracking becomes necessary because the entire team should know about the defect including managers, developers, and testers. Several tools are used to lock defects including QC, Bugzilla, Jira, etc.
- **Automation tool -** Knowledge of any automation tool such as Seleniuim, Ranorex, etc.

Software Testing Careers

- Associate QA Engineer/Junior QA Engineer
- QA Engineer
- QA Automation Engineer
- Senior QA Engineer
- Associate QA Lead
- Lead QA
- Senior Lead QA
- QA Manager/Test Manager
- Senior Test Manager

SLIIT
Discover Your Future

Software Quality Assurance

Functional and Non-functional Testing

SQA - IT4100

Learning Outcomes

- Functional testing
- Non-functional testing

Software Testing



Testing Software



Test Outputs



Acceptability Check



Test Inputs

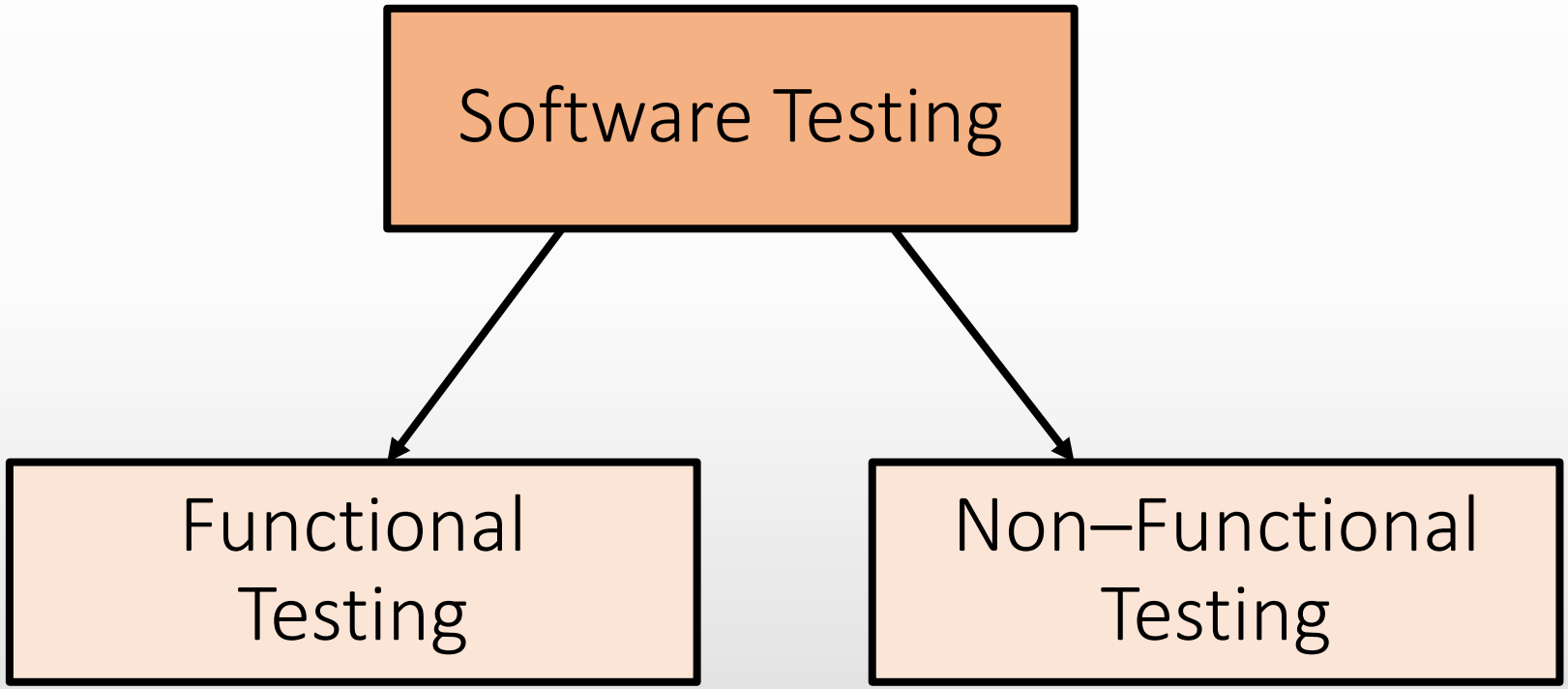


Test Case Pass



Test Case Fail

Software Testing Types



Functional Testing

Functional Testing

- A type of testing which verifies that each function of the software application operates in conformance with the requirement specification.
- Each functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results.

Unit Testing

- A process of testing the individual subprograms, subroutines, classes, or procedures in a program.
- Unit testing is a way of managing the combined elements of testing.
- Unit testing eases the task of debugging.



Test cases for unit testing



Unit Specification



Unit Source Code



Unit Testing

Smoke Testing

- Testing the core functionality of a program.
- The term smoke test in technology is broadly used to test product features in a limited time.



Smoke Testing

- Smoke testing is a subset of all defined test cases that cover the main functionality of a system.
- Examples of capabilities tested during the smoke test;
  - Access to the application
  - Logging in with a set of users
  - Main modules of a particular application

Regression Testing

- Regression testing is testing an existing software applications to ensure that a change or addition has not caused any errors with existing functionality.
- Regression testing re-runs the testing scenarios that were originally scripted.
- Regression testing typically require an automated testing tool.



Selecting regression testing

- Requires knowledge about the system and how it is affects by the existing functionalities.
- Select tests based on the area of frequent defects.
- Tests include the areas which has undergone code changes several times.
- Tests are selected based on the criticality of the features.

Sanity Testing

- Performed after receiving a software build, with minor changes in code or functionality.
- Checks whether the planned functionality is working as expected.
- Sanity testing is typically non-scripted.
- Sanity testing is a sub-set of regression testing.



System Testing

- System testing compare the entire system or program to its original objectives.
- Attempting to demonstrate how the entire system fails to meet its objectives.
- Requires a set of measurable objectives for the product.

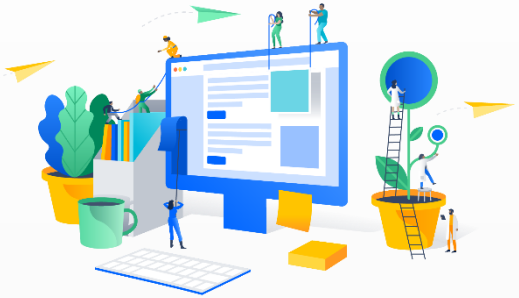


User Acceptance Testing

- Process of comparing the program to its initial requirements and the current needs of the end users.
- Usually is performed by the customer or end user.
- Developer will conduct user tests during the development cycle prior to delivering the finished product.



Example



TechSolutions



John Steve



Start Truck Cafe

Example

- Steve is the owner of Star Truck Café. He is a successful business man with many workers working under him. He has faced several difficulties in calculating the salaries of his employees.
- John is working in a company named as TechSolutions. This company develops software products for business requirements.
- Steve wanted an automated system to calculate the salaries of his employees at Star Truck Café. Therefore, he contacted John and signed a contract with TechSolutions to get an automated salary management system for his café.

Example - Specification

- Employees of Star Truck Café requires to work from 8.00a.m to 6.00p.m each day.
- Salary management system requires an attendance management system to record in time and out time of the employees using a fingerprint machine. This sub system should generate a report of total hours worked by an employee within a month.
- Employees can work overtime as well. If an employee works after 6.00p.m then that employee will have an overtime payment added to the monthly salary.

Example – Specification (Cont.)

- Overtime payment calculation is another sub system in salary management system. This sub system should calculate the overtime hours worked by an employee using the attendance management system and calculate the overtime payment of an employee.
- Salary calculation unit is the other sub system. In this sub system monthly salaries of all employees will be calculated by considering the attendance and overtime payments of the employee.
- As the owner of Star Truck Café, Steve should be able to login to the system and see all sub systems. All other users can login to the system and see his or her attendance details and salary details.

Example – Development of the System



Andrew



Oliver

Overtime Calculation

Attendance Management



Start Truck Café
Salary Management System

User Management

Salary Calculation



Jessica



Lilia

Unit Testing



Unit Specification



Unit Source Code



Unit Testing

Attendance Management – Unit Testing

- Attendance Management System

  Record in an out time of an employee using the fingerprint machine and generate a report of total hours worked by an employee within a month.

Oliver



Overtime Calculation – Unit Testing

- Overtime Calculation System

  Identify whether each employee has worked overtime and calculate the overtime payment that has to be added to each employee.

Andrew



User Management – Unit Testing

- User Management System

  When a user is logged in as the owner, that user can view all sub units of the system. When a user is logged in as a worker, that user can view his or her salary details and attendance details.

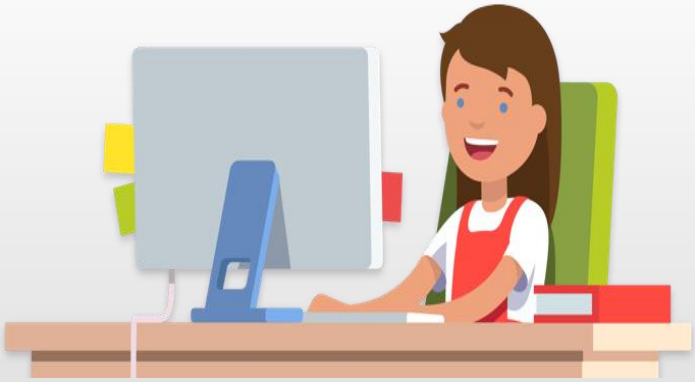
Jessica



Salary Calculation– Unit Testing

- Salary Calculation System

  Calculate the total monthly salaries of all employees by considering the attendance and overtime payments of each employee.



Lilia

Smoke Testing



Smoke Testing

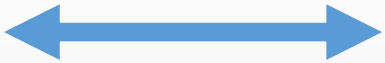
Smoke Testing	
	Handling user management
	Generating monthly salary for each employee

System Testing



System Testing







Start Truck Café
Salary Management System

Start Truck Café
Entire System Specification

Change Request

- Add the following feature in calculating overtime payment of an employee.
  - ➢ If an employee has worked after 9.00.p.m, double the overtime payment rate for hours after 9.00.p.m





Regression Testing



Regression Testing

Regression Testing	
	Attendance management
	Overtime calculation
	User management
	Salary Calculation

Minor Change Request

- Change the overtime calculation as follows.
  - If an employee has worked after **10.00.p.m**, double the overtime payment rate for hours after **10.00.p.m**





Sanity Testing



Sanity Testing

Sanity Testing	
	Calculating overtime payment with double overtime value only when the overtime is after 10.00. p.m.

User Acceptance Testing



Example







Start Truck Café
Salary Management System

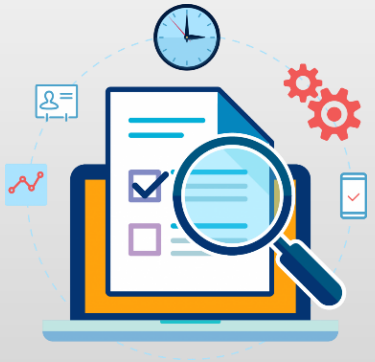
Non-functional Testing

Non-Functional Testing

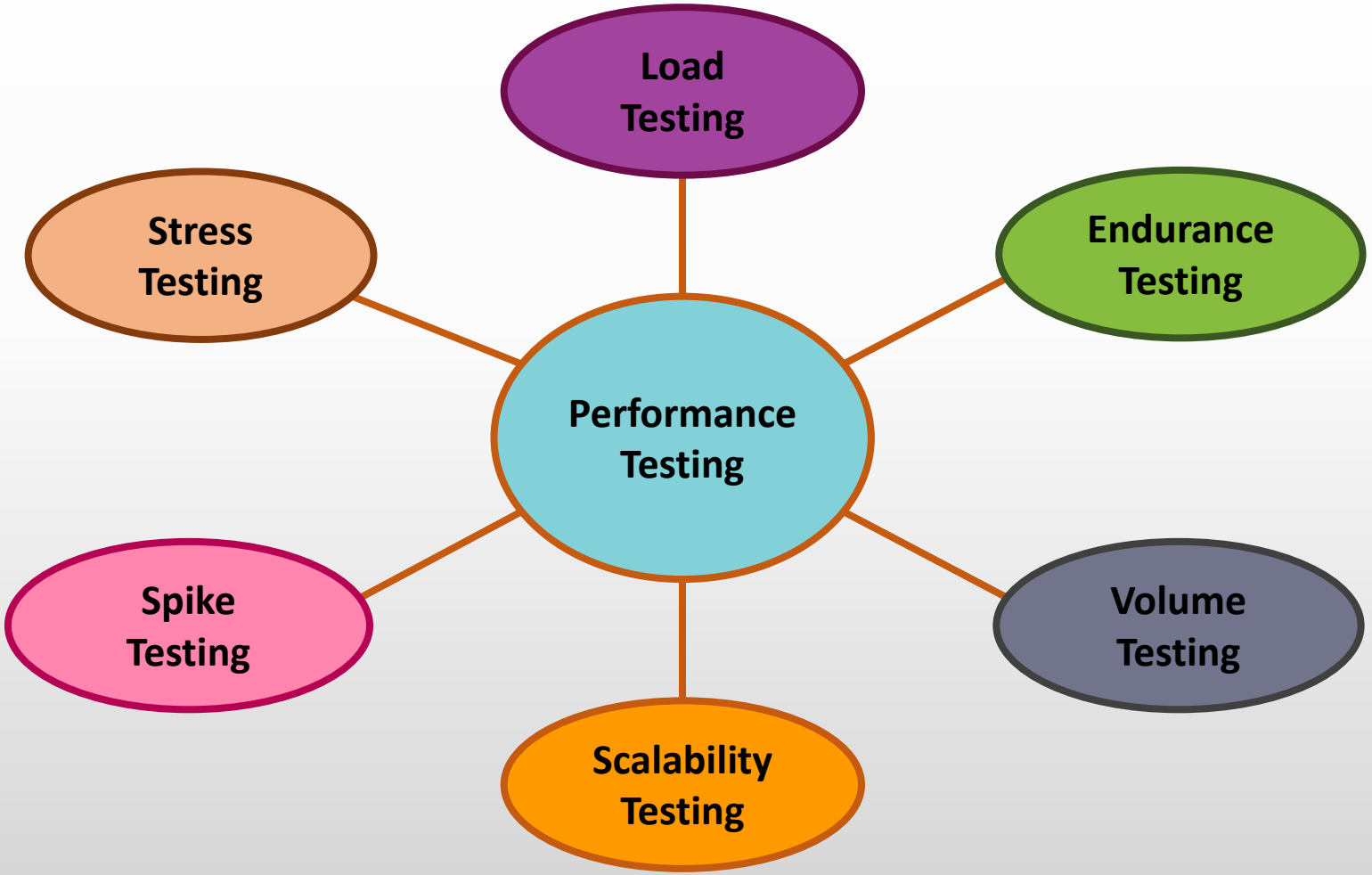
- A type of testing to check non-functional aspects of a software application.
- Examples for non- functional aspects of a software;
  - Performance
  - Usability
  - Reliability
- Explicitly designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.

Performance Testing

- Designed to test whether the program satisfy its performance objectives.
- Measure the performance of each component to identify which components cause the system to perform poorly.
- Performance testing involves quantitative tests done in a laboratory environments.
- Can compare the performance of two systems.



Types of Performance Testing



Load Testing

- Test a system with constantly increasing load until “the time to load” reaches to its threshold value.
- Use to distinguish the performance between two different systems.
- Monitor the response time and staying power of an application, when it is performing under a heavy load.



Load Testing - Examples

- Testing a printer by sending a large job.
- Editing a very large document in a word processor.
- Continuously reading and writing data into hard disk.
- Running multiple applications simultaneously on server.
- Testing of a mail server by accessing thousands of mailboxes.

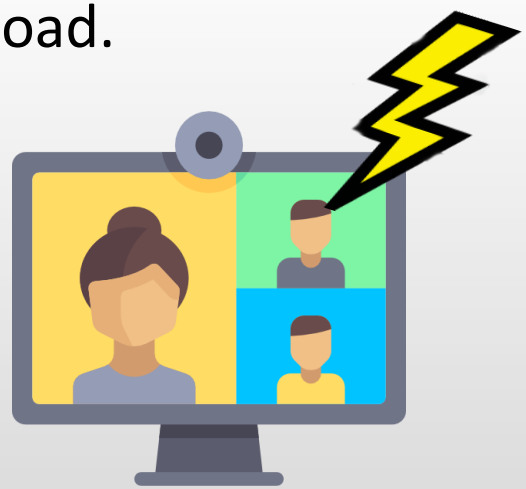
Stress Testing

- Validate an application's behavior when it is pushed beyond normal or peak load conditions.
- Checks the stability of a software when the hardware resources are insufficient.
- Determine failures of a system and identify how the system recover from the failures. This quality is known as the recoverability.



