Lesson 01

Introduction to Software Testing (P2)

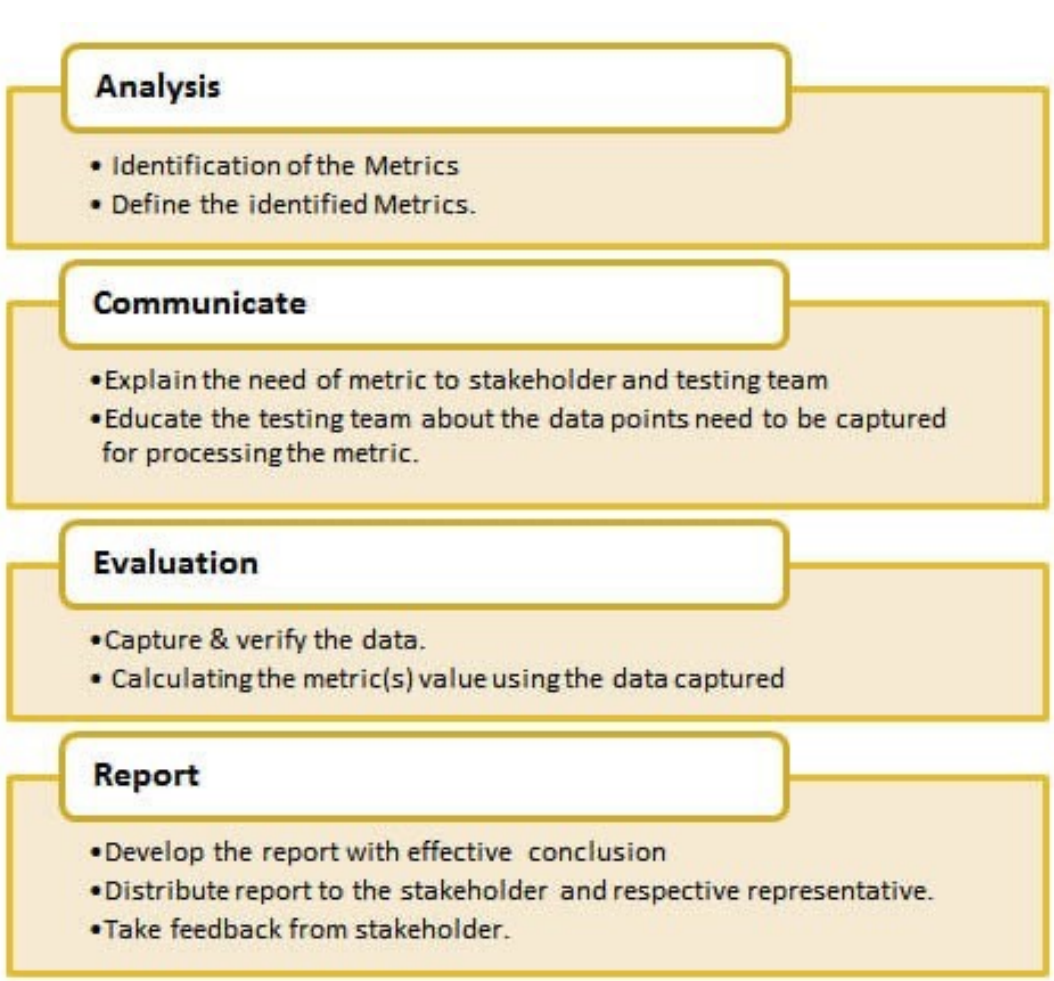
Outline

- Basic Concept
  - Test Objectives
  - Test Case
  - Test Module
  - Test Suite
- Test Scope, Test Strategy, Test Plan
- Test Metrics, Dashboard, Test Design and Process
- Test Report

Test Metrics

- **Test Metrics** are quantitative measures used to assess the effectiveness, quality, and progress of software testing activities. These metrics provide insights into the performance of testing processes, helping stakeholders understand the status and outcomes of testing efforts.
- Test metrics are essential for identifying issues, improving testing strategies, and ensuring that the software meets quality standards.

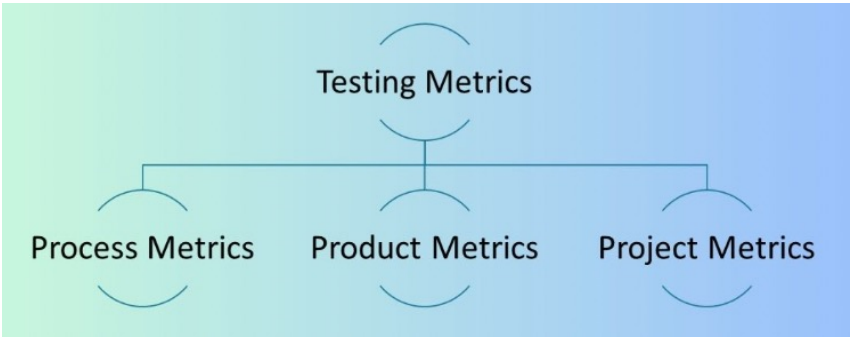
Test Metrics - Metrics Life Cycle



Quantitative Metrics

- The number of tests executed
- Passed test cases percentage
- Failed test cases percentage.
- Blocked test cases percentage
- Accepted defects percentage
- Rejected defects percentage.
- Deferred defects percentage
- Critical defects percentage
- High-priority defects percentage
- Medium-priority defects percentage
- Low-priority defects percentage
- Planned testing hours
- Actual testing hours spent

Test Metrics



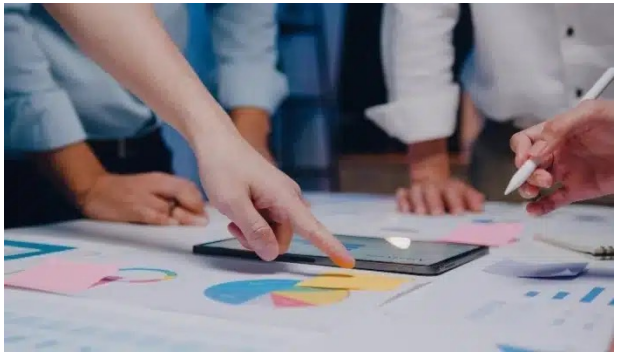
- **Types of Test Metrics**
  1. **Process Metrics**: Evaluate the efficiency of the testing process.

     Example: Test Coverage = (Number of tested requirements / Total number of requirements) * 100
  2. **Product Metrics**: Measure the quality of the product being tested.

     Example: Defect Density = (Number of defects found / Size of the software module) * 1000
  3. **Project Metrics**: Focus on the overall progress and management of the testing phase in a project.

     Example: Test Case Execution Status = (Number of test cases executed / Total number of test cases) * 100

Test Metrics



- **Examples of Common Test Metrics**

  1.Test Coverage:

  Definition: Measures the percentage of the software that has been tested.

  Formula:

$$\text{Test Coverage} = \left(\frac{\text{Number of items tested}}{\text{Total number of items}} \right) \times 100$$

  Example: If 80 out of 100 requirements are tested, test coverage is 80%.

Test Metrics

- **Examples of Common Test Metrics**

  2. Defect Density:

  Definition: Number of defects identified in a software module per unit of size (e.g., per 1000 lines of code).

  Formula:

$$\text{Defect Density} = \frac{\text{Total Defects}}{\text{Total Size of Software (e.g., KLOC)}}$$

  Example: If there are 10 defects in 2000 lines of code, the defect density is 5 per 1000 lines of code.

Test Metrics

- **Examples of Common Test Metrics**

  3. Defect Removal Efficiency (DRE):

  Definition: Measures the effectiveness of defect removal during testing.