Spike Testing

- Performed by suddenly increasing the number of users by a very large amount
- The main aim is to determine whether the system will be able to sustain the sudden heavy workload.



Endurance Testing

- Testing a system with an expected amount of load over a long period of time.
- Test cases are executed to check the behavior of a system.
- Consider factors such as memory leaks or system fails or random behavior.



Scalability Testing

- Testing of a software application to determine the capability of scaling up in terms of any non-functional requirement.
- Determines the peak of a system, when it has reached a level which prevents from more scaling.



Volume Testing

- Testing a software application with a large amount of data.
- Monitor the performance of application under varying database volumes.



Uses of Performance Testing

- Improve user experience.
- Gather metrics useful for tuning the system.
- Identify bottlenecks such as database configuration.
- Determine if a new release is ready for production.
- Provide reporting to business stakeholders regarding performance against expectations.

Top Performance Testing Tools

- LoadRunner
- Apache JMeter
- NeoLoad
- Rational Performance Tester
- Loadster
- QEngine (ManageEngine)
- Testing Anywhere
- Loadstorm

















Compatibility Testing

- Test an application to ensure that it is compatible across operating systems, hardware platforms, web browsers, etc.
- Validation for compatibility requirements that have been set at the planning stage of the software.
- Validates that the application runs properly in versions.



Types of Compatibility Testing

- **Backward compatibility Testing**

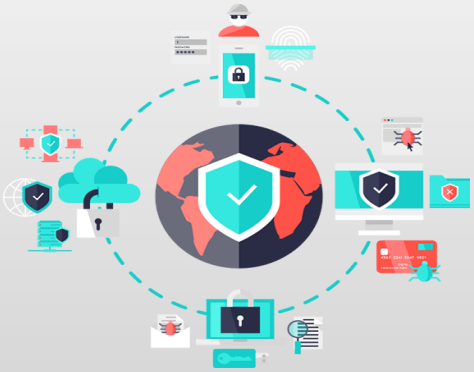
  Verify the behavior of the developed application with the older versions of the application.

- **Forward compatibility Testing**

  Verify the behavior of the developed application with the newer (upcoming) versions of the application.

Security Testing

- Security testing is the process of testing an application to check whether it is according to the specific security objectives.
- Security testing test cases can be derived by studying known security problems in similar systems.
- Web-based applications often need a higher level of security testing than most applications.



Usability Testing

- Test the user-friendliness of an application and identify usability defects.
- Testing is performed by using a small set of target end-users.
- Mainly focuses on the user's ease to use the application, flexibility in handling controls and the ability of the system to meet its objectives.



Localization Testing

- Test whether the software behaves according to the local culture or settings.
- Test whether the application has appropriate linguistic and cultural aspects for a particular locality.
- Localization testing of a globalized application is the process of identifying whether all components of the application are designed according to the local culture of target countries and regions.



Non-Functional Testing



Performance Testing



Localization Testing



Compatibility Testing



Security Testing



Usability Testing

Thank You...

TEST STRATEGY AND PLANNING



INTRODUCTION TO TEST STRATEGY AND PLANNING

TEST STRATEGY OVERVIEW

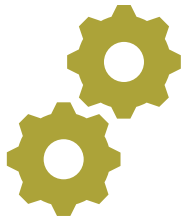
- Definition and purpose

  High-level plan for accomplishing the testing objectives for a software product or system

TEST STRATEGY OVERVIEW

- Importance in Software Testing
  - **Guidance**
  - **Efficiency**
  - **Risk Management**
  - **Quality Assurance**
  - **Cost Savings**

ROLE OF TEST PLANNING

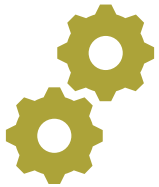


Integration into the software development lifecycle



Ensuring quality and reliability

BENEFITS OF A WELL-DEFINED STRATEGY AND PLAN



Efficiency in testing processes



Minimization of risks



Enhanced product quality

UNDERSTANDING THE TEST STRATEGY

PURPOSE AND OBJECTIVES



Aligning with project goals



Meeting requirements with standards

COMPONENTS OF TEST STRATEGY

- Test scope definition
- Testing approach determination
- Selection of appropriate techniques

TEST PLANNING PROCESS

- Requirement gathering
- Define test objectives
- Identifying test scenarios
- Prioritizing testing activities

TEST COVERAGE AND TRACEABILITY

SIGNIFICANCE OF TEST COVERAGE



Ensuring thorough testing



Functional, structural, and requirement-based coverage criteria

TRACEABILITY CONCEPT



Linking test cases to requirements



Maintaining alignment with project goals

TEST ENVIRONMENT AND DATA MANAGEMENT

SETTING UP TEST ENVIRONMENT



Considerations for setup



Ensuring consistency and reliability

TEST DATA MANAGEMENT

Strategies for privacy and security

- Data masking
- Anonymization
- Encryption
- Access controls
- Data minimization

Maintaining data integrity

- Validation rules
- Version control
- Backup and recovery
- Data auditing
- Quality assurance processes

RESOURCE ALLOCATION AND SCHEDULING

EFFECTIVE RESOURCE ALLOCATION

Human resources, tools, infrastructure

Realistic scheduling and timelines

Task prioritization

Estimation

Buffer management

communication

RISK-BASED TESTING



Identifying and prioritizing risks



Mitigation strategies

MONITORING AND REPORTING

TEST PROGRESS MONITORING



Evaluate test results



Generate meaningful reports

CONTINUOUS FEEDBACK AND ADAPTATION



Importance in refining strategies



Ensuring ongoing quality improvement

Understanding the Testing Life Cycle and Bug Reporting



Outline

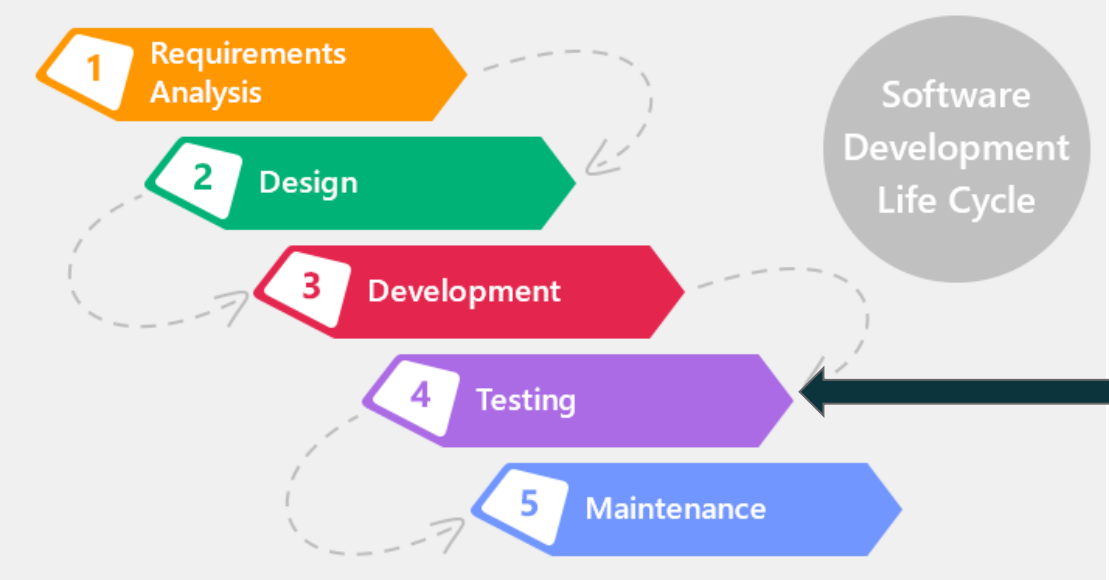
01 Introduction

02 Software Testing Life Cycle

03 Bug Reporting

Introduction

- **Software Development Life Cycle** is the standard business practices to building software applications.



Introduction

- **Quality** is the ability of a product or service to meet a customer's expectations for that product or service.
- **Quality Assurance** in Software Testing is defined as a procedure to ensure the quality of software products.
- Also, **meeting the client requirements** is the key attribute of QA.
- Other Quality attributes,

Correctness Performance Reliability Usability Availability Security Maintainability

Quality Attributes

- **Correctness:** To ensure that the software functions correctly without errors while meeting end-user requirements.
- **Performance:** Measure the time and resources a software utilizes to achieve its intended purposes.
- **Reliability:** Measure whether the software delivers the intended service accurately under various conditions.
- **Usability:** Measure the software's user-friendliness.
- **Availability:** Measures the percentage of time a system is operational and accessible to users without downtime.
- **Security:** Evaluate how well the software handles potential security risks and vulnerabilities.
- **Maintainability:** Assess the ease of maintaining the software.
- **Integrity:** Evaluate the ease of integrating the software with other related systems to enhance its functionality and assess the control over the integration of unauthorized software.

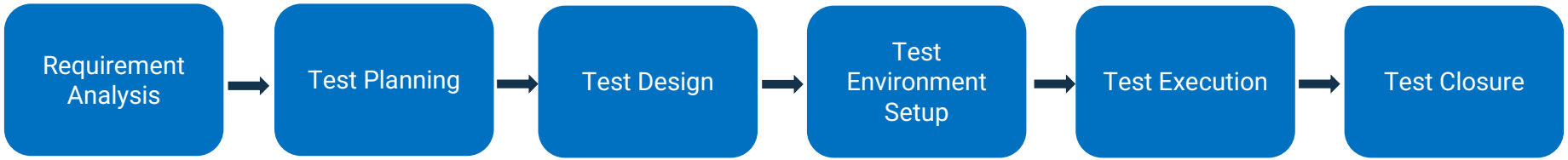
Definitions of Quality

- **Conformance to requirements (Philip Crosby)**
  - Producer view: characterized by:
    - Doing the right thing
    - Doing it the right way
    - Doing it right the first time
    - Doing it on time without exceeding cost

- **Fit for use (Joseph Juran & Edwards Deming)**
  - Customer view: characterized by:
    - Receiving the right product for their use
    - Being satisfied that their needs have been met
    - Meeting their expectations
    - Being treated with integrity and other quality attributes

Software Testing Life Cycle (STLC)

- Software Testing Process



Software Testing Life Cycle (STLC)

Requirement Analysis

1. Requirement Analysis

- In this phase quality assurance team understands the requirements like what is to be tested
- The activities that take place during the Requirement Analysis stage include:
  - **Reviewing the Software Requirements Specification (SRS) and other related documents**
  - **Interviewing stakeholders to gather additional information**
  - **Identifying any missing or incomplete requirements**
  - **Creating a Requirement Traceability Matrix (RTM) to map requirements to test cases**

    RTM : Cross-references requirements with test cases, ensuring comprehensive coverage and validation of all requirements
  - **Output : RTM**

Test Planning

2. Test Planning

- ❏ Most efficient phase of the software testing life cycle where all testing plans are defined
- ❏ The activities that take place during the Test Planning stage include:
  - **Identifying the testing objectives and scope**
  - **Identifying testing types**
    - **Manual Testing :**
      - Manual testing is testing of the software where tests are executed manually by a tester to find the defects/bugs.
      - Testers execute the test cases.
    - **Automation Testing :**
      - Automation testing is the process of testing the software using an automation tool to find the bugs. (Examples: Selenium, Testlink, Junit)

Introduction to Quality Assurance

- Identifying the testing environment and resources needed
- Identifying the functional and non-functional areas to be tested
- Estimating the time and cost required for testing
- Assigning roles and responsibilities to the testing team
- Identifying the test deliverables
- Identifying any potential risks or issues that may impact the testing process
- Reviewing and approving the test plan
- **Output : Test Plan**

Test Design

3. Test Design

- In this phase testing team notes down the detailed test cases
  - **Test Case**
    - A test case is a set of actions performed on a system to determine if it satisfies software requirements and functions correctly.
    - A test case document includes test steps, test data, preconditions, expected outputs and actual outputs.
- The activities that take place during the Test Case Development stage include:
  - **Identifying the test cases**
  - **Creating test data and test scenarios that will be used in the test cases**
  - **Identifying the expected results for each test case**
  - **Reviewing and validating the test cases**
  - **Updating the Requirement Traceability Matrix (RTM) to map requirements to test cases**
  - **Output : Test Cases**

Test Environment Setup

4. Test Environment Setup

- ❑ The software to be tested.
- ❑ The operating system, database, and testing server.
- ❑ Test data.
- ❑ Network configuration.
- ❑ The device on which the software is to be tested – desktop or mobile devices.
- ❑ Output: Testable Environment

Test Execution

5. Test Execution

- In this phase testing team starts executing test cases based on prepared test cases in the earlier step.
- The activities that take place during the test execution stage include:
  - **Test execution**
  - **Defect logging**
  - **Test result analysis**
  - **Defect retesting**
  - **Test Reporting**
  - **Output: Test Report**

Test Closure

6. Test Closure

- The main objective of the test closure stage is to ensure that all testing-related activities have been completed
- The activities that take place during the test closure stage include:
  - **Test summary report**
  - **Test closure report**
  - **Defect Tracking**
  - **Feedback and improvements**
  - **Test environment clean-up**

Outputs

Bug Reporting

- Reporting bugs is a fundamental process involving the documentation and communication of software defects, commonly known as “bugs.”
- Relevant developers responsible for correcting them.
- **These bugs are essentially unintended errors or flaws in a software system that can result in malfunctions, unexpected behaviors, or disruptions to its normal operation.**
- **Severity**: The seriousness of the defect on the product functionality
- **Priority**: The order in which the developer should resolve a defect

Bug Reporting

- **Types of Severity**
  - Critical Severity:
    - The defect causes a complete system failure or prevents key functionalities from working.
    - **Examples:**
      - Application crashes on startup.
      - Login failure for all users.
      - Security breaches, such as unauthorized access to sensitive data.

Bug Reporting

- **Types of Severity**
  - Major Severity:
    - The defect significantly impacts functionality but does not cause a complete system failure.
    - **Examples:**
      - A report generation feature does not work, but users can retrieve data manually.
      - Payment processing works, but receipts are not generated.
      - Login failure for all users.

Bug Reporting

- **Types of Severity**
  - Minor Severity:
    - The defect affects a small part of the application but does not stop major functionality.
    - The issue might cause confusion or inconvenience but does not significantly impact business operations.
    - **Examples:**
      - A misplaced icon.
      - Alignment issues in a table.

Bug Reporting

- **Types of Priority**
  - High Priority:
    - These defects must be fixed immediately as they impact critical functionality or business operations.
    - **Examples:**
      - Application crashes when logging in.
      - Payment gateway not processing transactions.
      - Incorrect logo displayed on the website.

Bug Reporting

- **Types of Priority**
  - Medium Priority:
    - These defects affect important functionalities.
    - Fixing them is necessary, but they can be addressed after high-priority defects.
    - **Examples:**
      - Incorrect error messages displayed.
      - Some UI elements misaligned but still functional.
      - Minor performance issues that slow down processes.

Bug Reporting

- **Types of Priority**
  - Low Priority:
    - These defects cause minimal inconvenience and have little impact on functionality or business operations.
    - They can be fixed in future releases or updates.
    - **Examples:**
      - Spelling errors in UI text.
      - Minor color or alignment issues in non-critical areas.
      - Slightly slower-than-expected response times in non-essential features.

Bug Reporting

- **Exercise : Determine the severity and priority levels for the following scenarios and explain the reasoning behind your selections.**
  - A spelling mistake in the logo text on the homepage.
  - A security vulnerability that exists in an internal admin panel, but it’s not exposed to the public.
  - An application feature that fails to update real-time information but still allows users to manually refresh the page.
  - The login form fails under rare conditions, specifically with a certain browser version.
  - A button on a webpage is slightly misaligned.
  - The system crashes when a user tries to upload a 10GB file.

Bug Reporting

- **Scenario: A spelling mistake in the logo text on the homepage.**
  - Severity: Minor – It’s a cosmetic issue and doesn't affect functionality.
  - Priority: High – It’s a branding issue, and since it’s visible to customers, it needs to be fixed quickly to maintain the company’s image.
- **Scenario: A security vulnerability that exists in an internal admin panel, but it’s not exposed to the public.**
  - Severity: Critical – This vulnerability is severe and could lead to a breach if exploited.
  - Priority: Low – Since it’s not exposed in the live environment and only exists internally, the fix can be delayed in favor of more urgent production issues.

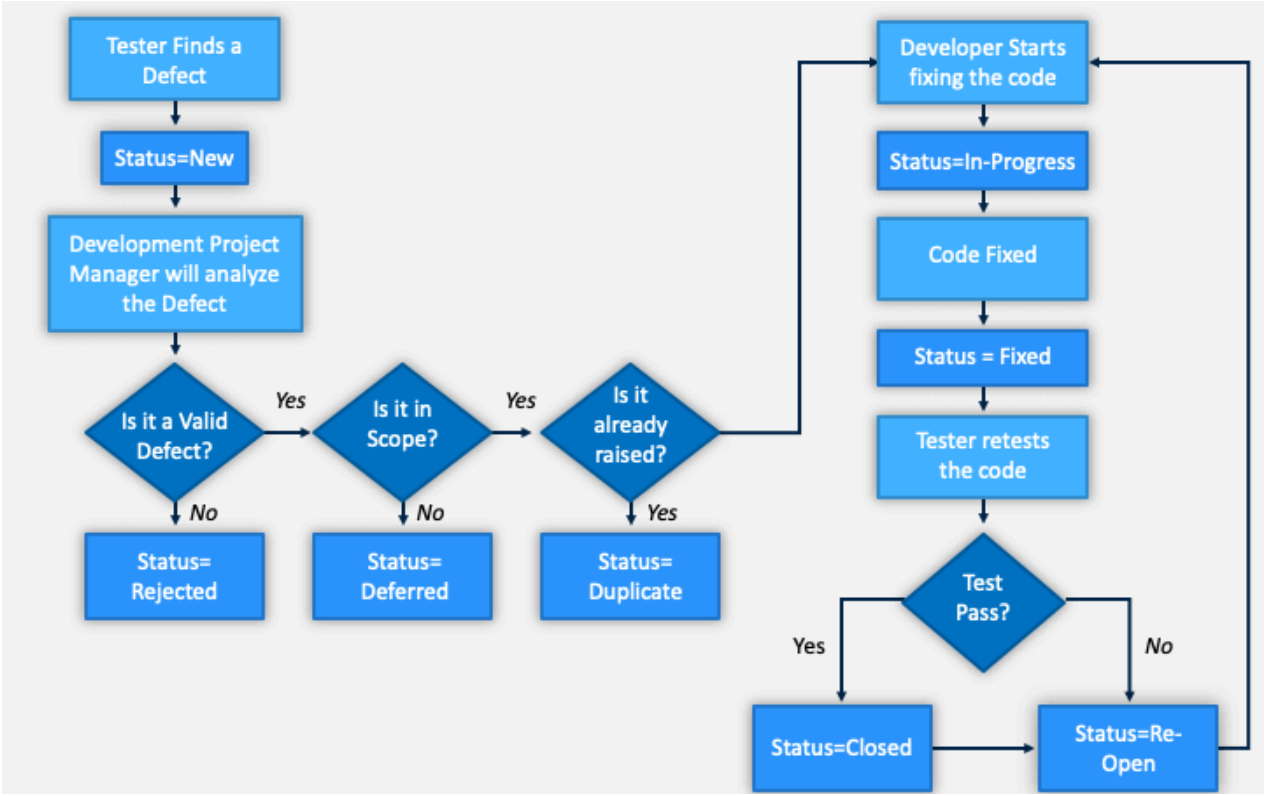
Bug Reporting

- **Scenario: An application feature that fails to update real-time information but still allows users to manually refresh the page.**
  - Severity: Major – The feature doesn't work as intended, but users can still use the application by refreshing manually.
  - Priority: Medium – It’s important to fix, but it’s not urgent because users can still perform the task with a workaround.
- **Scenario: The login form fails under rare conditions, specifically with a certain browser version.**
  - Severity: High – The issue is severe for those affected as users cannot log in.
  - Priority: Medium – Although it's a high-severity bug, it only affects a small subset of users, so it may be fixed in the next release or sprint.

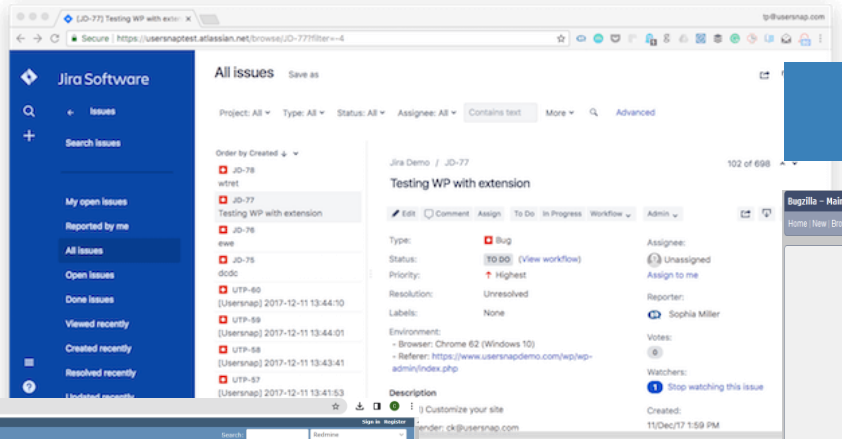
Bug Reporting

- **Scenario: A button on a webpage is slightly misaligned.**
  - Severity: Minor – The misalignment is a cosmetic issue that doesn’t impact functionality or user experience significantly.
  - Priority: Low – Since it doesn’t affect core functionality, it can be addressed later, after more critical issues are resolved.
- **Scenario: The system crashes when a user tries to upload a 10GB file.**
  - Severity: Major – While the system crashes under extreme conditions, it's not a common scenario.
  - Priority: High – Since this issue affects users who deal with large files, it should be fixed soon, especially if it's a common business requirement.

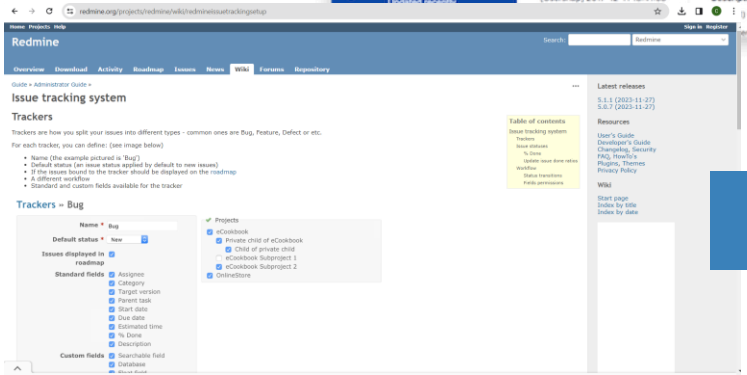
Defect Life Cycle



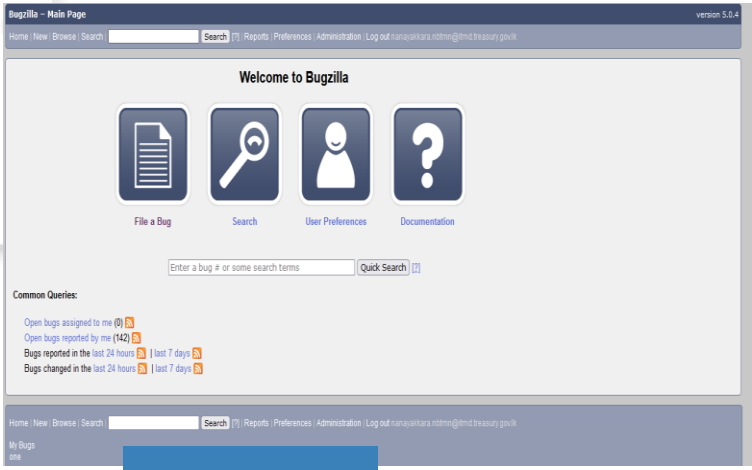
Bug Reporting Tools



Jira



Redmine



Bugzilla

Challenges and Best Practices

- **Challenges**
  - **Changing Requirements:** Managing evolving requirements can lead to scope creep, affecting test coverage and stability.
  - **Resource Constraints:** Limited resources, including time, budget, and skilled personnel, can hinder thorough testing.
  - **Complex Systems:** Testing complex systems with intricate interdependencies poses challenges in identifying and addressing defects effectively.
  - **Testing Environment Setup:** Establishing and maintaining the necessary test environments can be time-consuming and error-prone.
  - **Test Data Management:** Generating and managing test data that accurately represents real-world scenarios can be challenging.

Challenges and Best Practices

- **Best Practices to Overcome Challenges**
  - **Requirements Management:** Implement robust requirements management processes, including clear documentation, frequent reviews, and effective change management.
  - **Prioritization:** Prioritize testing efforts based on risk assessment, criticality, and business impact to optimize resource utilization.
  - **Test Automation:** Invest in test automation tools and frameworks to streamline repetitive tasks, increase test coverage, and accelerate testing cycles.

Challenges and Best Practices

- **Best Practices to Overcome Challenges**
  - **Continuous Integration/Continuous Deployment (CI/CD):** Implement CI/CD pipelines to automate the deployment and testing of software updates, ensuring rapid feedback and quality assurance.
  - **Collaboration and Communication:** Foster a culture of collaboration and open communication among stakeholders, including developers, testers, and business analysts, to facilitate knowledge sharing and alignment of objectives.

Skills of a Good Software Tester

- **Non-Technical Skills**
  - **Analytical skills:**
    - A good software tester should have sharp analytical skills.
    - Analytical skills will help break up a complex software system into smaller units to gain a better understanding and create test cases.
  - **Communication skill:**
    - A good software tester must have good verbal and written communication skills.
    - Testing artifacts (like test cases/plans, test strategies, bug reports, etc.) created by the software tester should be easy read and comprehend.

Skills of a Good Software Tester

- **Non-Technical Skills**
  - **Time Management & Organization Skills:**
    - Testing at times could be a demanding job especially during the release of code, a software tester must efficiently manage workload, have high productivity, exhibit optimal time management, and organization skills.
  - **Great Attitude:**
    - To be a good software tester you must have a GREAT attitude. An attitude to 'test to break', detail orientation, willingness to learn and suggest process improvements.

Skills of a Good Software Tester

- **Technical Skills**
  - knowledge of Database Management Systems
  - Test Management
  - Defect Tracking tool
  - Automation Tools

Test Automation and Metrics

SQA - SLIIT
By Deshal Weerasinghe

About Me!

- BSc. (IT) – Curtin University, 2009
- MBA – PIM (University of Sri Jayewardenepura), 2014
- ISTQB Certified Advanced Test Manager
- 16+ Years Industry experience with 6 di as a QA Engineer and Team Lead
- Worked with five companies in different domaing
- Working as Senior Lead QA Engineer at Acumatica
- Visiting Lecturer at SLIIT

Table of Contents

01 **A Quick Intro**

Why do we test?
Why automate tests?

02 **Some Basics**

Basic test process
Where does automation come in?

03 **Test Automation**

The importance, advantages and drawbacks

04 **Automation in Practice**

How do we practically use test automation

05 **Test Automation Tools**

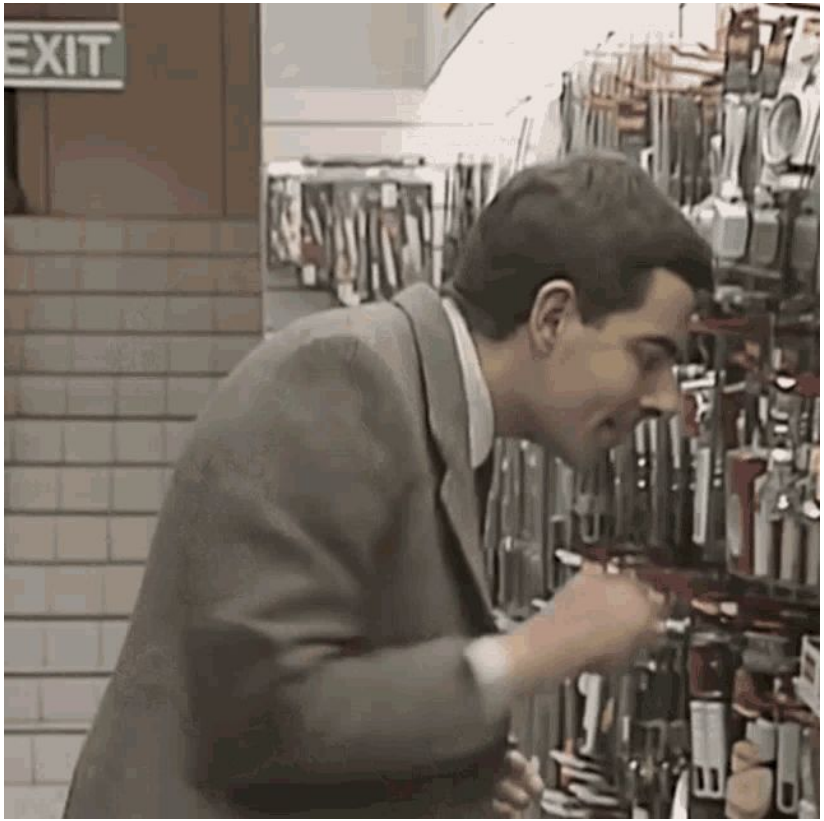
Tools in use at present, how to select one, which one is the best?

06 **Test Automation Metrics**

How automation can help decision making

01
A Quick Introduction to Testing

Why do we test something?



Why do we test something?



Why do we test something?



Why do we test something?

- To know whether it is ‘working’
- To know whether it ‘fits the purpose’
- To know whether ‘something is wrong’ with it
- To know ‘how it fails’, when it fails

Ultimately, it’s about verifying whether it meets the quality requirements!

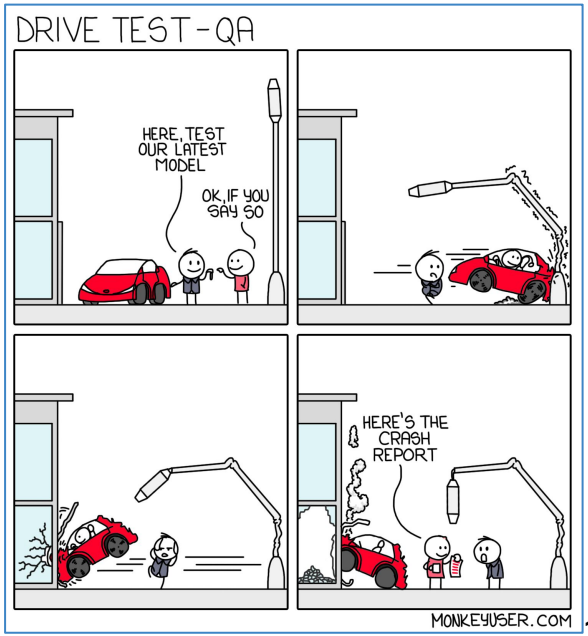
Why software should be tested?

- Humans develop software
- They may make errors/ mistakes
- These errors built in to software are ‘defects’
- The defects will cause the software to ‘fail’
- The failures cause damages – financial and otherwise
- Testing helps to identify these defects/ failures before the software is used by customers
- To reduce (not eliminate) defects
- To ensure the right thing is built the right way (Verification and Validation)

It’s all about gathering information!

“I am only a good tester when I don't care if it works or not, only that I provide the information needed to confirm or deny that it works.”

- B. Hal Metz (Test Lead Consultant, Fusion Alliance)



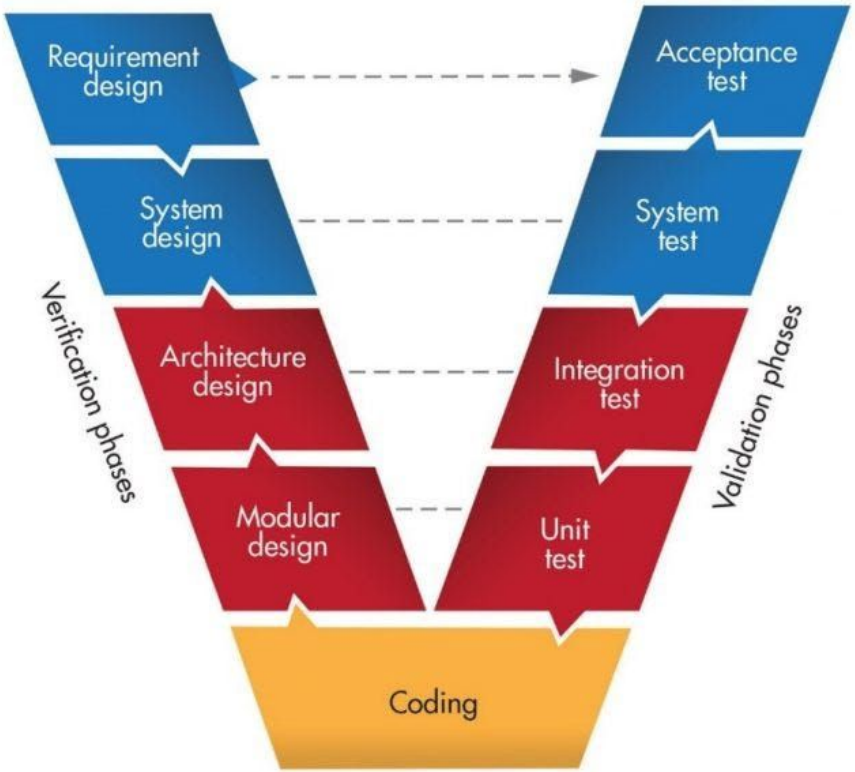
02

Basics of Software Testing

How software are tested

- Identify test context/ conditions
- Prepare test scenarios/ test cases
    - Use test design techniques
- Test in different levels of software development
    - This is where test automation comes in
- Use multiple test types
- And different test execution types to suit the task at hand
- Report test outcomes

The V-model


Requirement design
System design
Architecture design
Modular design
Coding
Unit test
Integration test
System test
Acceptance test
Verification phases
Validation phases

The V-model

VERIFICATION

Are we building it right it right?