  Formula:

$$\text{DRE} = \left(\frac{\text{Defects removed during testing}}{\text{Total defects (including those found in production)}} \right) \times 100$$

  Example: If 90 defects were found during testing and 10 were found after release (total 100), the DRE is 90%.

Test Metrics

- **Examples of Common Test Metrics**

  4. Test Case Execution Status:

  Definition: Tracks the progress of executed test cases..

  Formula:

$$\text{Execution Status} = \left(\frac{\text{Executed Test Cases}}{\text{Total Test Cases}} \right) \times 100$$

  Example: If 75 out of 100 test cases are executed, the execution status is 75%.

Test Metrics

- **Examples of Common Test Metrics**

  5. Mean Time to Detect (MTTD):

  Definition: Average time taken to detect defects after a particular phase.

  Formula:

$$\text{MTTD} = \frac{\text{Total Detection Time}}{\text{Number of Defects Found}}$$

  Example: If it takes a total of 500 hours to find 50 defects, the MTTD is 10 hours.

Test Metrics

- **Examples of Common Test Metrics**

  6. Mean Time to Repair (MTTR)

  Definition: Average time taken to fix a defect.

  Formula:

$$\text{MTTR} = \frac{\text{Total Repair Time}}{\text{Number of Defects Fixed}}$$

  Example: If it takes a total of 200 hours to fix 20 defects, the MTTR is 10 hours.

Test Metrics

- **Examples of Common Test Metrics**

  7. Test Case Pass Rate

  Definition: Percentage of test cases that passed during execution.

  Formula:

$$\text{Pass Rate} = \left(\frac{\text{Number of passed test cases}}{\text{Total number of executed test cases}} \right) \times 100$$

  Example: If 80 out of 100 executed test cases passed, the pass rate is 80%.

Test Metrics

- **Importance of Test Metrics**
  - **Quality Assurance**: Helps measure the quality of the product and ensures that it meets customer requirements.
  - **Process Improvement**: Identifies bottlenecks and areas where testing processes can be optimized.
  - **Risk Management**: Assesses risks based on testing progress and defect trends.
  - **Project Tracking**: Provides clear visibility into testing progress, helping project managers make informed decisions.
- Test metrics are valuable tools in assessing both the performance of testing activities and the quality of the software product.

Test Metrics

- **Example : banking application**.
- The team has completed the testing phase, and they are using test metrics to evaluate the effectiveness of their testing efforts
- **Scenario:**
- The team is testing a new **online banking system** that includes features like user login, funds transfer, bill payments, and transaction history.

1. **Total Number of Requirements**: 120
2. **Number of Tested Requirements**: 100
3. **Total Lines of Code (KLOC)**: 50,000
4. **Total Number of Defects Found During Testing**: 150
5. **Number of Defects Found After Release**: 20
6. **Total Number of Test Cases**: 200
7. **Test Cases Executed**: 180
8. **Test Cases Passed**: 150
9. **Total Time to Detect Defects**: 3000 hours
10. **Total Time to Repair Defects**: 1500 hours

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

1. **Test Coverage**:
   - **Formula**:

$$\text{Test Coverage} = \left(\frac{\text{Number of tested requirements}}{\text{Total number of requirements}} \right) \times 100$$

   - **Calculation**:

$$\text{Test Coverage} = \left(\frac{100}{120} \right) \times 100 = 83.33\%$$

   - **Interpretation**: 83.33% of the application's requirements have been tested.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

2. **Defect Density**:
   - **Formula**:

$$\text{Defect Density} = \frac{\text{Total Defects}}{\text{Total Size of Software (KLOC)}}$$

   - **Calculation**:

$$\text{Defect Density} = \frac{150}{50} = 3 \text{ defects per KLOC}$$

   - **Interpretation**: There are 3 defects for every 1000 lines of code in the application.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

3. **Defect Removal Efficiency (DRE)**:
   - **Formula**:

$$\text{DRE} = \left(\frac{\text{Defects removed during testing}}{\text{Total defects (including those found in production)}} \right) \times 100$$

   - **Calculation**:

$$\text{DRE} = \left(\frac{150}{150 + 20} \right) \times 100 = \left(\frac{150}{170} \right) \times 100 = 88.24\%$$

   - **Interpretation**: The testing process was able to detect and remove 88.24% of defects before the software was released.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

4. **Test Case Execution Status**:
   - **Formula**:

$$\text{Execution Status} = \left(\frac{\text{Executed Test Cases}}{\text{Total Test Cases}} \right) \times 100$$

   - **Calculation**:

$$\text{Execution Status} = \left(\frac{180}{200} \right) \times 100 = 90\%$$

   - **Interpretation**: 90% of the total test cases have been executed.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

5. **Mean Time to Detect (MTTD)**:
   - **Formula**:

$$\text{MTTD} = \frac{\text{Total Detection Time}}{\text{Number of Defects Found}}$$

   - **Calculation**:

$$\text{MTTD} = \frac{3000 \text{ hours}}{150} = 20 \text{ hours}$$

   - **Interpretation**: On average, it takes 20 hours to detect a defect.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

6. **Mean Time to Repair (MTTR)**:
   - **Formula**:

$$\text{MTTR} = \frac{\text{Total Repair Time}}{\text{Number of Defects Fixed}}$$

   - **Calculation**:

$$\text{MTTR} = \frac{1500 \text{ hours}}{150} = 10 \text{ hours}$$

   - **Interpretation**: It takes an average of 10 hours to repair a defect once it is detected.

Test Metrics

- **Example :** Applying Test Metrics on **banking application**

7. **Test Case Pass Rate**:
   - **Formula**:

$$\text{Pass Rate} = \left(\frac{\text{Number of passed test cases}}{\text{Total number of executed test cases}} \right) \times 100$$

   - **Calculation**:

$$\text{Pass Rate} = \left(\frac{150}{180} \right) \times 100 = 83.33\%$$

   - **Interpretation**: 83.33% of the executed test cases have passed.

Test Metrics

- **Example :**
- **Summary of Metrics for the Banking Application:**
  - **Test Coverage**: 83.33%
  - **Defect Density**: 3 defects per KLOC
  - **Defect Removal Efficiency**: 88.24%
  - **Test Case Execution Status**: 90%
  - **Mean Time to Detect (MTTD)**: 20 hours
  - **Mean Time to Repair (MTTR)**: 10 hours
  - **Test Case Pass Rate**: 83.33%
- The high **Defect Removal Efficiency** and moderate **Test Coverage** suggest the testing process is effective, but further testing might be needed to cover the remaining untested requirements. The **Defect Density** is also a good indicator that the code quality is fairly stable, but additional refinements may be required to improve the overall quality.

Test Dashboard

- A **Test Dashboard** is a visual representation of key metrics and data related to the software testing process. It provides stakeholders with a clear, real-time overview of testing progress, quality metrics, and potential risks.
- A well-designed test dashboard helps teams track performance, identify issues early, and make informed decisions during the software development lifecycle.

Test Dashboard

- **Key Features of a Test Dashboard:**
  1. **Real-Time Data**: Displays up-to-date information on testing activities and progress.
  2. **Key Metrics Visualization**: Summarizes important testing metrics such as test execution status, defect count, and test case pass/fail rates.
  3. **Customizable Views**: Allows different stakeholders to focus on specific areas (e.g., management might focus on overall progress, while testers focus on detailed defect data).
  4. **Alerts and Notifications**: Flags critical issues or areas that need immediate attention.
  5. **Drill-Down Capabilities**: Users can click on data points to see more detailed reports or specific test cases.