- Building the product right
- Implementation meets specifications
- Detects defects early

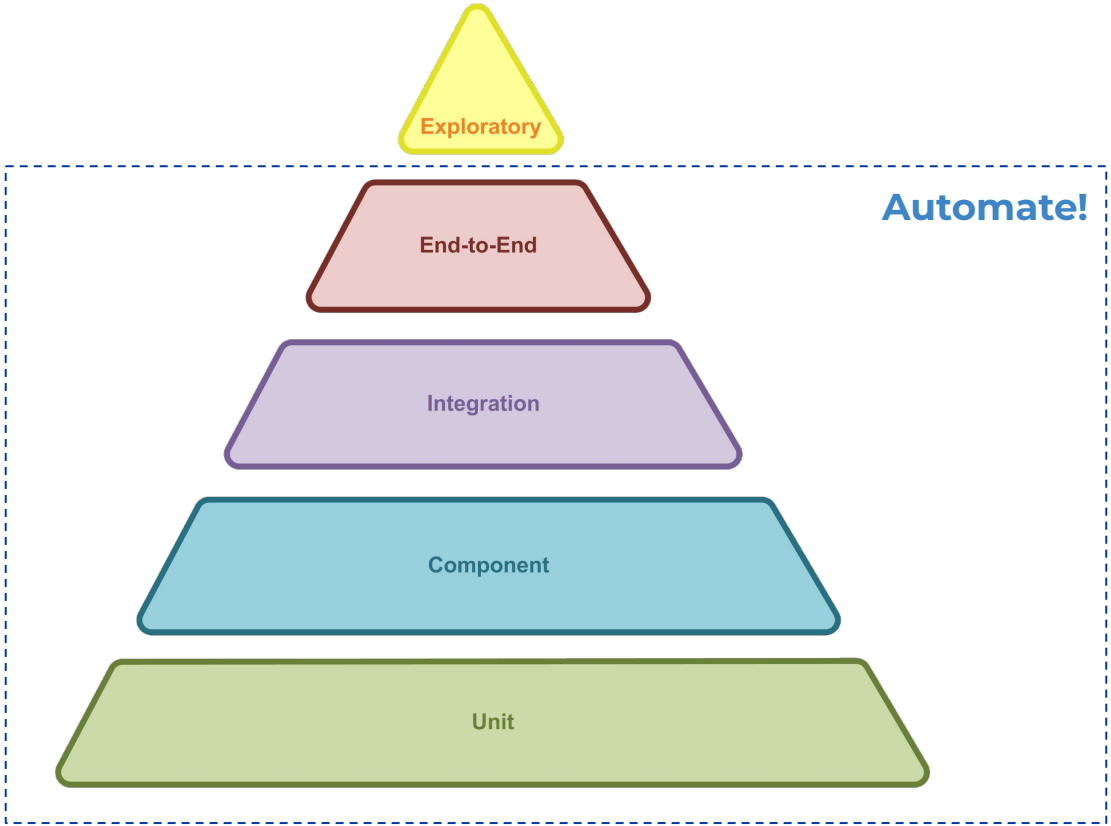
VALIDATION

Are we building the right thing?



- Building the right product
- Meets user needs
- Evaluates final outcome

Test Pyramid



03

Test Automation

What is Test Automation?

- Making a computerized tool/ framework execute the test cases and report outcomes
  - With minimal human intervention for execution and reporting
  - Consuming considerably less amount of time
  - Executing at anytime, in automated processes such as CI/CD

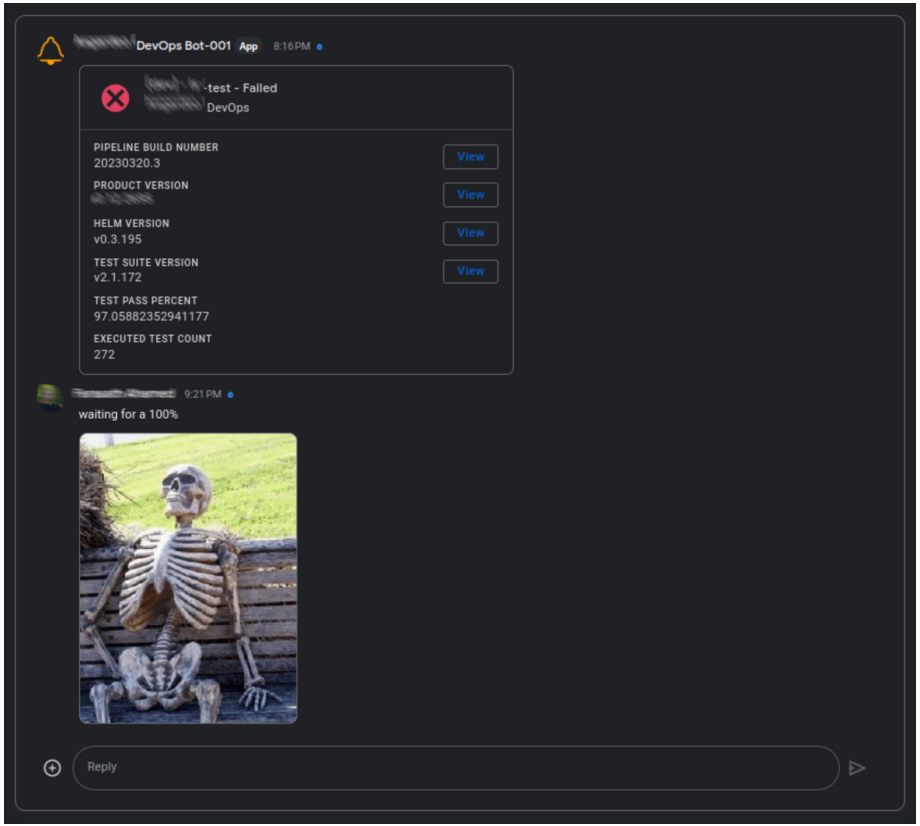
Why Test Automation is Required?

- Resilience
  - Execute the exact set of steps repeatedly, whenever demanded and as many times as required, without getting exhausted
  - Not subject to human errors (when executing the test)
- Speed of execution
  - An automated test typically takes a fraction of the time consumed for manually running the same test
  - A critical requirement for speedy delivery with CI/CD model
  - Faster feedback loop
- Better reporting
  - Reports can be auto-generated, without human intervention
- Frees-up human testers for non-automatable tests
  - This improves test coverage

Drawbacks of Automated Testing

- Requires additional investment on test frameworks
  - Additional learning required for using and maintaining the framework
  - Some automated frameworks require paid subscription
- Human errors can be made when automating tests, which could deliver false results
  - This could lead to false-negatives (consumes time to investigate false failures) and false-positives (bugs can go undetected)
- Requires development support
  - Eg. standardized element identifier usage when developing UI
- Unstable automation suite could be problematic, especially when integrated to CI/CD process

Practical Problems with Automation



[redacted] DevOps Bot-001 App 8:16 PM

[redacted]-test - Failed
[redacted] DevOps

PIPELINE BUILD NUMBER
20230320.3
View

PRODUCT VERSION
[redacted]
View

HELM VERSION
v0.3.195
View

TEST SUITE VERSION
v2.1.172
View

TEST PASS PERCENT
97.05882352941177

EXECUTED TEST COUNT
272

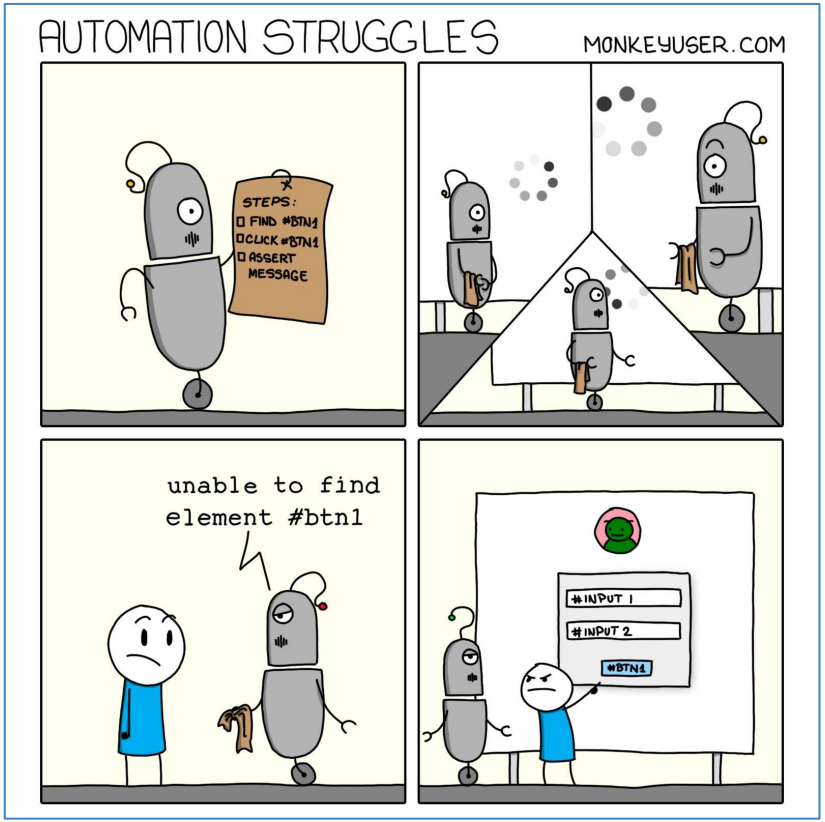
[redacted] 9:21 PM

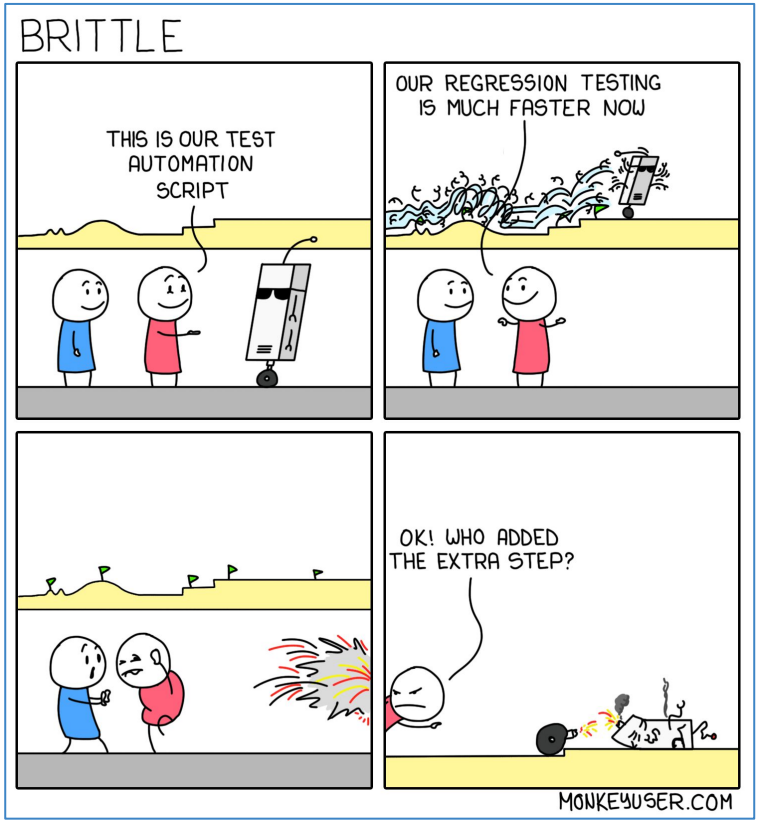
waiting for a 100%



Reply

Practical Problems with Automation





Practical Problems with Automation

“Flaky Tests”

- Outcomes are not dependable
- Often difficult to fix/ make stable
- Automation frameworks provide workarounds, such as retrying



Practical Problems with Automation

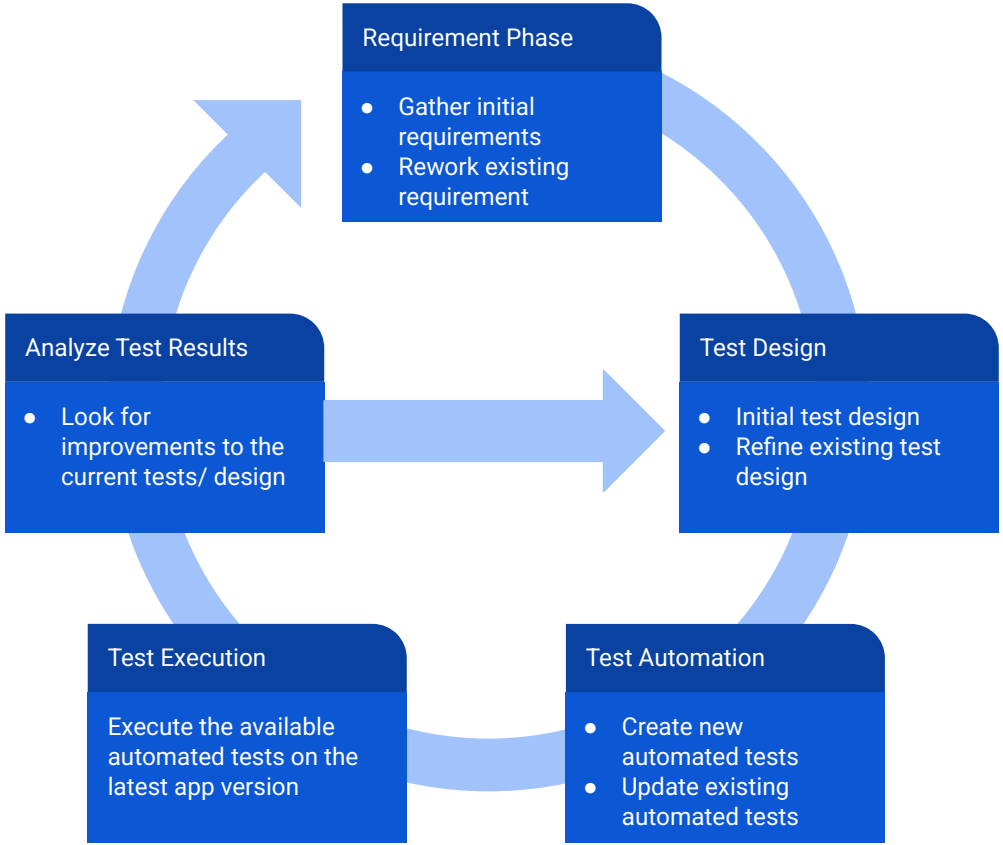
“Slow Tests”

- UI-based tests are often slower than API-based tests
  - This could be due to network and rendering delays
  - Also could be caused by unnecessary waiting for elements to load
    - ‘Wait’ vs ‘Timeout’ - can choose depending on framework support
    - ‘Wait for element/ response’ could be ideal
    - Modern automation frameworks provide ‘smart-wait’ mechanisms

04

Automation in Practice

Automation Workflow



How do we do it

Show the process with actual project references

05

Test Automation Tools

What are the available tools?















Which tool should I use?

- Points to consider:
  - What kind of tests to be automated
    - On which platform(s) the tests are executed (mobile/ web/ different browsers etc.)
    - Types of test to automate (functional, UI, performance etc.)
  - Features/ capabilities (eg. smart-waits, flaky test management etc.)
  - Ease of integration (with existing other tools/ processes)
  - Reporting capabilities
  - Ease of ues/ Learning curve for the team
  - Budget/ ROI/ Trial periods
  - Cost of support/ maintenance

Key Points on Playwright

Built for testing

Auto-wait and web-first assertions

Playwright waits for elements to be actionable before performing actions. Assertions automatically retry until conditions are met. No artificial timeouts, no flaky tests.

Test isolation

Each test gets a fresh browser context — equivalent to a brand new browser profile. Full isolation with near-zero overhead. Save authentication state once and reuse it across tests.