Test Dashboard

- **Common Components of a Test Dashboard:**
  1. **Test Case Execution Status**: Displays the progress of test cases (e.g., how many have been executed, passed, failed, or are blocked).
  2. **Defect Summary**: Shows the number of defects found, their severity, and their current status (open, fixed, closed, etc.).
  3. **Test Coverage**: Visualizes the percentage of requirements or features that have been tested.
  4. **Test Case Pass/Fail Rate**: Provides a breakdown of how many test cases have passed, failed, or are still in progress.
  5. **Defect Trends**: Tracks how many defects are being reported over time, showing whether the defect rate is improving or worsening.
  6. **Mean Time to Detect (MTTD) and Mean Time to Repair (MTTR)**: Displays the average time taken to detect and resolve defects.
  7. **Risk Indicators**: Flags high-risk areas, such as components with many open defects or low test coverage.

Test Dashboard

- **1. Test Execution Dashboard :** provides real-time insight into the status of the test cases being executed in a project.

  Metrics:

  - Number of test cases planned vs. executed
  - Test case pass rate
  - Number of blocked test cases
  - Time spent on test execution

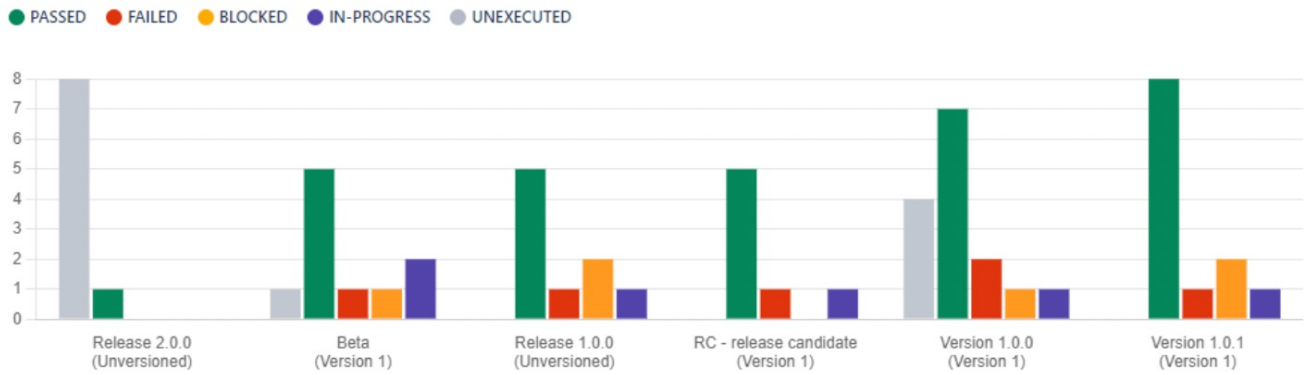
- **Visualization**:
  - Bar chart or pie chart showing test case execution status (e.g., passed, failed, in-progress)
  - A trend line showing daily or weekly test case execution progress

Test Dashboard

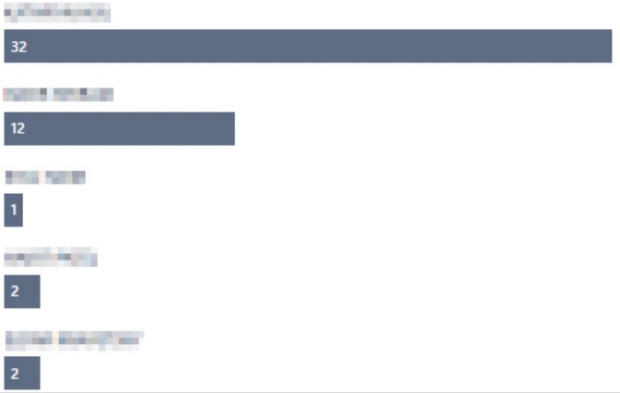
- **1. Test Execution Dashboard**
- **Example Visualization**:
  - **Pie Chart**: Shows the percentage of test cases passed, failed, and blocked.
  - **Bar Chart**: Tracks the number of executed test cases per module (e.g., Login, Transactions, Bill Payment)

Test execution result by cycle

Stacked Clustered

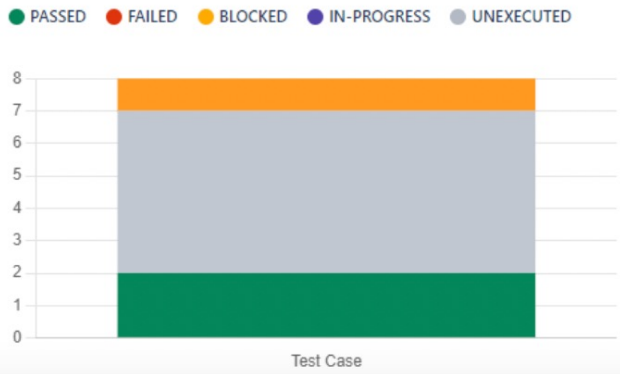


Test execution by tester



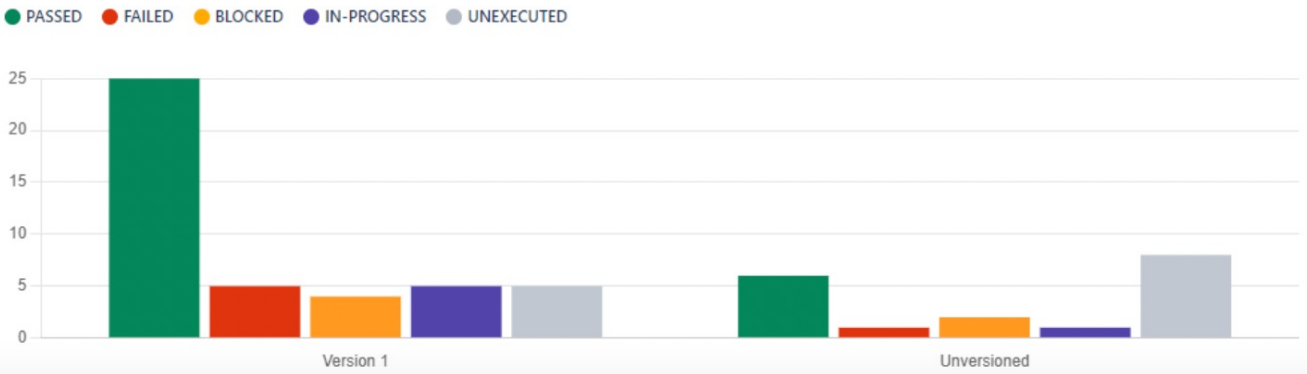
Test execution by component

Stacked Clustered



Test execution result by version

Stacked Clustered



Test Dashboard

2. Defect Summary Dashboard tracks all the defects raised during testing, helping teams prioritize and manage defect resolution.

- **Metrics**:
  - Total number of defects
  - Defects by severity (e.g., critical, major, minor)
  - Defects by status (e.g., open, in progress, fixed, closed)
  - Average time to fix defects (MTTR)
- **Visualization**:
  - Stacked bar chart showing defects by severity and status
  - Heat map to identify high-defect areas in the system

Test Dashboard

2. Defect Summary Dashboard

- **Example Visualization**:
  - **Stacked Bar Chart**: Displays the number of open, fixed, and closed defects across different modules.
  - **Heat Map**: Highlights which features or components have the highest defect count.

Defect management activity tracking dashboard

This slide illustrates summary dashboard for defect planning for project management. It includes defect management, defects by project, defects by type, defects by status, defects by reporter, etc.

Defect Management

Projects

Components

Escaped Defects

▲ 3

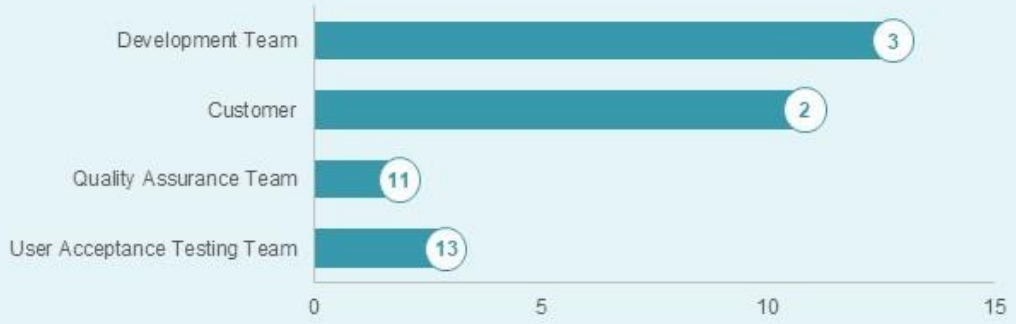
Current Month	Last Month
4	5

Defects Gap Percentage

56.55%

Defects by Project

Defects by Reporter



Defects by Status

Defects by Type

Total Defects Over Time by Priority

This graph/chart is linked to excel, and changes automatically based on data. Just left click on it and select "Edit Data".

Test Dashboard

3. Test Coverage Dashboard

- A **Test Coverage Dashboard** tracks how much of the software's functionality has been covered by tests, ensuring that all critical areas are tested.
- **Metrics**:
  - Percentage of requirements covered by tests
  - Code coverage (unit tests, integration tests, etc.)
  - Feature coverage (e.g., which features have been tested)
- **Visualization**:
  - A donut chart showing overall test coverage
  - A table listing features with the percentage of tests executed

Test Dashboard

3. Test Coverage Dashboard

- **Example Visualization**:
- **Donut Chart**: Shows that 85% of the requirements are covered by tests.
- **Table**: Lists all the features with corresponding test coverage percentages (e.g., 100% for Login, 75% for Fund Transfers).

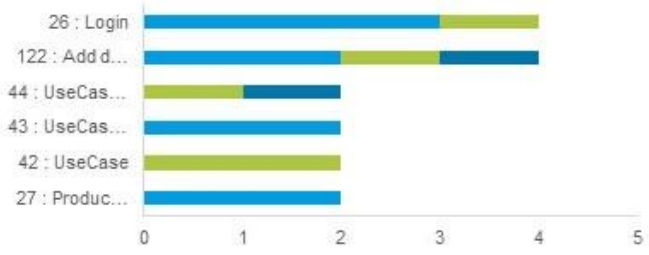
Determine Software Testing Initiatives in DevOps Dashboard

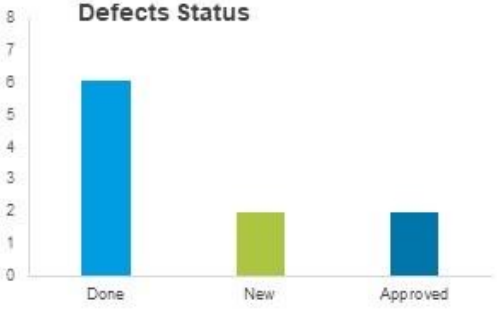
This slide provides information regarding essential software testing initiatives in DevOps dashboard in terms of test case status, defect status, etc.\

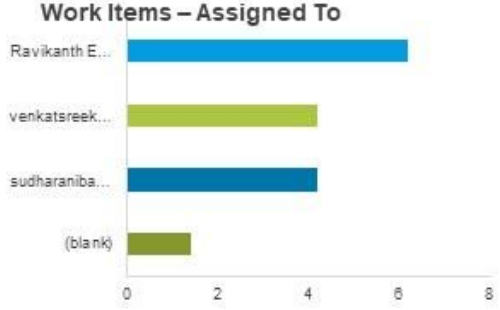
Requirement Base Suite – Chart

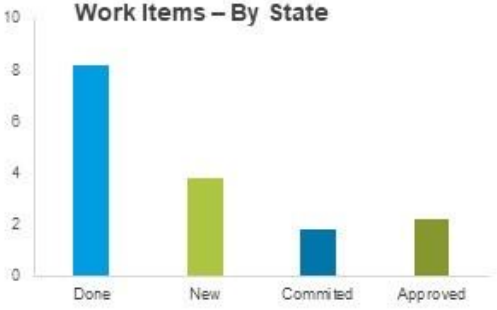
	Passed	Not Run	Failed	Total
26 : Login ...	2	1	0	4
122 : Add D...	2	1	1	4
44 : Use Cas...	0	1	1	2
43 : Use Ca...	2	0	0	2
42 : Use Ca...	0	1	0	1
27 : Produc...	2	0	0	2
Total	9	5	2	16

Requirement Base Suite – Charts







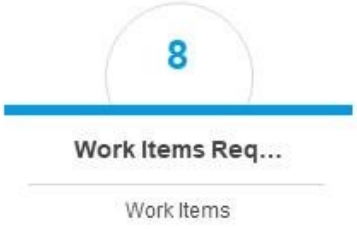


Requirements Linked Bugs (2)

ID	Work...	Title	Assi...	State
44	Produ...	UseCase 3	Venka...	App...
122	Produ...	Add Description o...	Sudha...	Co...

Requirements Linked to Test Case (6)

ID	Work...	Title	Assi...	State
26	Produ...	Login Functionalit...	Venka...	Co...
27	Produ...	Product Search fu...	Sudha...	Done
42	Produ...	Use Case 1	Ravika...	New



This graph/chart is linked to excel, and changes automatically based on data. Just left click on it and select "Edit Data".

Test Dashboard

4. Defect Trends Dashboard

- A **Defect Trends Dashboard** helps track how defects are evolving over time, showing whether the software's quality is improving or deteriorating as testing progresses.
- **Metrics**:
  - Number of defects reported over time (e.g., weekly, monthly)
  - Defect fix rate over time
  - Defect reopen rate
- **Visualization**:
  - Line graph showing the number of defects opened vs. closed over time
  - Pie chart showing defects by severity

Test Dashboard

4. Defect Trends Dashboard

- **Example Visualization**:
- **Line Graph**: Displays the trend of defects reported over the past four weeks. If the line is going down, it indicates fewer new defects are being reported.
- **Pie Chart**: Shows the distribution of defects by severity, e.g., 40% critical, 30% major, and 30% minor.