Resilient locators

Find elements with selectors that mirror how users see the page: `getByRole`, `getByLabel`, `getByPlaceholder`, `getByTestId`. No brittle CSS paths.

Parallelism and sharding

Tests run in parallel by default across all configured browsers. Shard across multiple machines for faster CI. Full cross-browser coverage on every commit.

Source: Playwright

Tools Differ, Basics don’t

- When automating tests, you follow the same basic set of steps - regardless of the tool you use
  - Arrange → Act → Assert
- The tool may support different ways to create the test scripts (eg. Test Generator) but you have to review and make it perfect
  - Make sure the correct locators are used (eg. there could be more than one locator for the same element, one of which is the best to use depending on the tool)
  - Broad vs narrow selectors, XPath vs ClassName/ ID
- Once the initial work is done (especially if you’re using a new tool) - break it down to a better structure with page objects, reusable methods etc.

06

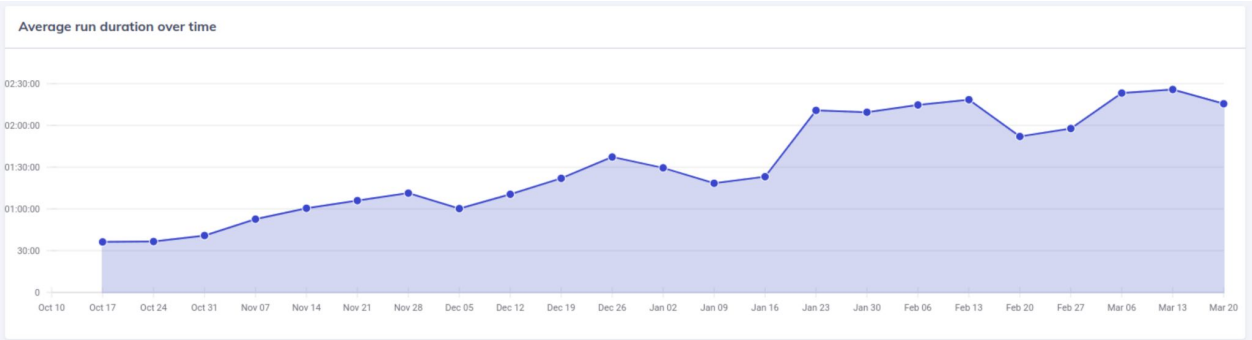
Test Automation and Metrics

Why Metrics are Required?

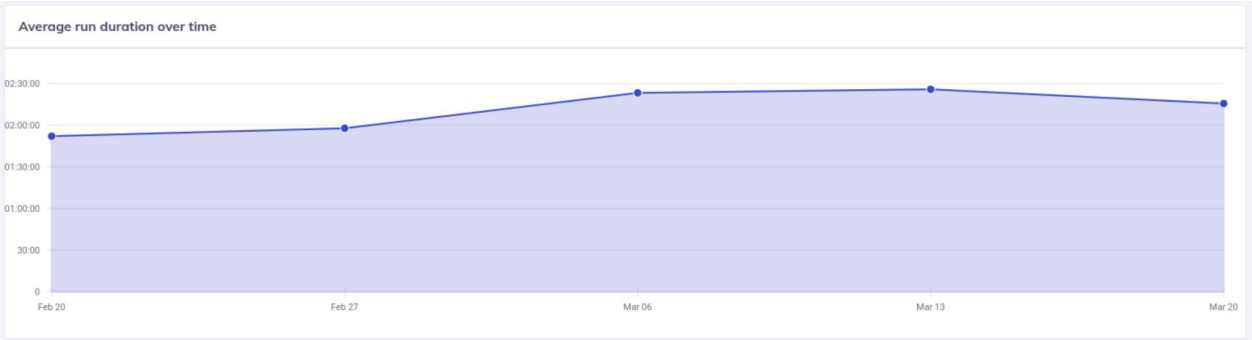
- Remember, we test to *gather information*!
- Metrics provide windows to get insights about the product, the development cycles as well as the tests itself
    - When it comes to decision making, these insights are often considered vital.
    - Measuring is a step at the core of any management process
        - (Plan - do - check - act)
    - Eg. determine the stability of the product/ build before a release
- Helps to improve accuracy of estimations

Key Metrics of Software Testing

- Total Test Execution Time
  - How long it takes to complete a test execution round
  - Especially for automated tests, the longer it takes, the lesser the value it provides
- Automated Test Coverage
  - How many from the total number of tests are automated
  - This provides an insight about how much of a manual work has to be done when testing the product
  - Combined with other detailed metrics such as automation coverage based on test priority, this could help to take the go/ no-go decision of a release

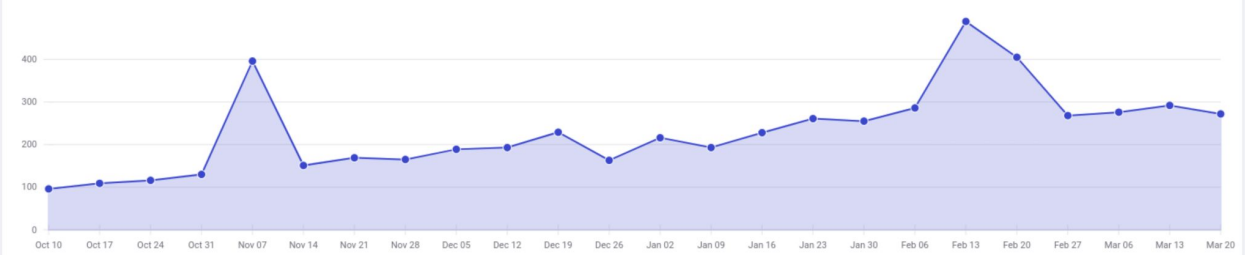


Average run duration	Typical concurrency ℹ	Time saved from parallelization
1h 41m	1 machine	—



Average run duration	Typical concurrency ℹ	Time saved from parallelization
2h 7m	1 machine	—

Test suite size over time



Total test cases

833

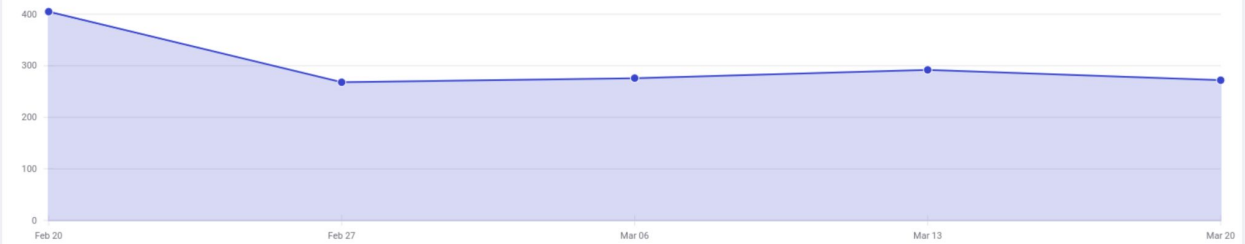
↑ 1%

Spec files

147

↓ 18%

Test suite size over time



Total test cases

438

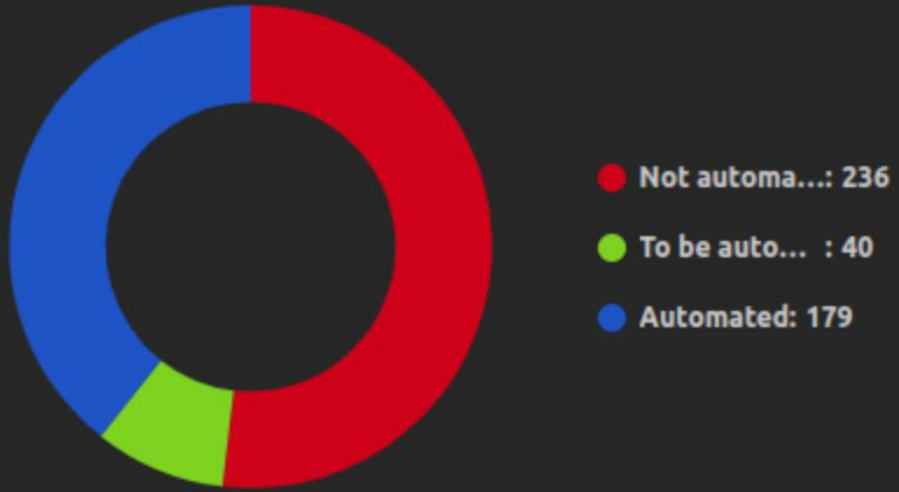
↓ 12%

Spec files

69

↑ 3%

Test case automation distribution



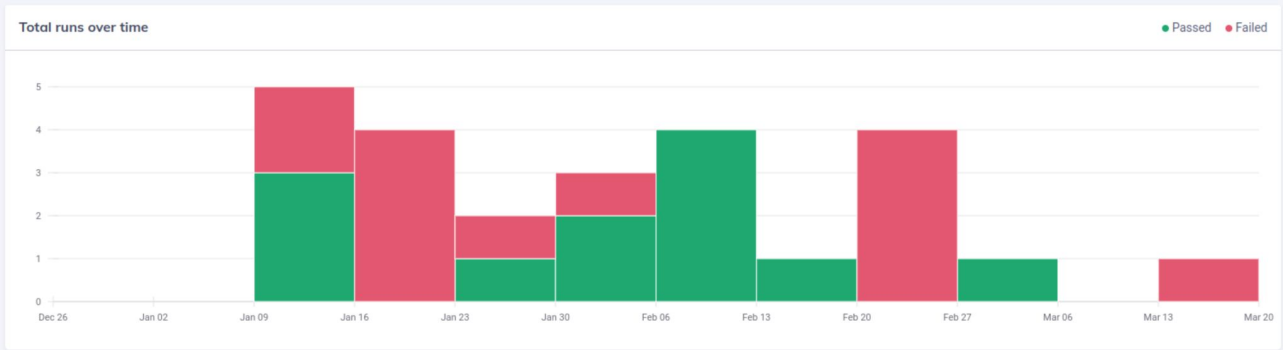
Test cases automation ratio

ASG

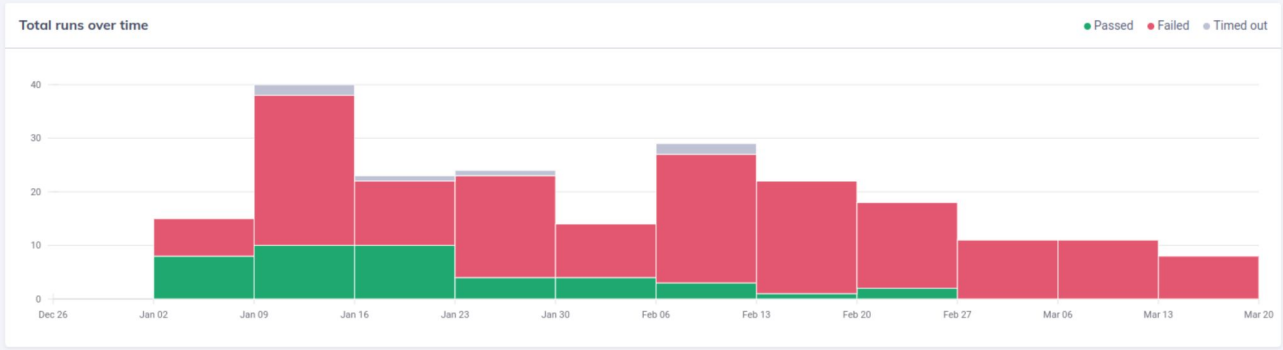
39.34 % ↑ 58.41% vs last 30 days

Key Metrics of Software Testing (contd.)

- Automation Stability
  - How well the automation suite keeps up over time
  - There could be unstable - flaky - tests, which could only be revealed after being executed over a period of time
  - This determines to which extent we can rely on the test execution results when making decisions
- Test Passing (or Failing) Rate
  - Number of passing (or failing) tests as a percentage of total number of tests

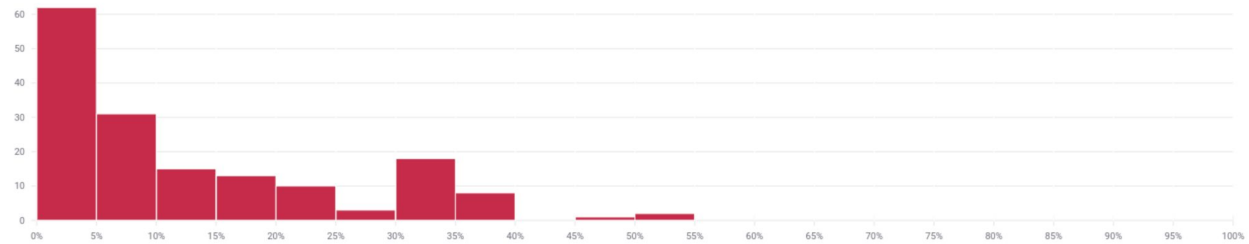


Total runs	Average per day	Passed runs	Failed runs
25	1.25	12	13
↑ 27%	↑ 4%		



Total runs	Average per day	Passed runs	Failed runs
215	3.52	42	167
↑ 27%	↑ 4%		

Number of test cases by failure rate



Median failure rate

9%

Number of failures

163

Test Cases

Distribution of test cases by duration



Median duration

00:20

Number of test cases

438

Key Metrics of Software Testing (contd.)

- Defect Distribution
  - Testing principle: Defect clustering
  - The biggest number of defects are clustered around a small number of functional points (often the most complex ones)
  - Gives an insight about application/ component stability
- Defect Open/ Close Rate
  - Ratio between the number of defects before and after a delivery
  - Over a period of time, this gives an insight about whether the products quality is improving

Question Time!

Thanks!

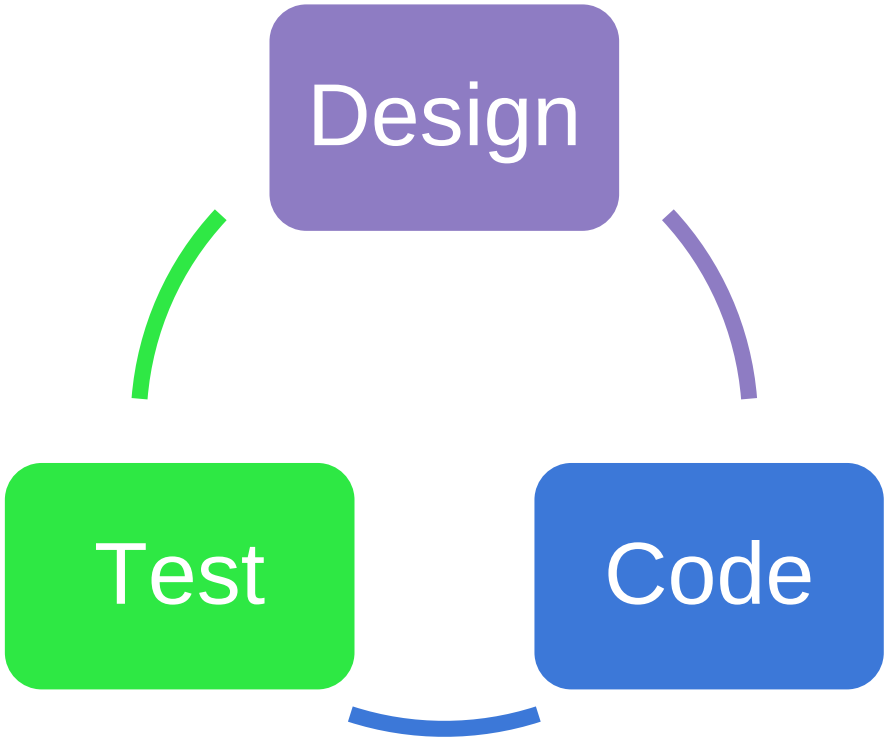
In case if a question pops-up later:
deshal.lec@gmail.com

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**

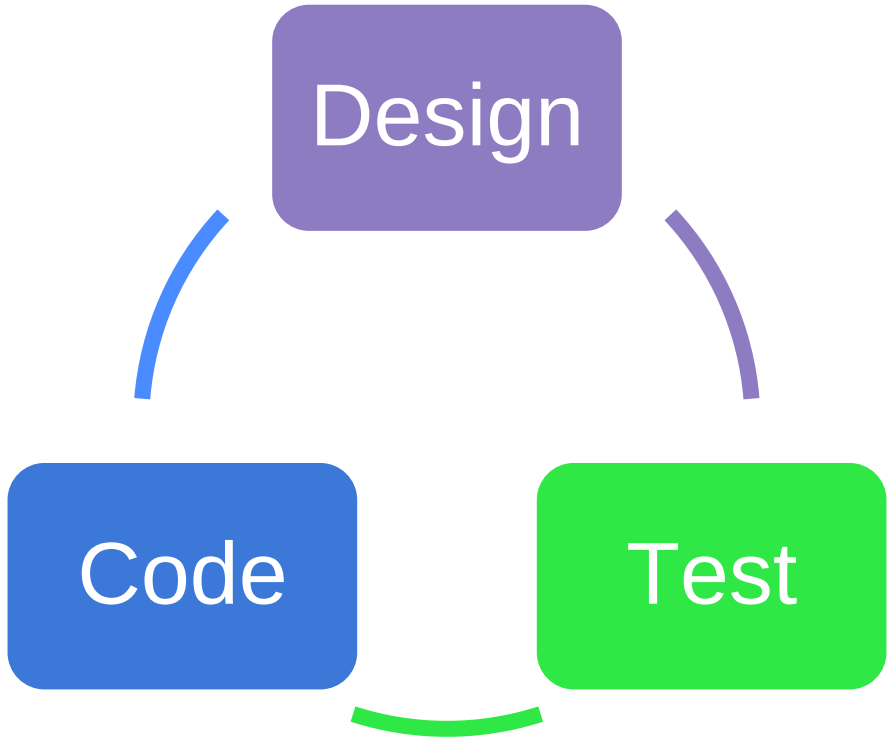
Test Driven Development

The Key Difference

Traditional Approach



TDD



What is TDD?

- Test-driven Development (TDD) is an evolutionary approach of development
  - Stems from Extreme Programming - an Agile development method
- It is a specification technique, which ensures the source code is thoroughly tested at confirmatory level
  - The tests and the code are built together, iteratively and one use case at a time
- It combines the approaches of Test-first Development and ‘Refactoring’
  - Test-first Development: write tests first, then write just enough code to pass the test
  - Refactoring: make small changes to the code, to improve the design and to make it easier to understand and modify - ‘Evolve the code slowly over time’
- A good way of thinking-through the requirements in contrast to coding logic, before starting to code

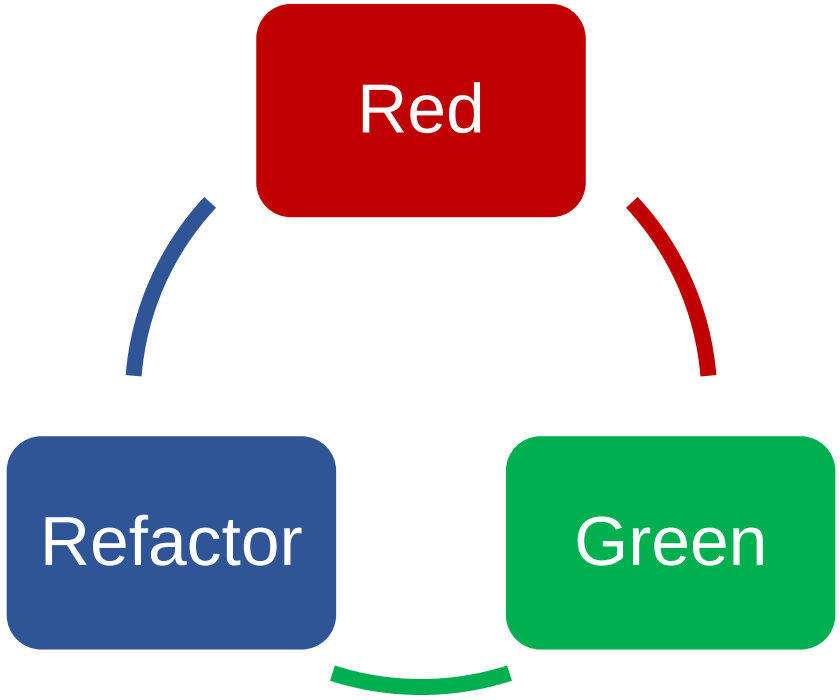
TDD vs. Traditional Development & Testing

- TDD focuses more on improving the production code – whereas traditional approach focuses on designing test cases and code to cover the functional requirements
- TDD ensures the system requirements are met from the design of the code itself.
- Both approaches help find and fix defects, but TDD does it at very early stages of development lifecycle, which reduces the cost

How TDD Works – at a High Level

- A test is developed first
  - The test specifies and validates what the code would do
  - Normally, TDD begins with designing and implementing test cases for granular functionalities of the system
- Execute the tests
  - If a test fails, implement code sufficient to pass that test
  - This high-level approach can also be called as ‘Test-first Development (TFD)’
  - The tests are almost always automated
- Refactor the code to fix any functionalities relevant to failed tests

The ‘Red-Green-Refactor’ Cycle



- **Red Phase:** Write an *automated* test for a behavior which you are going to implement as a function. This could be based on a user requirement. Execute the test, it will fail.
- **Green Phase:** Write the production code, but only just enough to pass the failed test from the previous phase. Keep things as simple as possible, it’s ok to ignore code duplication and following best practices at this stage.
- **Refactor Phase:** Make the code better. Remove duplication at a minimum. Make the code more generic. The tests will not allow you to break the code by trying to improve it.

Three Laws of TDD



01

You are not allowed to write any production code unless it is to make a failing unit test pass.



02

You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.



03

You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Source: Professionalism and Test-Driven Development, Robert C. Martin – 2007 - via IBM

TDD Laws - Simplified Version



01

Write only enough of a unit test to fail.



02

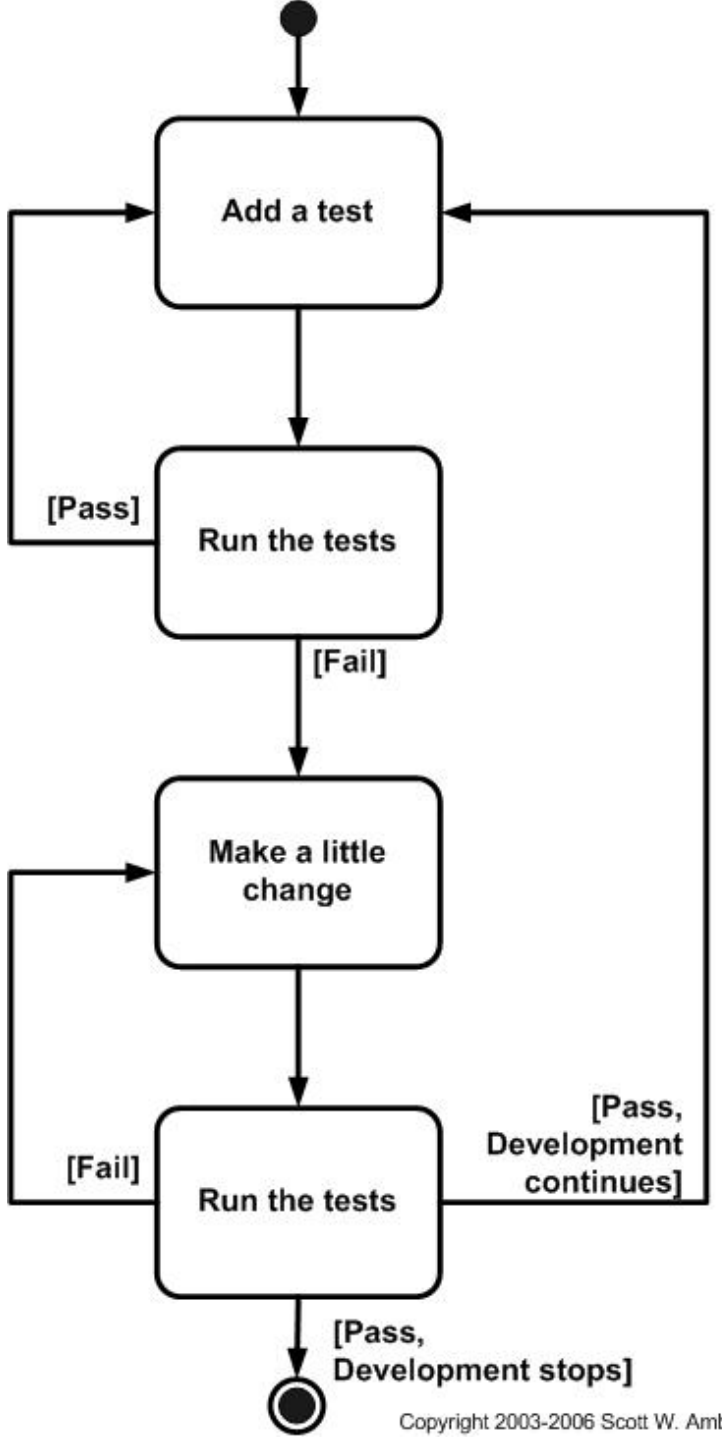
Write only enough production code to make the failing unit test pass.

Source: Javier Trevino Saldana - via IBM

First Level of TDD : ATDD

- ATDD : Acceptance-test Driven Development
  - Team members representing three different perspectives (the ‘three amigos’) of the application collaboratively prepare acceptance tests
    - Customer: What is the problem that has to be solved?
    - Developer: How are we going to solve the problem?
    - Tester: What if <something> happens?
- Then implement just enough code to pass the tests which are failing
- Focus is more on overall behavior of the system
- Acceptance tests could act as a set of requirements of the application under development
- Also known as ’Story Driven Development (SDD)’ or ‘Behavior Driven Development (BDD)’

TFD/ ATDD/ SDD/ BDD
Common Cycle

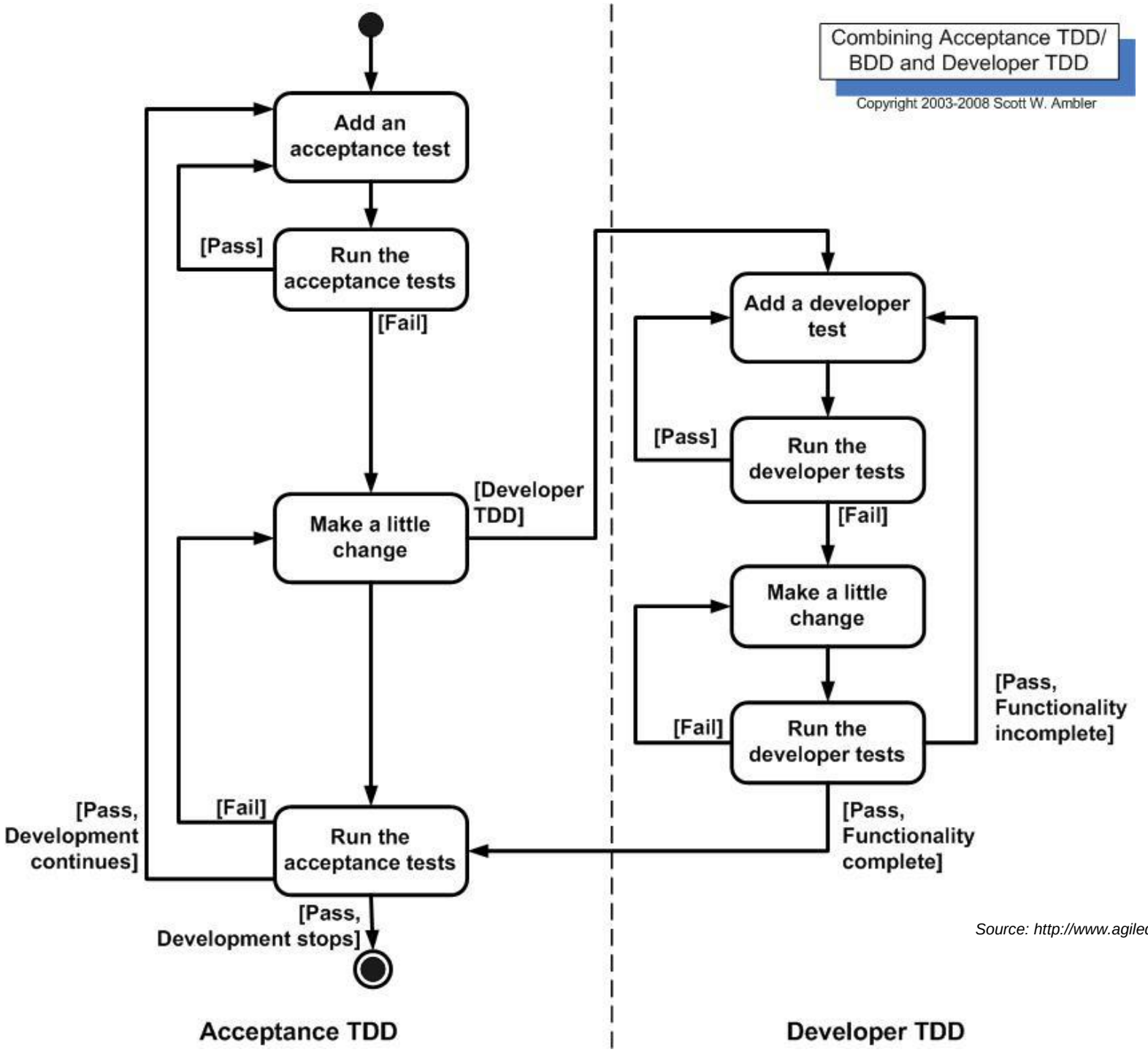


Source: http://www.agiledata.org/essays/tdd.html

Second Level of TDD : DTDD

- Prepare tests more focused on ‘Developer perspective’ first
  - ie. Unit Tests
- Focus is on a selected granular functionality of the system
- Collectively these tests could form an executable design of the application
- ATDD in most cases requires having an associated DTDD practice in place
- However, DTDD can be done without being associated with ATDD

DTDD Cycle



Source: http://www.agiledata.org/essays/tdd.html

Advantages of TDD

- Helps designing compact code focused on individual functionalities, rather than monolithic procedures which perform multiple functionalities
  - Code is cleaner and simpler
  - East to understand, hence easy to maintain
  - Helps to avoid code duplication
- A set of automated tests are readily available along with the code
  - Refactoring and changing can be done more safely and with more confidence
  - Minimal level of risk when updating the code or fixing defects
  - The tests themselves were tested

Advantages of TDD (contd.)

- Find, debug and fix bugs early in development lifecycle
- Provides information about whether the system meets its requirements
- Guarantees achieving 100% code coverage by testing almost every single line of code
- The tests could serve as documentation
- The requirements are implemented with little to no wasted effort because only the function that is needed is written.

TDD Best Practices

- Use naming conventions
  - This helps to organize the tests better. Increases readability
  - Name test classes in similar way to the functional classes
  - Use descriptive names for test methods
- Separate functional code from test code
  - This helps to avoid packaging tests together with the prod binaries
  - Some build tools expects this separation to be there
- Re-run all tests when functional code changes
- Write new tests/ refactor only when all tests are passing
- Assert first, limit assertion scope

TDD Best Practices (contd.)

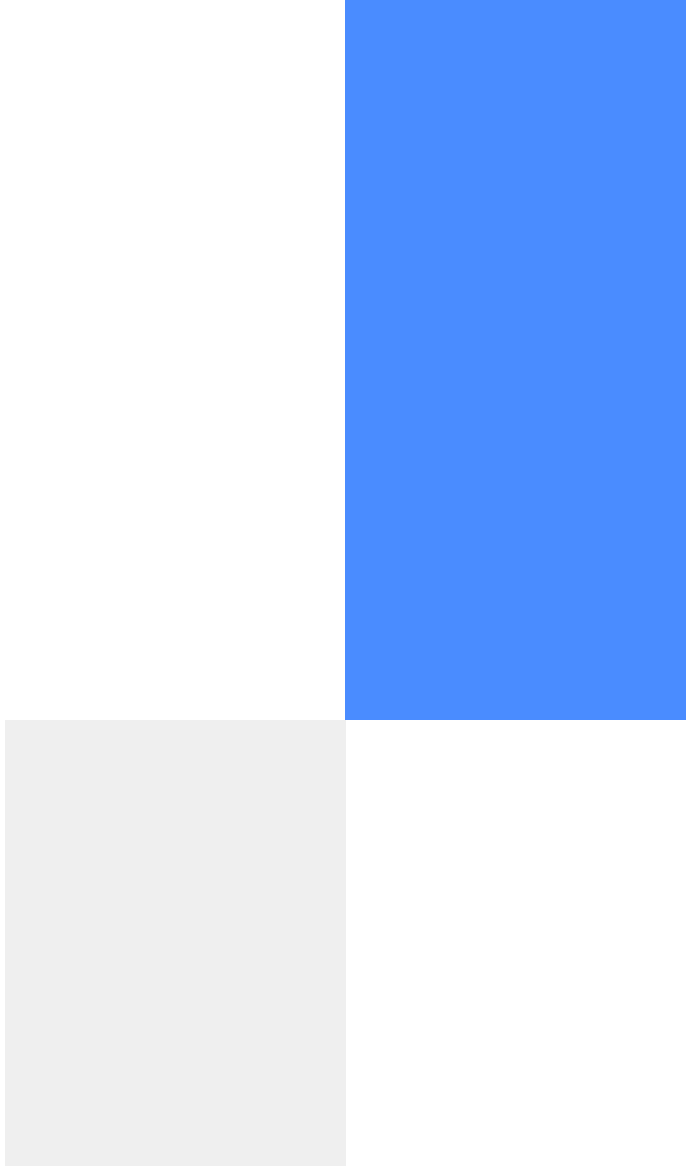
- Avoid dependencies between tests
- Ensure test execution efficiency
  - Avoid ‘waits’ and ‘sleeps’, use timeouts
- Use ‘setup’ and ‘teardown’ methods
  - Execute pre/post conditions for each tests separately
- Avoid base classes
  - This improves clarity by not having to recursively navigate to understand a test
- Run all tests that are developed as part of your development pipeline. A failing test must stop the pipeline.