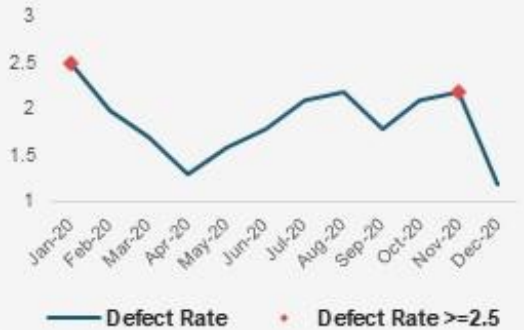
2.0%
Defect Rate

5.0
Lead Time (days)

85.5%
On Time Supplies

62.1%
Defects without Impact

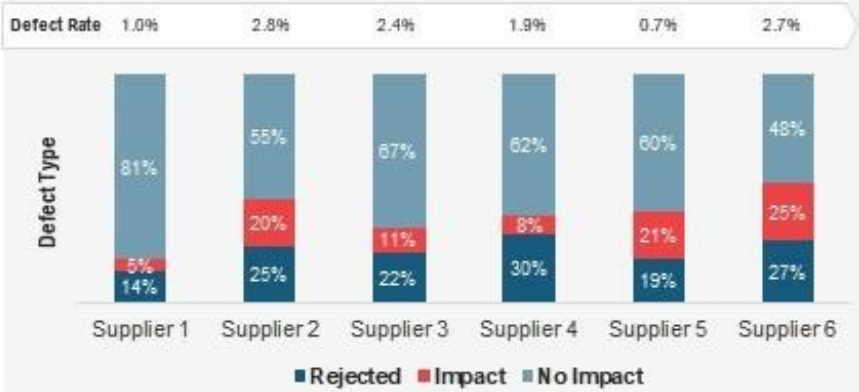
Supplier Defect Rate



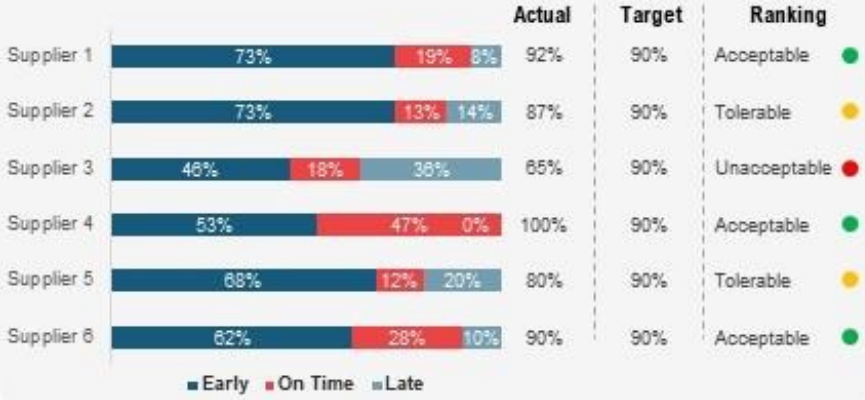
Supplier Availability

Lead Time (in Days)

Supplier Defect Rate & Defect Type



Delivery Time



Test Dashboard

5. Risk Management Dashboard

- A **Risk Management Dashboard** helps teams identify areas of the project that are high-risk, either due to low test coverage or high defect counts.
- **Metrics**:
  - Modules with low test coverage
  - Modules with high defect density
  - Defects not fixed within SLA (Service Level Agreement)
- **Visualization**:
  - Risk matrix showing high-risk areas (e.g., high-defect areas with low coverage)
  - Heat map indicating risky components

Test Dashboard

5. Risk Management Dashboard

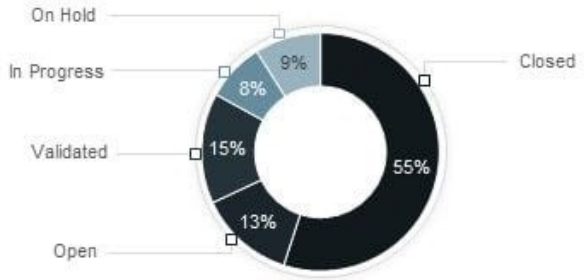
- **Example Visualization**:
- **Risk Matrix**: Plots different modules on a risk matrix, with defect density on one axis and test coverage on the other.
- **Heat Map**: Shows the risk levels for each feature based on the number of defects and coverage.

Agile sprint defect management summary report

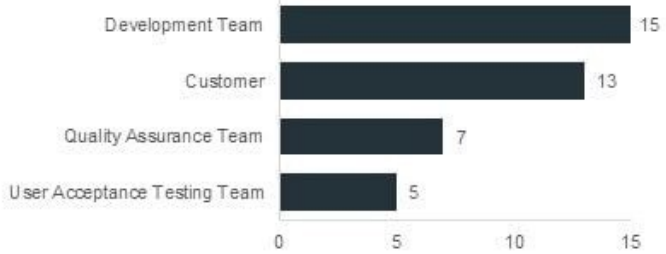
This slide illustrates summary dashboard for sprint defect planning for project management. It includes agile defect management, defects by project, bug fix time, defects by type, defects by status, defects by reporter, etc.

Agile Defect Management

Defects by Status

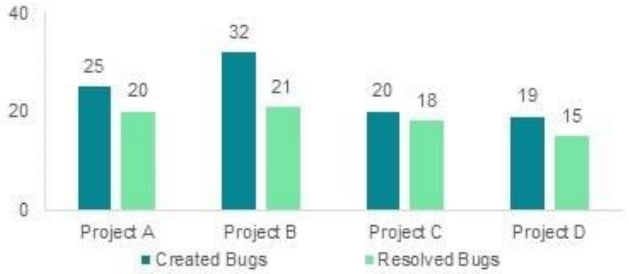


Defects by Reporter



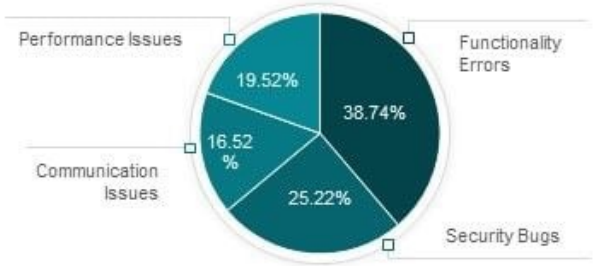
Total Defects Over Time by Priority

Defects by Project



Bug Fix Time

Defects by Type



Projects

Project A

Components

All

Escaped Defects

3

Current Month	Last Month
8	5

Defects Gap Percentage

62.21%

This graph/chart is linked to excel, and changes automatically based on data. Just left click on it and select "Edit Data".

Test Dashboard

Tools for Creating Test Dashboards:

- Several tools offer test dashboard functionalities, including:
- **Jira with Zephyr**: Allows teams to create test execution dashboards, defect tracking dashboards, and test coverage reports.
- **TestRail**: Provides built-in dashboards for test case management and defect reporting.
- **Azure DevOps**: Offers customizable dashboards with real-time reporting of testing metrics and defect management.
- **Selenium with Jenkins**: Can be integrated with reporting tools like Allure or ExtentReports to create custom test dashboards.
- **Power BI/Tableau**: These business intelligence tools can visualize test data pulled from testing management systems like Jira, allowing more flexibility in designing test dashboards.

Test Dashboard

- **Example of a Test Dashboard:**
- If a team is working on an **e-commerce application**, a test dashboard might include:
  1. **Test Execution Status**: A pie chart showing that 70% of test cases have passed, 20% failed, and 10% are blocked.
  2. **Defect Summary**: A bar chart showing that the payment gateway has 5 critical defects, while the product catalog has 3 minor defects.
  3. **Test Coverage**: A donut chart showing that 85% of the functionality is covered by test cases.
  4. **Defect Trends**: A line graph showing that the number of defects opened per week has decreased from 20 to 5 in the last month.

Test Design and Process

- **Test Design** is the process of creating a set of test cases or test procedures that evaluate whether the software system meets its requirements.
- It involves defining test objectives, identifying the necessary test data, conditions, and scenarios, and preparing detailed test cases to verify that the system works as expected.

There are a dozen of test design techniques you can use, but let's focus on the most popular ones:

- Equivalent Class Partitioning.
- Boundary Value Analysis.
- State Transition.
- Pairwise Testing.
- Error Guessing.