Challenges Faced when Implementing TDD

- Obtaining buy-in for TDD from project leadership
  - TDD is often viewed as causing additional development overhead
  - If done properly, TDD will drastically reduce technical debt and hence will save cost later in the product life cycle
- Ensure that the development team understands TDD. Writing the test can sometimes require more effort than writing the code
  - ‘Tester mindset’ is required to formulate the tests
  - There could be resistance due to misunderstanding of the concepts
- Measure and monitor the value that is gained by implementing TDD
  - The primary gain comes from reduced technical debt and time to market - could be difficult to show value without comparison
  - Coverage figures however are measurable

TDD vs. Plain-old Unit Tests

- In ATDD, the tests are not ‘unit tests’
- Just having unit tests does not necessarily mean that the team/ organization follows TDD – Unit tests can be written after code
- Unit tests written after code do not give sufficient direct feedback on coding and the design
- Unit tests written after code often focuses only on the ‘happy-path’



TDD in Practice

Example

- https://www.browserstack.com/guide/tdd-in-java#toc2

Behavioral Driven Development (BDD)

- Definition and concept of BDD

  - Behavior Driven Development
  - software development methodology
  - Focused on the behavior of a system rather than its implementation
  - emphasizes collaboration among developers, testers, and non-technical stakeholders to ensure that the software behaves as expected from a business perspective.
  - BDD uses scenarios written in a human-readable format, often using a Given-When-Then structure, to describe the desired behavior of the system

Behavioral Driven Development (BDD)

- Relationship between TDD and BDD

- TDD: developers write tests before writing the actual code. These tests guide the development process and ensure that the code behaves as expected. TDD primarily focuses on unit testing and verifying the correctness of the code.

- BDD: BDD extends TDD by emphasizing the behavior of the system from an end-user perspective. BDD scenarios, written in a human-readable format, describe how the system should behave in different situations. These scenarios serve as both requirements documentation and executable tests, guiding development while ensuring alignment with business goals.

Behavioral Driven Development (BDD)

- Benefits of BDD

  - Improved Collaboration
  - Clearer Requirements
  - Early Detection of Issues
  - Maintainable Tests
  - Focus on Business Value

BDD Workflow

- Writing user stories in Gherkin syntax

Gherkin follows a very specific syntax:
Scenario -> Given -> When -> Then

Scenarios: All the actions a user could take (including bad input)

GIVEN: Sets the context. What page are we on and what state are we in? Is the user an admin? Signed-in? Have you created a campaign?

WHEN: What actions the user is performing? What event occurred?

THEN: What should the system do in response? What is the expected outcome?

BDD Workflow

Example:

Feature: User Authentication
As a user
I want to be able to log in
So that I can access my personalized dashboard

Scenario: Successful login
Given I am on the login page
When I enter valid credentials
And I click the login button
Then I should be redirected to my dashboard

BDD Workflow

Example (cont):

Scenario: Invalid login attempt

Given I am on the login page

When I enter invalid credentials

And I click the login button

Then I should see an error message

BDD Workflow

Activity:

Write the following user story using the Gherkin Syntax

As a MailChimp user, I want to access my account so I can start working

BDD Workflow

- Mapping user stories to executable specifications

In this example, we've written two scenarios based on the user story. Each scenario represents a different case: one where the login is successful and one where it fails due to invalid credentials.

BDD Workflow

- Implementing step definitions

```
const { Given, When, Then } = require('cucumber');
const LoginPage = require('../pageObjects/LoginPage');
const DashboardPage = require('../pageObjects/DashboardPage');

Given('I am on the login page', async function () {
  await LoginPage.open();
});

When('I enter valid credentials', async function () {
  await LoginPage.enterCredentials('validUsername', 'validPassword');
});

When('I enter invalid credentials', async function () {
  await LoginPage.enterCredentials('invalidUsername', 'invalidPassword');
});

When('I click the login button', async function () {
  await LoginPage.clickLoginButton();
});

Then('I should be redirected to my dashboard', async function () {
  await DashboardPage.waitForLoad();
  expect(await DashboardPage.isVisible()).to.be.true;
});
```

Implementing BDD with Tools

- Overview of BDD frameworks (e.g., Cucumber, SpecFlow)

BDD frameworks provide tools and utilities to implement Behavior-Driven Development practices effectively. Two popular BDD frameworks are Cucumber and SpecFlow.

Cucumber:

- supports multiple programming languages, including Java, JavaScript, Ruby, and others.
- It allows writing feature files in Gherkin syntax, which is human-readable and understandable by all stakeholders.
- Cucumber enables the mapping of feature files to step definitions, where step definitions are implemented to execute the scenarios described in feature files.
- Cucumber facilitates collaboration among developers, testers, and business stakeholders by providing a common language for expressing requirements and verifying behavior.

Implementing BDD with Tools

- Overview of BDD frameworks (e.g., Cucumber, SpecFlow)(cont)

SpecFlow:

SpecFlow is a BDD framework specifically designed for .NET applications and supports languages such as C#, VB.NET, and F#.

enables writing feature files in Gherkin syntax and mapping them to step definitions.

SpecFlow integrates seamlessly with Visual Studio and other .NET development environments, making it a popular choice for teams working in the Microsoft ecosystem.

Similar to Cucumber, SpecFlow promotes collaboration and ensures alignment between business requirements and implemented functionality.

Implementing BDD with Tools

- Executing BDD tests

Once you have written feature files and corresponding step definitions, you can execute BDD tests using the respective BDD framework's test runner.

For example:

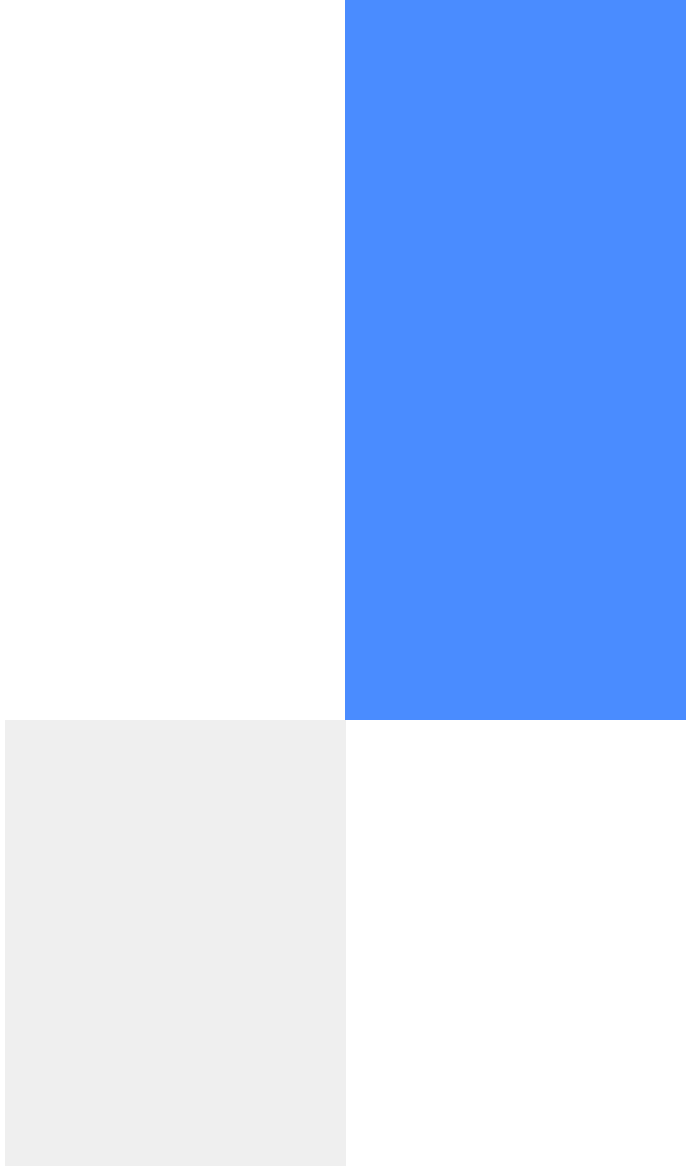
For Cucumber, you can use tools like **cucumber-js** for JavaScript-based projects or **cucumber-jvm** for Java-based projects to execute tests.

For SpecFlow, you can execute tests using the built-in test runner provided by Visual Studio or integrate SpecFlow with other CI/CD tools for automated test execution.

Best Practices and Tips

This section will be a self-study activity for students

- Dos and Don’ts of TDD and BDD
- Common pitfalls and how to avoid them
- Strategies for introducing TDD and BDD in real-world projects



BDD in Practice

A Practical Example

- https://www.linkedin.com/pulse/real-world-example-bdd-1-behavior-driven-agile-fred-mastropasqua/

Software Maturity Models

What is Software Process

- Set of activities and associated outcome that produce a software product.

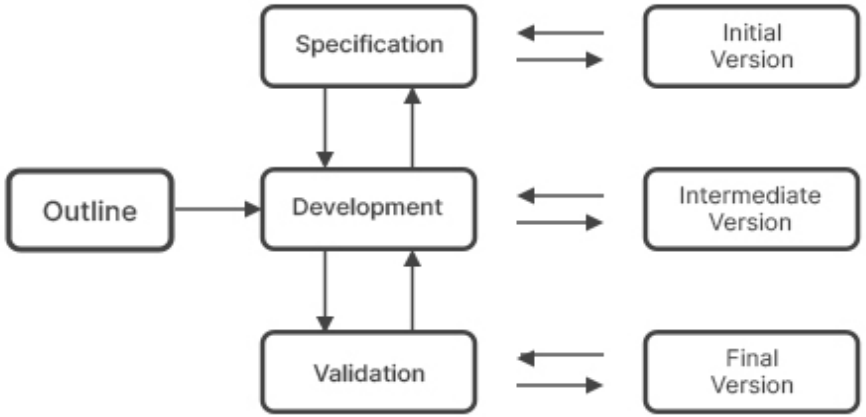
- Consist of four key process activities:
    1. Software specification.
    2. Software development.
    3. Software validation.
    4. Software evolution.

Software Process Model

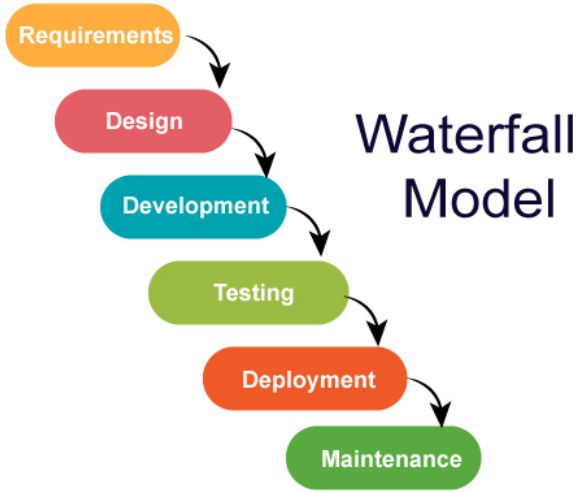
- Software process model is an abstraction of the actual process, which is being described.
- Process models may contain activities, which are part of the software process, software product, and the roles of people involved in software engineering.
- Some examples :
    1. **A workflow model**
    2. **A dataflow or activity model**
    3. **A role/action model**

Software Process Model

- There are several various general models or paradigms of software development:
  1. **The waterfall approach**
  2. **Evolutionary development**
  3. **Formal transformation**
  4. **System assembly from reusable components**

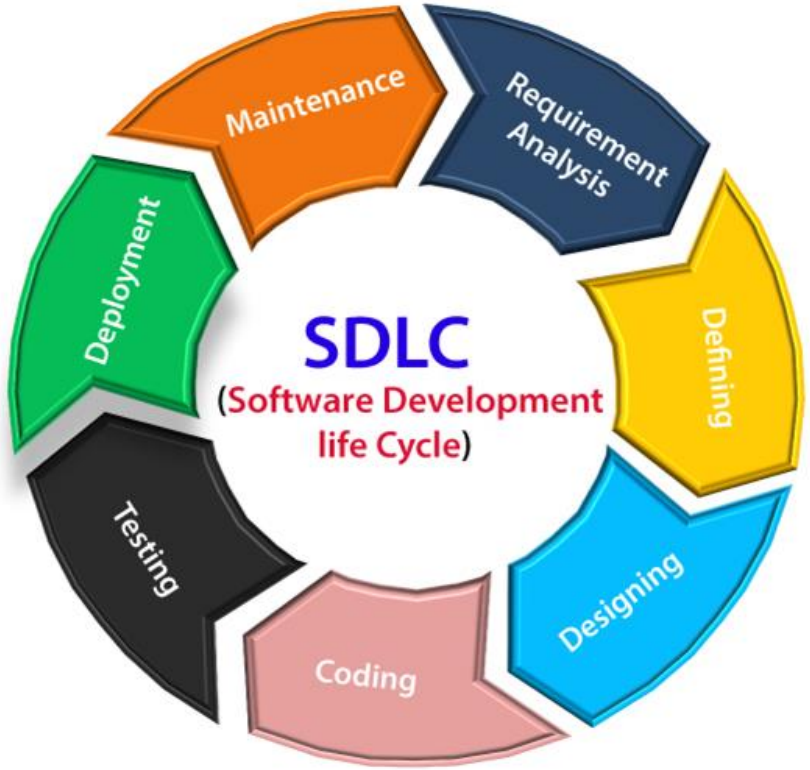


Evolutionary Development Of Software Development



Waterfall Model

Software Development Life Cycle (SDLC)



- A software life cycle model is a pictorial and diagrammatic representation of the software life cycle.
- A life cycle model represents all the methods required to make a software product transit through its life cycle stages.

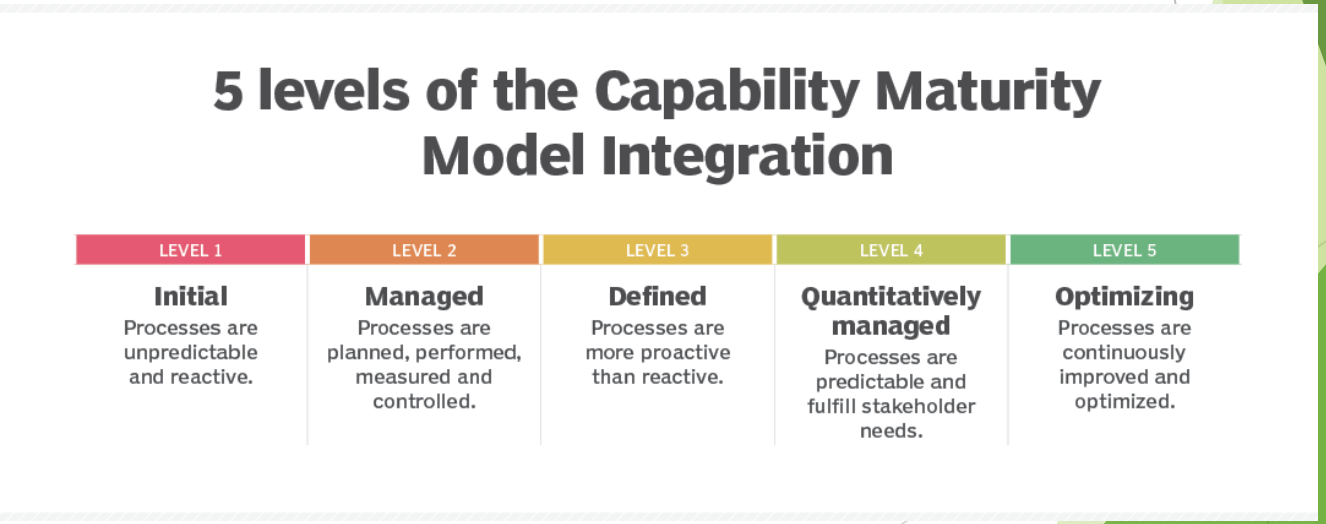
Why we need SDLC?

- The development team must determine a suitable life cycle model for a particular plan and then observe to it.
- Without using an exact life cycle model, the development of a software product would not be in a systematic and disciplined manner.

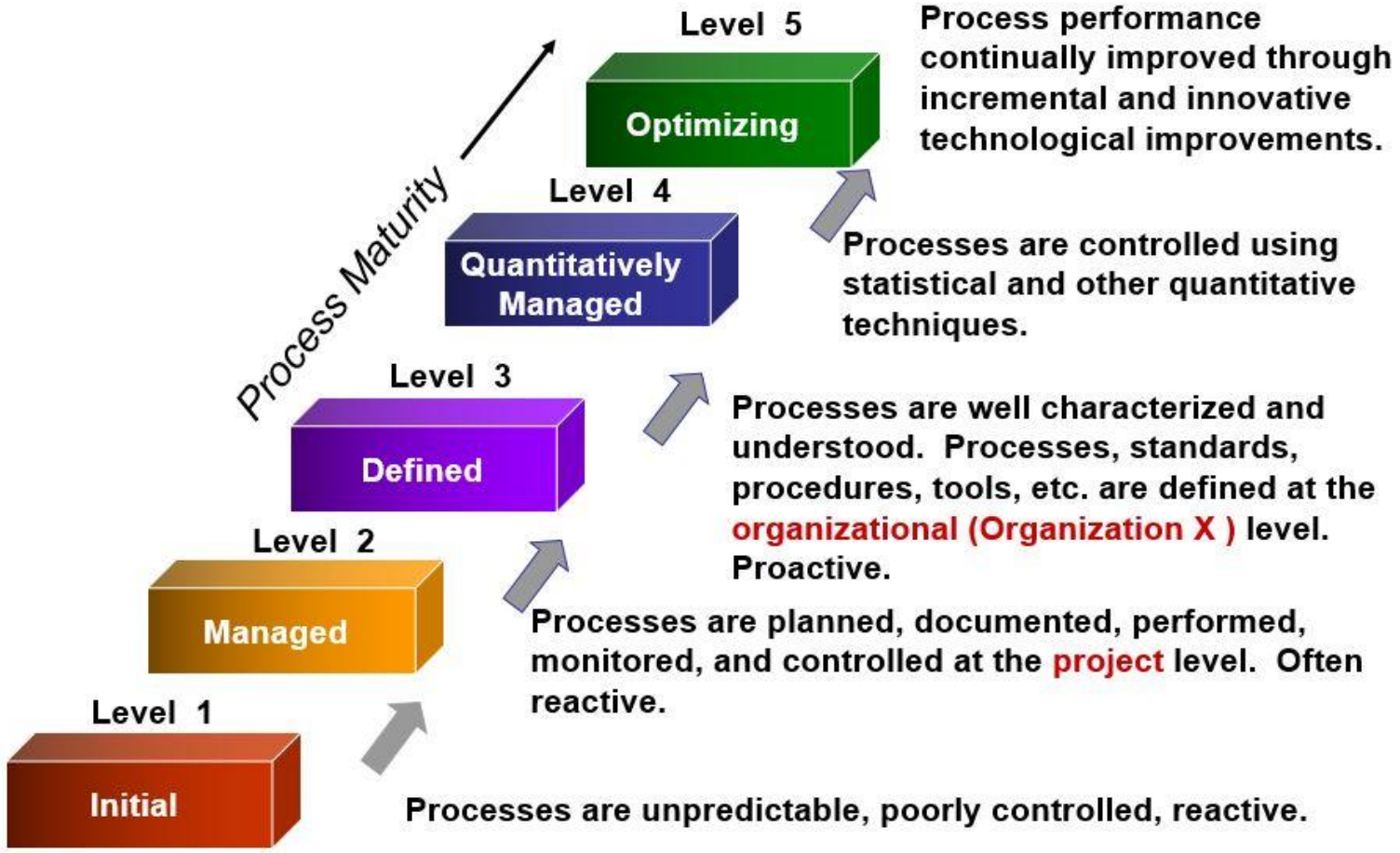
Capability Maturity Model (CMM)

- The Capability Maturity Model (CMM) is a methodology used to develop and refine an organization's software development process.
- CMM was developed and is promoted by the Software Engineering Institute (SEI), a research and development center sponsored by the U.S. Department of Defense (DOD) and now part of Carnegie Mellon University.
- The model describes a five-level evolutionary path of increasingly organized and systematically more mature processes.

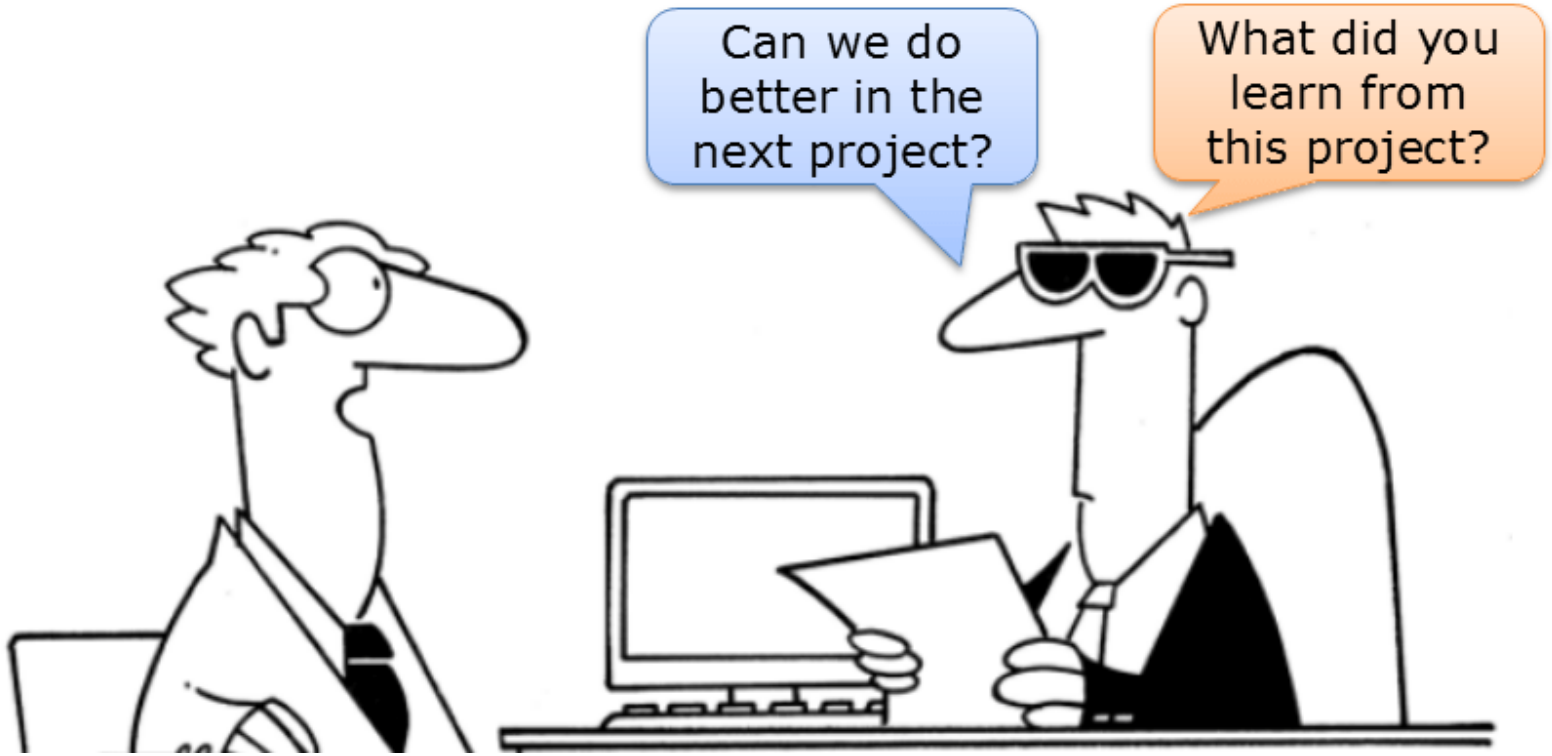
1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimizing



Capability Maturity Model (CMM)



Test Process Improvement (TPI)



Test Process Improvement (TPI)

- In software development process developers face issues like,
  1. Testing took much time than expected.
  2. Low quality of testing process.
  3. The project team is not well managed.

- Many organizations realize that improving the test process can solve these problems.
- Learning from past mistakes can help in improving test management process.

Implement Test Process Improvement

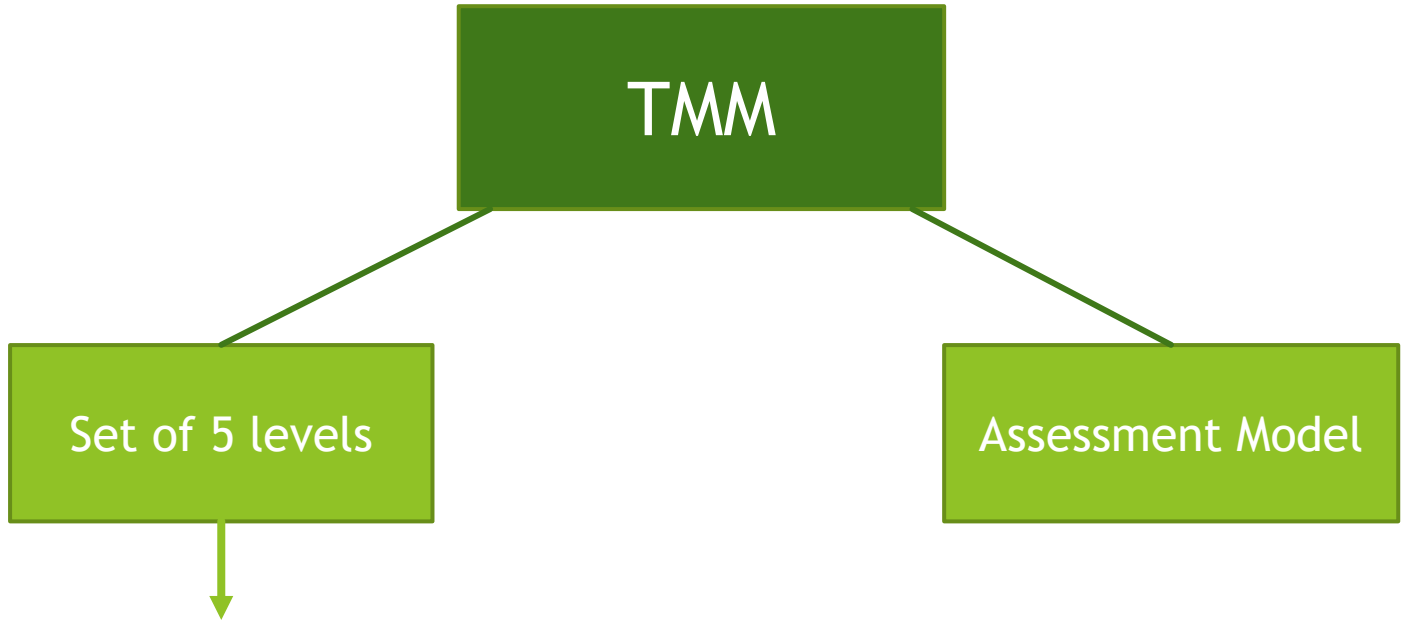


- The Test Manager can follow the **PDCA** model.
- PDCA (Plan-Do-Check-Act) is a four-step management method used in business for control and continuous improvement of process.

Test Maturity Model (TMM)

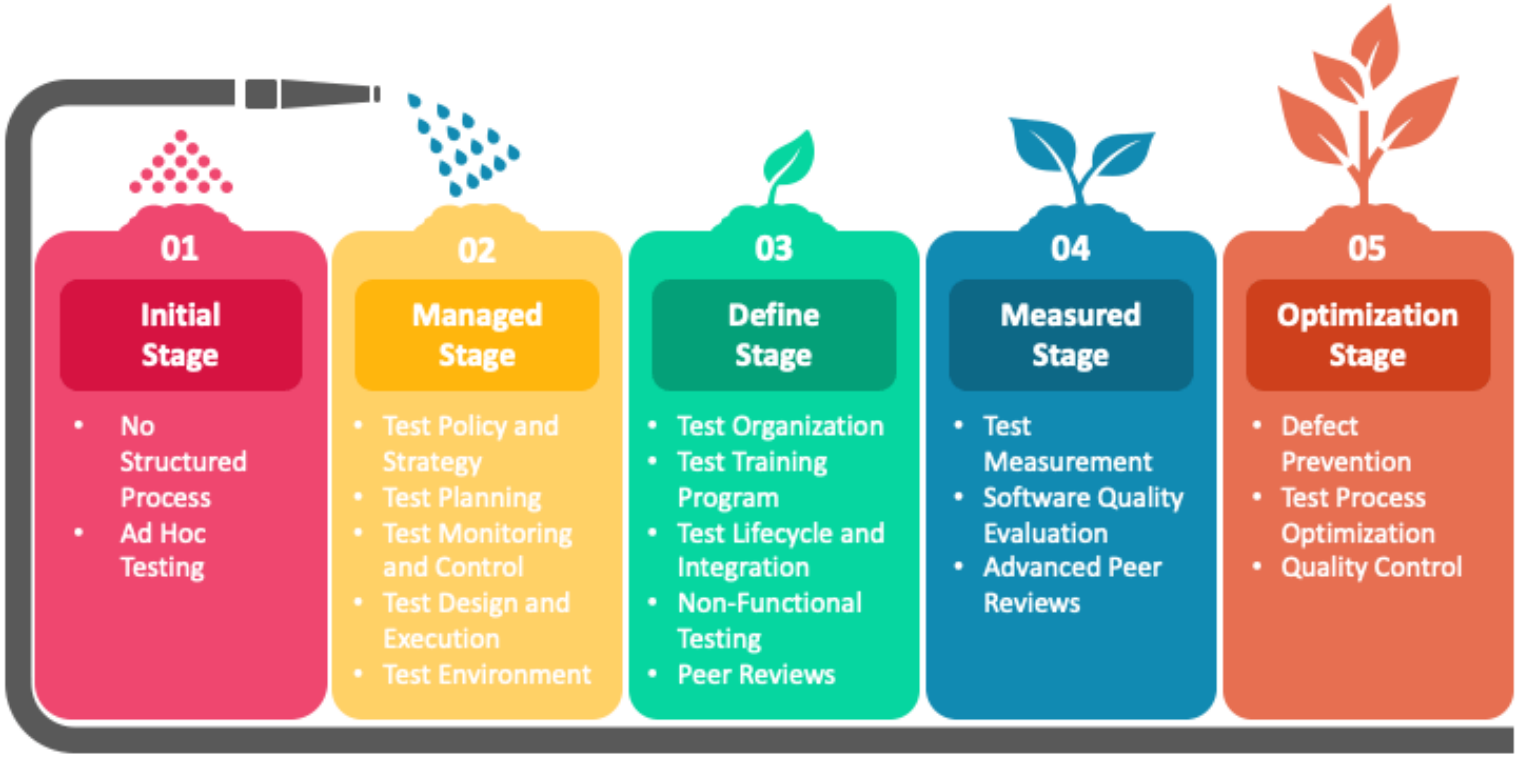
- Testing Maturity Model (TMM) in Software Testing is a framework for evaluating the maturity of software testing processes.
- The purpose of using testing maturity model is identifying maturity and providing targets to improve the software testing process to achieve progress.
- It can be complemented with any process improvement model or can be used as a stand-alone model.
- The Test Maturity Model (TMM) is based on the Capability Maturity Model (CMM).
- It was first developed by the Illinois Institute of Technology.
- It is a detailed model for test process improvement.

Test Maturity Model (TMM)



1. Initial
2. Defined
3. Integrated
4. Management & Measurement
5. Optimized

Levels of Test Maturity Model(TMM)



Benefits of Test Maturity Model

- The process is organized as each level is well defined all the deliverable are achieved.
- All the codes are reviewed, and test plans are properly executed so, its leads to no contradiction and therefore the requirements are clear.
- Quality of the software is assured as the testing procedures are integrated with all the phases of the software lifecycle.
- Risk are reduced and time is saved.

TMM vs CMM

Test Maturity Model	Capability Maturity Model
1. It is used specifically for testing the software testing model quality.	It is used specifically for determining the maturity of organizational software processes.
2. It is used as a framework along with the Capability Maturity Model.	It controls the software development procedures.

Activity

Compare and contrast three different software process models: a workflow model, a dataflow model, and a role/action model. Define each model and create a simple example that illustrates how it works.

Limit your work to 1-2 pages

THANK YOU