Test Design and Process

- **Test Design**

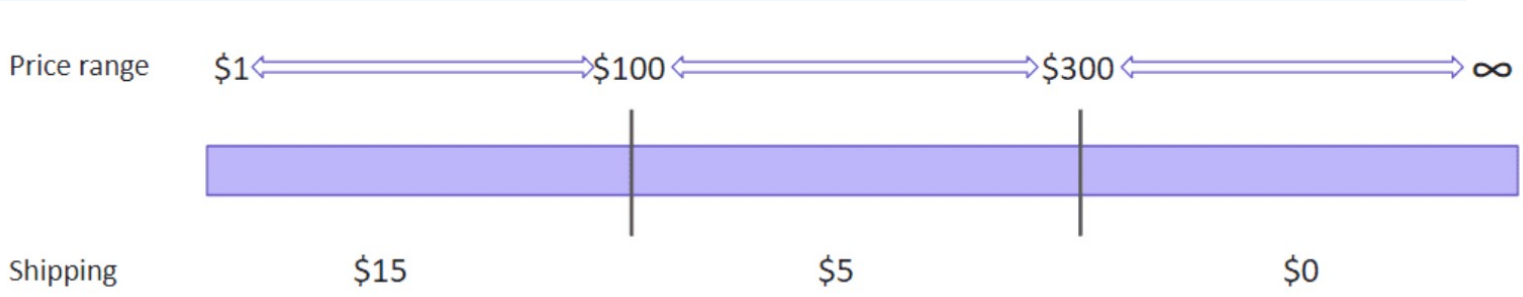
AN EXAMPLE OF EQUIVALENT CLASS PARTITIONING

Let's say, there is an online store that offers different shipping rates depending on a cart price. For example:

1. The shipping price for orders below $100 is $15.
2. The shipping price for orders over $100 is $5.
3. Free shipping on orders over $300.

We have the following price ranges to work with:

1. From $1 to $100.
2. From $100 to $300.
3. $300 and higher.



If you use the equivalent class partitioning technique, you get three sets of data to test:

1. From $1 to $100:

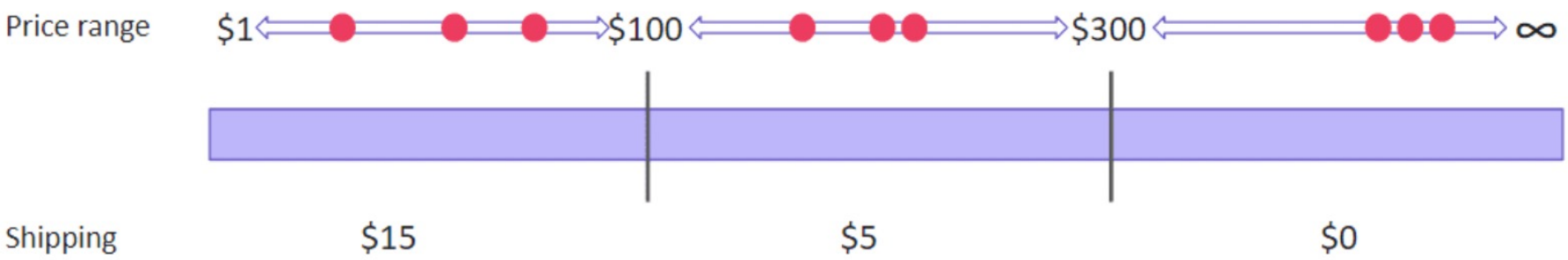
– valid boundary conditions: any price in the range from 1 to 99.99;
– invalid boundary conditions: any price below 1 or above 99.99;

2. From $100 to $300:

– valid boundary conditions: any price in the range from 100 to 299.99;
– invalid boundary conditions: any price below 100 or above 299.99;

3. $300 and higher:

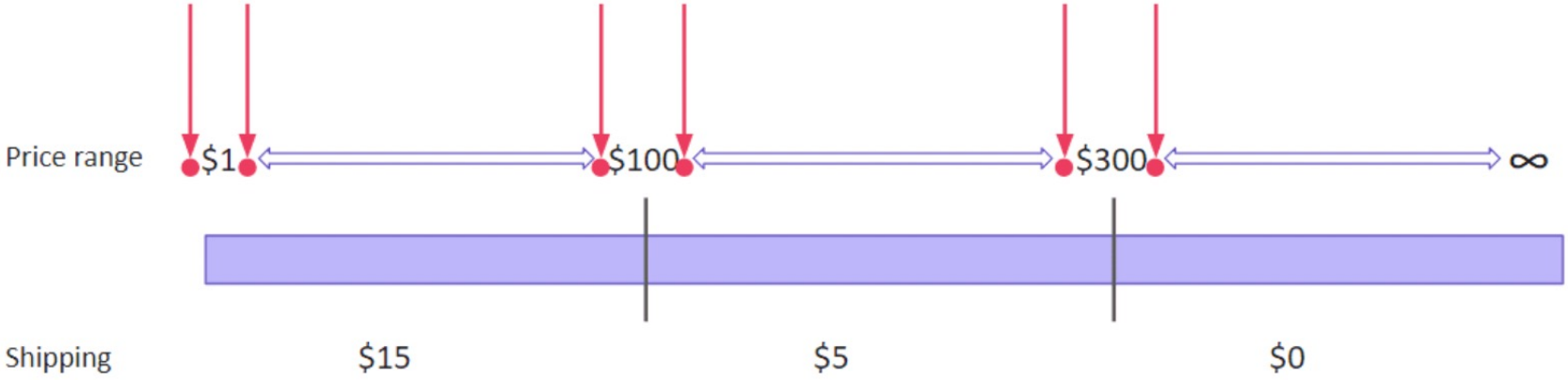
– valid boundary conditions: any price above 299.99;
– invalid boundary conditions: any price below 300.



AN EXAMPLE OF BOUNDARY VALUE ANALYSIS

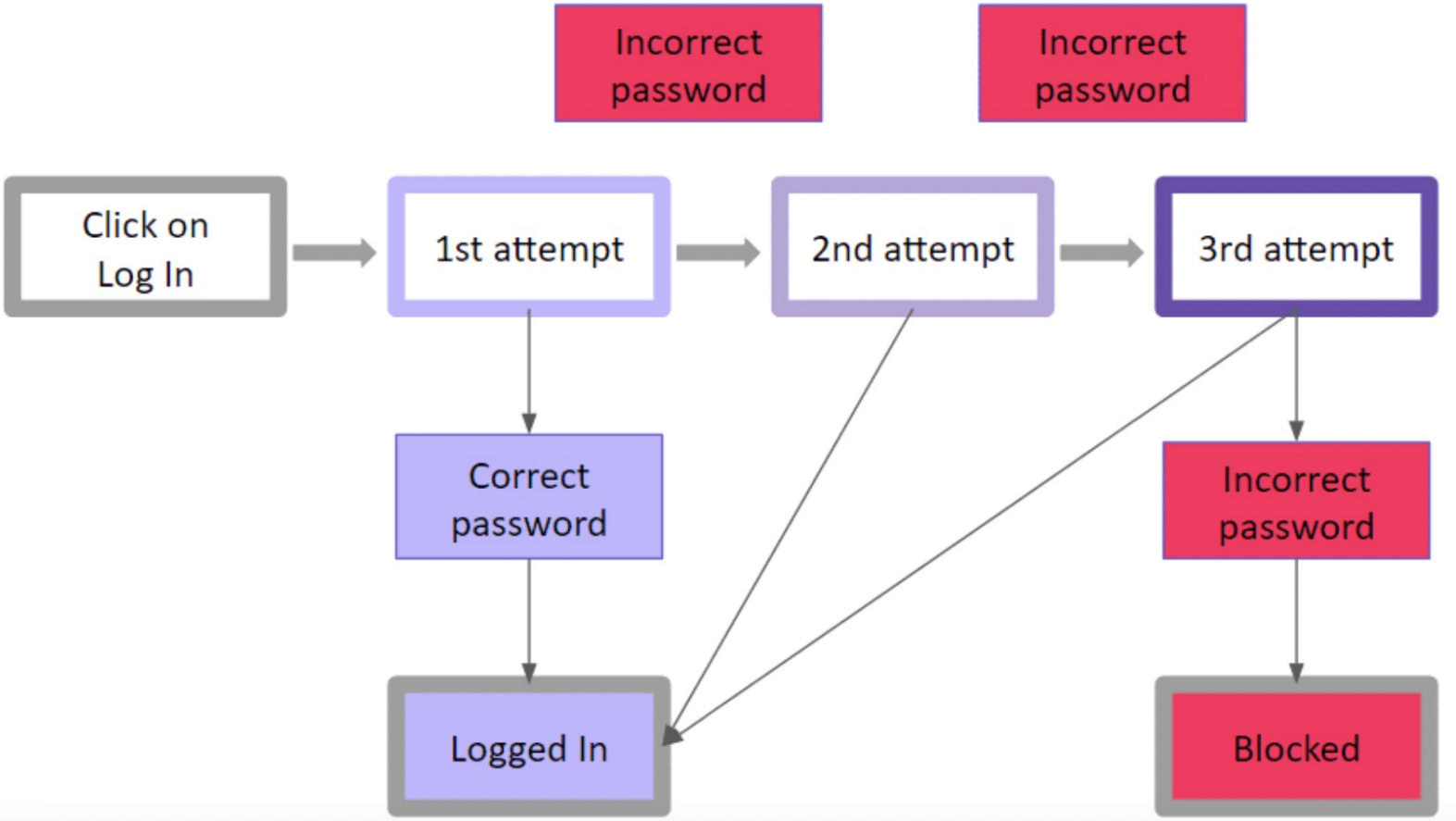
Let's take the previous scenario with varying shipping rates. We have the same data but a different approach to using it. Assuming that errors are the most likely to occur at the boundaries, we test only the 'boundary' numbers:

1. From $1 to $100:
– valid boundary conditions: 1.00, 1.01, 99.99;
– invalid boundary conditions: 0.99, 100.00, 100.01;
2. From $100 to $300:
– valid boundary conditions: 100.00, 100.01, 299.99;
– invalid boundary conditions: 99.99, 300.00;
3. $300 and higher:
– valid boundary conditions: 300.00, 300.01;
– invalid boundary conditions: 299.99.

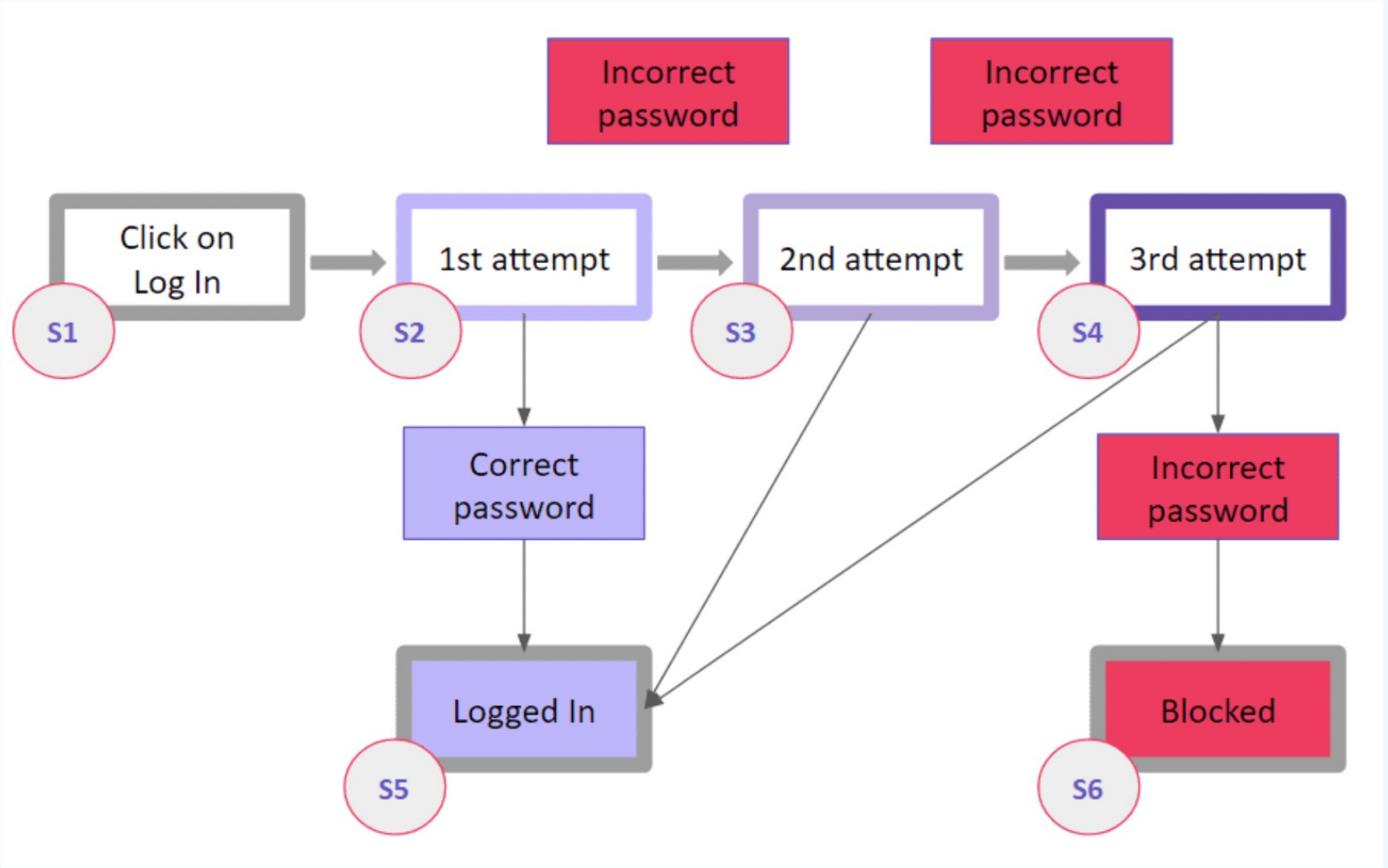


AN EXAMPLE OF STATE TRANSITION

The simplest example of the state transition is visualizing logging into an account during web or mobile app testing. Let's say, we are testing a system that offers a limited number of attempts to enter a correct password. If a user fails to enter a correct password, the system blocks the access (temporarily or permanently, it doesn't matter now). A logic diagram would look like this:



Blocks of different colors designate specific states of the system. Let's add the labels designating states, and we'll get the following:



Test Design and Process

A chart like this makes it easier to match possible inputs with expected outputs. Having a visualization right in front of your eyes helps to keep a clear head and connect the states correctly. You can later arrange the data concisely and conveniently – for example, in a table to look up to during testing:

		Correct password	Incorrect password
State 1	Click on Log In	State 5	State 2
State 2	1st attempt	State 5	State 3
State 3	2nd attempt	State 5	State 4
State 4	3rd attempt	State 5	State 6
State 5	Logged In		
State 6	Blocked		

Test Design and Process

AN EXAMPLE OF PAIRWISE TESTING

Let's say, there is a network of bakeries selling apple pies and cheesecakes online. Each is available in three sizes – small, medium, and big. The bakery offers immediate and scheduled address delivery, as well as a pick-up option. The bakery works in three cities – New York, Los Angeles, and Chicago. Also, a user can order up to three items at a time.

Order	Size	City	Quantity	Delivery	Time
Apple Pie	Small	New York	1	Address	Now
Cheesecake	Medium	Los Angeles	2	Pick-Up	Schedule
	Big	Chicago	3		

If you want to test all possible inputs, that would be 2x3x3x3x2x2=216 valid order combinations. However, checking each of those would be unreasonable. Instead, you can arrange the variables in a way that will allow covering maximum scenarios.

Test Design and Process

AN EXAMPLE OF PAIRWISE TESTING

Let's say, there is a network of bakeries selling apple pies and cheesecakes online. Each is available in three sizes – small, medium, and big. The bakery offers immediate and scheduled address delivery, as well as a pick-up option. The bakery works in three cities – New York, Los Angeles, and Chicago. Also, a user can order up to three items at a time.

Order	Size	City	Quantity	Delivery	Time
Apple Pie	Small	New York	1	Address	Now
Cheesecake	Medium	Los Angeles	2	Pick-Up	Schedule
	Big	Chicago	3		

If you want to test all possible inputs, that would be 2x3x3x3x2x2=216 valid order combinations. However, checking each of those would be unreasonable. Instead, you can arrange the variables in a way that will allow covering maximum scenarios.

To do this, you'll need to group the variables or use one of the tools that can do it for you. We used **Pairwise Online Tool** o create this example. As a result, we got 17 scenarios able to cover all 216 combinations. You can see the list of combinations below.

	Order	Size	City	Quantity	Delivery	Time
1	Apple Pie	Big	Chicago	3	Address	Now
2	Cheesecake	Big	New York	2	Address	Schedule
3	Cheesecake	Small	Los Angeles	1	Pick-Up	Now
4	Cheesecake	Medium	New York	2	Address	Schedule
5	Cheesecake	Small	Los Angeles	3	Pick-Up	Now
6	Apple Pie	Big	Los Angeles	2	Pick-Up	Now
7	Apple Pie	Small	New York	3	Address	Schedule
8	Apple Pie	Small	Chicago	2	Pick-Up	Schedule
9	Apple Pie	Medium	New York	1	Address	Now
10	Cheesecake	Medium	Chicago	1	Pick-Up	Schedule
11	Cheesecake	Medium	Los Angeles	3	Address	Schedule
12	Cheesecake	Big	New York	1	Pick-Up	Now
13	Apple Pie	Medium	New York	3	Pick-Up	Now
14	Apple Pie	Small	Los Angeles	1	Address	Schedule
15	Apple Pie	Medium	New York	2	Pick-Up	Now
16	Apple Pie	Big	Los Angeles	1	Address	Schedule
17	Apple Pie	Small	Chicago	2	Address	Now

Test Design and Process

- **Test Design**

AN EXAMPLE OF ERROR GUESSING

As a rule, QA engineers start with testing for common mistakes, such as:

- Entering blank spaces in text fields.
- Pressing the Submit button without entering data.
- Entering invalid parameters (email address instead of a phone number, etc.).
- Uploading files that exceed the maximum limit.
- ... and so on.

The more experience a QA specialist has, the more error guessing scenarios they can come up with quickly.

Test Design and Process

- **Test Design Process** typically involves the following steps:

Test Design and Process

- **Test Design Process** typically involves the following steps:
- **1. Requirements Analysis**
  - Understand the system's requirements, user stories, and functionality that need to be tested.
  - Identify the testable aspects of the system from both functional and non-functional perspectives.
- **2. Test Planning**
  - Define the scope of testing (features to be tested and features not to be tested).
  - Identify the test levels (unit, integration, system, etc.).
  - Determine the types of testing needed (e.g., functional, performance, security, usability).
  - Define the objectives, resources, tools, and timelines for the test.

Test Design and Process

- **Test Design Process** typically involves the following steps:
- **3. Test Case Design**
  - Create individual test cases based on the requirements and user stories.
  - Each test case should cover one or more conditions that need to be verified.
  - Include details such as test inputs, expected results, and execution steps.
  - Organize test cases into test suites (logical grouping).
- **4. Test Data Preparation**
  - Identify or create the data necessary for executing the test cases.
  - The data should simulate realistic conditions to thoroughly test the software's functionality.
  - Test data can be input values, configuration settings, or any other values used in test execution.

Test Design and Process

- **Test Design Process** typically involves the following steps:
- **5. Test Environment Setup**
  - Set up the hardware, software, network, and configurations required for the test.
  - Ensure that the environment mimics the production environment to get reliable results.
- **6. Test Execution**
  - Execute the designed test cases on the software system.
  - Capture the actual results and compare them with the expected outcomes.

Test Design and Process

- **Test Design Process** typically involves the following steps:
- **7. Defect Reporting**
  - If any discrepancies arise between expected and actual results, defects are logged and reported.
  - Defects should be categorized by severity and priority for resolution.
- **8. Test Monitoring and Control**
  - Continuously monitor test progress to ensure it aligns with the test plan.
  - Adjust test execution strategies if issues arise or new test requirements are identified.
- **9. Test Closure**
  - Conclude testing when exit criteria (such as sufficient test coverage, bug resolution, etc.) are met.
  - Summarize test results and document lessons learned for future projects.

Test Report

- A **Test Report** is a formal document that provides a detailed summary of the testing activities, results, and overall evaluation of the software under test.
- It serves as a key communication tool between the testing team and stakeholders (e.g., developers, project managers, clients) by offering insight into the quality, risks, and readiness of the software for release.

Test Report

- A **Test Report** is a formal document that provides a detailed summary of the testing activities, results, and overall evaluation of the software under test.
- It serves as a key communication tool between the testing team and stakeholders (e.g., developers, project managers, clients) by offering insight into the quality, risks, and readiness of the software for release.

Test Report

- **Objectives of a Test Report:**
  - **Communicate Test Results**: Show whether the software meets its requirements and quality standards.
  - **Identify Issues**: Provide detailed information about defects and issues found during testing.
  - **Evaluate Software Quality**: Assess the overall stability and functionality of the system.
  - **Facilitate Decision-Making**: Help stakeholders decide whether the software is ready for release, further testing, or additional development.

Test Report

Key Elements of a Test Report:

1. **Introduction/Overview**
    - Provides a brief description of the software system or project being tested.
    - Outlines the purpose and objectives of the test.
    - May include project background, scope, and the test strategy used.
2. **Test Summary**
    - **Test Scope**: Summarizes the features or modules tested.
    - **Test Types**: Lists the types of testing conducted (e.g., functional, performance, security).
    - **Test Environment**: Describes the hardware, software, and network configurations used in the testing process.
    - **Test Tools**: Mentions the tools used, such as Selenium for automation or JIRA for bug tracking.

3. **Test Execution Summary**
    - **Total Test Cases**: The total number of test cases designed and executed.
    - **Passed Test Cases**: The number and percentage of test cases that passed.
    - **Failed Test Cases**: The number and percentage of test cases that failed.
    - **Blocked Test Cases**: Test cases that couldn't be executed due to external factors (e.g., incomplete features, environment issues).
    - **Skipped Test Cases**: Test cases not executed as they were out of scope or deprioritized.
4. **Defect Summary**
    - **Total Defects Identified**: The total number of defects found during testing.
    - **Defect Status**: A breakdown of defects by status (e.g., open, closed, fixed).
    - **Defect Severity**: Categorization of defects based on severity (e.g., critical, major, minor).
    - **Defect Priority**: Prioritization of defects based on their impact on the system.
    - **Defect Trends**: Graphs or charts showing the number of defects over time or across modules.

Test Report

5. **Test Coverage**
    - Provides a percentage of requirements or functionalities covered by testing.
    - May include details on areas not tested or partially tested (e.g., due to time constraints).
6. **Risks and Issues**
    - Describes any risks or issues encountered during the testing process.
    - Includes potential risks for post-release if certain bugs were not fixed or features were not tested.
    - May mention known limitations of the system based on testing results.
7. **Recommendations**
    - Provides expert suggestions based on test results.
    - May include recommendations to postpone the release if critical defects are present or to move forward with production based on system stability.

Test Report

8. **Test Metrics**
    - Provides quantitative data to summarize testing progress and results, such as:
        - Test case execution rate (e.g., % of test cases executed vs. planned).
        - Defect density (e.g., defects found per lines of code or test cases).
        - Test pass/fail ratio.
9. **Conclusion**
    - Summarizes the overall assessment of the software's quality and readiness for release.
    - Concludes whether the software passed the testing phase, needs more work, or is ready for deployment.

Example: Lesson 01 - Test Report